Tutorial de JavaServer Faces

Tutorial de JavaServer Faces

Índice

Indice	• • • • • • • • • • • • • • • • • • • •	<u>1</u>
• • • • • • • • • •	• • • • • • • • • • • • • • • • • • • •	<u>3</u>
Capitulo 1		· • • • <u>4</u>
Primeros Pa	asos	4
1.1	Preliminares	
1.2	Características principales	
1.3	Comparativa con Tecnologías similares	
1.4	Beneficios de la Tecnología JavaServer Faces	
1.5	¿Qué es una aplicación JavaServer Faces?	
1.6	Una sencilla aplicación JavaServer Faces	
1.0	end senema apricación suvasciver races	· · · · <u>·</u>
Capitulo 2		<u>11</u>
	asos	
2.1		
	2.1.1 Instalación del JDK (Java Development Kit):	<u>11</u>
	2.1.2 Instalación de Apache Tomcat:	<u>12</u>
	2.1.3 Instalación de <i>JavaServer Faces</i>	
2.2	Entornos de desarrollo	<u>14</u>
	2.2.1 Java Studio Creator de Sun Microsystems	<u>14</u>
	2.2.2 NetBeans	<u>15</u>
	2.2.3 Eclipse-SDK	<u>16</u>
Canitula 3		18
	••••••	· · · <u>10</u>
Modelo Vist	ta Controlador en JSF	18
3.1	Introducción	
3.2	Modelo	
3.3	Vista	
3.4	Controlador	
3.5	Ciclo de vida de una página JavaServer Faces	
3.3	3.5.1 Escenarios de Procesamiento del Ciclo de Vida de una Petición .	
	3.5.2 Ciclo de Vida Estándar de Procesamiento de Peticiones	
3.6	Una sencilla aplicación <i>JavaServer Faces</i> en detalle	
3.7	Beans y páginas JSF	
5.1	3.7.1 Beans	
	3.7.2 Paginas JSF	
3.8	Navagagián	$\frac{27}{20}$

Tutorial de JavaServer Faces

3.9	Configuración Servlet	<u>29</u>
Capitulo 4		<u>31</u>
Managed Bea	ans, navegación y etiquetas básicas JSF	31
4.1	Managed Beans	
	4.1.1 Concepto	
	4.1.2 Ámbitos de los <i>beans</i>	
	4.1.3 Configuración de un <i>bean</i> a través de XML	
4.2	Navegación	
1.2	4.2.1 Concepto	
	4.2.2 Navegación estática	
	4.2.3 Navegación dinámica	
	4.2.4 Navegación avanzada	
4.3	Etiquetas básicas	
4.4	Un ejemplo completo de etiquetas básicas con JSF	
7.7	On ejemplo completo de cuiquetas basicas con 351	· · · · · 10
Capitulo 5		<u>60</u>
-	F estándares	
5.1	Introducción	
5.2	Etiquetas Core	
5.3	Etiquetas HTML	· ·
	5.3.1 Atributos comunes	
5.4	Formularios	
5.5	Campos de texto y áreas de texto	
	5.6 Botones y enlaces	
5.7	Etiquetas de selección	
	5.7.1 selectBooleanCheckbox	
	5.7.2 selectManyCheckbox	
	5.7.3 selectOneRadio	
	5.7.4 selectOneListbox	<u>76</u>
	5.7.5 selectManyListbox	· · · · <u>77</u>
	5.7.6 selectOneMenu	· · · · <u>77</u>
	5.7.7 selectManyMenu	· · · · <u>78</u>
5.8	Mensajes	
5.9	Paneles	· · · · <u>80</u>
5.10	Data Table	<u>82</u>
	5.10.1 La etiqueta h:dataTable	<u>82</u>
	5.10.2 Cabeceras y pie de tabla	<u>83</u>
	5.10.3 Componentes JSF incluidos en celdas tabla	84
	5.10.4 Editando celdas	
	5.10.5 Estilos para filas y columnas	
	5.10.6 Técnicas de scroll	
	5.10.7 Un breve ejemplo de la etiqueta h:dataTable	85

Tutorial de JavaServer Faces

Capitulo 6.		<u>90</u>
Conversión	y Validación	90
6.1	El proceso de conversión y validación	
6.2	Usando conversores estándares	
0.2	6.2.1 Conversión de números y fechas	
	6.2.2 Conversores y atributos	
	6.2.3 Mensajes de error	
	6.2.4 Cambiar el texto de los mensajes de error estándares	
	6.2.5 Un ejemplo completo usando conversores	
6.3	Usando validadores estándares	
0.3	6.3.1 Validando longitudes de cadenas de caracteres y rangos numéric	
	· · · · · · · · · · · · · · · · · · ·	
	6.3.2 Chequeando valores requeridos	
	6.3.3 Mostrando errores de validación	
	6.3.4 Un completo ejemplo de validación	
	out. On tomplote of the control of t	· · · · <u></u>
Capitulo 7.		102
Manejo de E	Eventos	102
7.1	Eventos de cambio de valor	102
7.2	Eventos de acción	108
7.3	Eventos de fase	114
Capitulo 8.		116
8.1	Organización común	
8.2	Un visor de libro	
	8.2.1 Consideraciones para la especificación JSF	
	8.2.2 Inclusión de contenido en el visor de libro	<u>118</u>

Capitulo 1 Primeros Pasos

1.1 Preliminares

El objetivo de la tecnología JavaServer Faces es desarrollar aplicaciones web de forma parecida a como se construyen aplicaciones locales con Java Swing, AWT (*Abstract Window Toolkit*), SWT (*Standard Widget Toolkit*) o cualquier otra API similar.

Tradicionalmente, las aplicaciones web se han codificado mediante páginas JSP (*Java Server Pages*) que recibían peticiones a través de formularios y construían como respuesta páginas HTML (*Hiper Text Markup Language*) mediante ejecución directa o indirecta -a través de bibliotecas de etiquetas- de código Java, lo que permitía, por ejemplo, acceder a bases de datos para obtener los resultados a mostrar amén de realizar operaciones marginales como insertar o modificar registros en tablas relacionales, actualizar un carrito de la compra, etc.

JavaServer Faces pretende facilitar la construcción de estas aplicaciones proporcionando un entorno de trabajo (*framework*) vía web que gestiona las acciones producidas por el usuario en su página HTML y las traduce a eventos que son enviados al servidor con el objetivo de regenerar la página original y reflejar los cambios pertinentes provocados por dichas acciones. En definitivas cuentas, se trata de hacer aplicaciones Java en las que el cliente no es una ventana de la clase **JFrame** o similar, sino una página HTML.

Como el lector puede imaginar, cualquier evento realizado sobre una página JSF incurre en una carga sobre la red, ya que el evento debe enviarse a través de ésta al servidor, y la respuesta de éste debe devolverse al cliente; por ello, el diseño de aplicaciones JavaServer Faces debe hacerse con cuidado cuando se pretenda poner las aplicaciones a disposición del mundo entero a través de internet. Aquellas aplicaciones que vayan a ser utilizadas en una intranet podrán aprovecharse de un mayor ancho de banda y producirán una respuesta mucho más rápida.

1.2 Características principales

La tecnología JavaServer Faces constituye un marco de trabajo (*framework*) de interfaces de usuario del lado de servidor para aplicaciones web basadas en tecnología Java y en el patrón MVC (Modelo Vista Controlador).

Los principales componentes de la tecnología JavaServer Faces son:

- Una API y una implementación de referencia para:
- Representar componentes de interfaz de usuario (UI-*User Interface*) y manejar su estado
- Manejar eventos, validar en el lado del servidor y convertir 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 facilita de forma significativa la tarea de la construcción y mantenimiento de aplicaciones web con UIs en el lado servidor. Con un mínimo esfuerzo, es posible:

- Conectar eventos generados en el cliente a código de la aplicación en el lado servidor.
- Mapear componentes UI a una página de datos en el lado servidor.
- Construir una interfaz de usuario con componentes reutilizables y extensibles.

Como se puede apreciar en la **figura 1.1**, la interfaz de usuario que se crea con la tecnología JavaServer Faces (representada por **miUI** en el gráfico) se ejecuta en el servidor y se renderiza en el cliente.

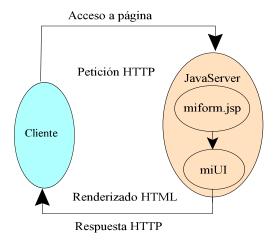


Figura 1.1 Diagrama de una aplicación JSF

En la figura no queda qué es físicamente miUI. La página JSP, **miform.jsp**, especifica los componentes de la interfaz de usuario mediante etiquetas personalizadas definidas por la tecnología JavaServer Faces. LA UI de la aplicación web (representada por miUI en la figura) maneja los objetos referenciados por la JSP, que pueden ser de los siguientes tipos:

- Objetos componentes que mapean las etiquetas sobre la página JSP.
- Oyentes de eventos, validadores y conversores registrados y asociados a los componentes.
- Objetos del modelo que encapsulan los datos y las funcionalidades de los componentes específicos de la aplicación (lógica de negocio).

1.3 Comparativa con Tecnologías similares

Al igual que Struts, JSF pretende normalizar y estandarizar el desarrollo de aplicaciones web. Hay que tener en cuenta que JSF es posterior a Struts y, por lo tanto, se a nutrido de la experiencia de éste, erradicando algunas sus deficiencias. De hecho el creador de Struts (Craig R. McClanahan) también es el líder de la especificación de JSF.

Como se ha indicado, JSF es muy similar a Struts, ya que la interfaz de usuario se implementa con páginas JSP, con la ventaja de que puede trabajar con diversas herramientas de desarrollo que pueden facilitar la elaboración de la interfaz de forma visual, pero esto es opcional, pudiendose hacer mediante editores de texto, editando cada uno de los ficheros que componen la aplicación, tal y como se ha trabajado en este tutorial. Sin embargo se abordará el

uso de estas herramientas, en concreto, Java Studio Creator, en el último capítulo.

No obstante, JSF no puede competir en madurez con Struts (en este punto Struts es claro ganador), pero sí puede ser una opción muy recomendable para nuevos desarrollos, sobre todo si todavía no tenemos experiencia con Struts.

1.4 Beneficios de la Tecnología JavaServer Faces

A continuación se explican algunos de los puntos por los que JSF es una tecnología muy interesante (incluso mas que *Struts*). Hay una serie de especificaciones que definen JSF, y que pueden encontrarse en los siguientes enlaces:

```
JSR 127 ( <a href="http://www.jcp.org/en/jsr/detail?id=127">http://www.jcp.org/en/jsr/detail?id=127</a> )
JSR 252 ( <a href="http://www.jcp.org/en/jsr/detail?id=252">http://www.jcp.org/en/jsr/detail?id=252</a> )
JSR 276 ( <a href="http://www.jcp.org/en/jsr/detail?id=276">http://www.jcp.org/en/jsr/detail?id=252</a> )
```

Una de las ventajas de que JSF sea una especificación estándar es que pueden encontrarse implementaciones de distintos fabricantes. Esto permite no vincularse exclusivamente con un proveedor concreto, y poder seleccionar el que más interese en cada caso según el número de componentes que suministra, el rendimiento de éstos, soporte proporcionado, precio, política de evolución, etc.

JSF trata la vista (la interfaz de usuario) de una forma algo diferente a lo que estamos acostumbrados en aplicaciones web, ya que este tratamiento es mucho más cercano al estilo de Java Swing, Visual Basic o Delphi, donde la programación de la interfaz se hacer a través de componentes y está basada en eventos (pulsación de un botón, cambio en el valor de un campo, etc.).

JSF es muy flexible. Por ejemplo nos permite crear nuestros propios componentes, y/o crear nuestros propios renderizadores para pintar los componentes en la forma que más nos convenga.

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 de los componentes o manejar elementos UI como objetos con estado en el servidor. La tecnología JavaServer Faces permite construir aplicaciones web que introducen realmente una separación entre el comportamiento y la presentación, separación sólo ofrecida tradicionalmente por arquitecturas UI del lado del cliente.

Separar la lógica de negocio de la presentación también permite que cada miembro del equipo de desarrollo de la aplicación web se centre en su parte asignada del proceso diseño, y proporciona un modelo sencillo de programación para enlazar todas las piezas. Por ejemplo, personas sin experiencia en programación pueden construir páginas JSF usando las etiquetas de componentes UI que esta tecnología ofrece, y luego enlazarlas con código de la aplicación sin escribir ningún *script* ni nada.

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, las APIs de JavaServer Faces se han creado directamente sobre el API *JavaServlet*. Esto nos permite, teóricamente, hacer algunas cosas avanzadas: 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, entre otras.

En definitivas cuentas, 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.

1.5 ¿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 de *servlets* de 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, (principalmente 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.

Toda aplicación JavaServer Faces debe incluir una librería de etiquetas personalizadas que define las etiquetas que representan componentes UI, así como una librería de etiquetas para controlar otras acciones importantes, como validadores y manejadores de eventos. La implementación de JavaServer Faces, de Sun proporciona estas dos librerías. La librería de etiquetas de componentes elimina la necesidad de codificar componentes UI en HTML u otro lenguaje de marcas, lo que se traduce en el empleo de componentes completamente reutilizables. Y la librería principal (*core*) hace fácil registrar eventos, validadores y otras acciones de los componentes.

Como librería de etiquetas de componentes puede usarse la librería **html_basic** incluida con la implementación de referencia de la tecnología JavaServer Faces, aunque también es posible definir una librería de etiquetas personalizadas que dibuje componentes propios o que proporcione 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 permite convertir y validar datos sobre componentes individuales e informar de cualquier error antes de que se actualicen los datos en el lado del servidor.

Debido a la división de labores que permite el diseño de la tecnología JavaServer Faces, el desarrollo y mantenimiento de una aplicación JavaServer Faces se puede realizar muy rápida y fácilmente. Abajo tenemos un listado de los roles principales de un equipo de desarrollo típico.

Aunque en muchos equipos, los desarrolladores individuales puede interpretar más de uno de esos roles, resulta muy útil considerar la tecnología JavaServer Faces desde varias perspectivas basadas en la responsabilidad principal que tiene cada participante:

- <u>Autores de páginas</u>, que utilizan un lenguaje de marcas, como HTML, para construir páginas para aplicaciones web. Cuando se utiliza la tecnología JavaServer Faces, los autores de páginas casi siempre usarán exclusivamente la librería de etiquetas.
- <u>Desarrolladores de aplicaciones</u>, que programan los objetos del modelo, los manejadores de eventos, los validadores, y la navegación de páginas. Los desarrolladores de aplicaciones también pueden proporcionar las clases de utilidad necesarias.
- <u>Escritores de componentes</u>, que tienen experiencia en programar interfaces de usuario y prefieren crear componentes personalizados utilizando un lenguaje de programación. Esta gente puede crear sus propios componentes desde cero, o puede extender los componentes estándares proporcionados por JavaServer Faces.
- Vendedores de herramientas, que proporcionan herramientas que mejoran la tecnología JavaServer Faces para hacer incluso más sencilla la construcción de interfaces de usuario en el lado servidor.

Para las aplicaciones web pueden ser muy convenientes *frameworks* como JSF, Struts, Spring, etc., pero éstos no servirán (a no ser a costa de un gran esfuerzo) para hacer portales. Para este segundo caso sería más adecuado usar gestores de contenidos como Lenya, OpenCMS, etc.

1.6 Una sencilla aplicación JavaServer Faces

Esta sección describe el proceso de desarrollo de una sencilla aplicación JavaServer Faces. Con ella se verá qué características contiene una aplicación típica JavaServer Faces.

Nuestra primera aplicación empieza con una pantalla de entrada con un *Login* y un *password*. Como vemos en la **figura 1.2**:

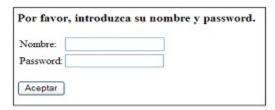


Figura 1.2 Salida del fichero ejemploBasico/index.jsp

Fichero: ejemploBasico/index.jsp

Discutiremos en detalle el contenido de este archivo más tarde . Por ahora, baste con comentar los siguientes puntos:

- Algunas de las etiquetas son etiquetas estándar de HTML: body, table, etc.
 Algunas etiquetas tienen prefijos, como f:view y h:inputText; éstas son etiquetas JSF.
- Las etiquetas **h:inputText**, **h:inputSecret** y **h:commandButton** corresponden al *TextField*, *PasswordField* y al botón *Submit* de HTML.
- Los campos de entrada son conectados al atributo del objeto. Por ejemplo, el atributo #"{usuario.nombre}" nos dice según la implementación JSF que se conecta el *TextField* con el atributo nombre del objeto usuario. Discutimos esta vinculación en más detalle más tarde en este capítulo.

Cuando el usuario introduce el nombre y la contraseña, aparece una página de bienvenida. El mensaje de bienvenida contiene el nombre de usuario, mientras que por ahora ignoraremos la contraseña.

Como el lector puede comprobar, el propósito de esta aplicación no es impresionar a nadie, sino simplemente mostrar las piezas que son necesarias para construir una aplicación JSF. Esta sencilla aplicación contiene las siguientes partes:

- Dos <u>páginas</u>, una que contiene casillas de texto para el login de usuario y su contraseña (**index.jsp**) y otra con la bienvenida (**hola.jsp**).
- Un bean que maneja los datos de usuario, (en este caso nombre y contraseña).
- Un *bean* es simplemente una clase Java en la que sus campos son accedidos siguiendo métodos *getter* y *setter* convencionales. El código está en el archivo **UsuarioBean.java**.
- Un <u>archivo de configuración</u> para la aplicación que contiene recursos del *bean* y las reglas de navegación. Por defecto, este archivo se denomina **face-config.xml**.
- Los archivos que son necesarios para el servlet: el archivo web.xml y el archivo index.html que redireccionan el usuario a la dirección correcta para la página de entrada en el sistema.

Fichero: ejemploBasico/WEB-INF/classes /UserBean.java

```
public class UsuarioBean {
    private String nombre;
    private String password;

// ATRIBUTO: nombre
```

9

```
public String getNombre() { return nombre; }
public void setNombre(String nuevoValor) { nombre = nuevoValor; }

// ATRIBUTO: password
public String getPassword() { return password; }
public void setPassword(String nuevoValor) { password = nuevoValor; }
}
```

Estructura de directorios para una aplicación JSF

Una aplicación JSF se corresponde con una estructura de directorios que sigue un modelo estándar, la cual se puede comprimir en un archivo con extensión WAR, para mayor comodidad y portabilidad. Esta estructura de directorios estándar es:

```
Aplicación/
Ficheros HTML y JSP
WEB-INF/
archivos de configuración
classes/
archivos .class
lib/
librerías
```

Con lo que la estructura del archivo WAR, queda tal que así:

```
ejemploBasico.war
     index.html
     index.jsp
     hola.jsp
       META-INF/
             MANIFEST.MF
         WEB-INF/
             faces-config.xml
              web.xml
                classes/
                     UsuarioBean.class
                lib/
                     common-beanutils.jar
                     common-collections.jar
                     common-digester.jar
                     common-logging.jar
                     jsf-api.jar
                     jsf-impl.jar
                     jstl.jar
                     standard.jar
```

donde el directorio META-INF se genera automáticamente al crear el archivo .war.

Capitulo 2 Primeros Pasos

2.1 Instalación

El tutorial está escrito usando la siguiente configuración:

- Sistema Operativo: Windows XP Home
- Maquina Virtual Java: JDK 1.5.0 06 de Sun Microsystems
- Tomcat 5.5.16
- Implementación de referencia de JavaServer Faces v1.1

Pasemos a ver cómo se instala y configura el software necesario:

2.1.1 Instalación del JDK (Java Development Kit):

Primero hay que descargar el kit de desarrollo de Java del enlace: http://java.sun.com/j2se/1.5.0/download.jsp

Una vez descargado el fichero ejecutable, la instalación es sencilla: doble clic y seguir los pasos del asistente, seleccionando las opciones por defecto. Recomendamos la creación de un directorio \ArchivosDePrograma con el mismo nombre que el que utiliza automáticamente Windows, pero sin espacios; aquí deberían ir todas las carpetas creadas por el instalador del JDK. Utilizar esta carpeta en lugar de la estándar de Windows tiene la ventaja de que, bajo ciertas circunstancias, evita problemas en el acceso a los directorios por parte de algunos IDEs.

Una vez instalado, pasamos a su configuración. Habrá que modificar la variable de entorno **PATH**. Para ello hay que acceder a las variables de entorno de Windows: *Panel de control->Sistema->Opciones avanzadas* (Figura 2.1) *->Variables de entorno* (Figura 2.2)

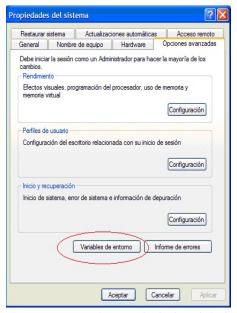


Figura 2.1 Ventana de propiedades del sistema

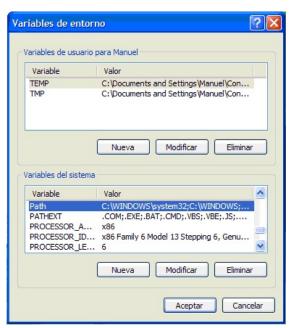


Figura 2.2 Ventana de variables de entorno

Localice la variable **PATH** y la modifiquela para que contenga la ruta del directorio **bin** de la instalación de Java, por ejemplo:

c:\ArchivosDePrograma\Java\jdk1.5.0 06\bin

2.1.2 Instalación de Apache Tomcat:

Primero hay que descargar Tomcat de la página oficial de Apache: http://tomcat.apache.org/download-55.cgi

en la cual deberá buscar la última versión disponible y seleccionarla.

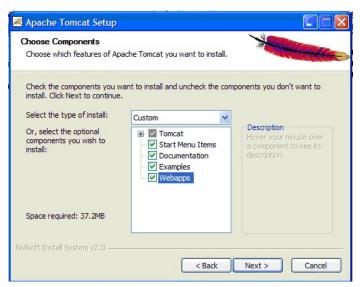


Figura 2.3 Instalación de Apache Tomcat

Descargado el fichero ejecutable, la instalación es sencilla: doble clic y seguir los pasos del asistente, seleccionando las opciones por defecto, salvo el tipo de instalación que en lugar de ser normal, deberá ser completa (Figura 2.3), el motivo de la instalación completa es porque, aunque parezca mentira, al incluir *Examples* en la instalación, dispondremos de las librerias *standard.jar* y *jstl.jar*, ubicadas en c:\ArchivosDePrograma\Tomcat5.5\webapps\jsp-examples\WEB-INF\lib, necesarias para cualquier aplicación JSF. En cuanto a la ubicación del directorio de instalación (por comodidad puede poner c:\ArchivosDePrograma\Tomcat5.5). Durante el proceso se muestra el puerto por el que el servidor se mantiene a la escucha (8080) y se pide una contraseña para actuar como administrador. Para nuestras pruebas se recomienda dejar el puerto por defecto y no poner contraseña. Una vez instalado, se pasa a su configuración.

Es necesario añadir la variable de entorno **JAVA_HOME** al sistema, apuntando al directorio donde se haya instalado el JDK (para que el Tomcat sepa donde buscar el compilador de Java y demás utilidades), por ejemplo:

c:\ArchivosDePrograma\Java\jdk1.5.0_06

Establezca la variable de entorno **CATALINA_HOME** para que apunte al directorio donde haya instalado el servidor Tomcat, por ejemplo:

c:\ArchivosDePrograma\Tomcat5.5

y vuelva a modificar la variable **PATH**, para añadir la ruta del directorio bin de la instalación de Tomcat, por ejemplo:

c:\ArchivosDePrograma\Tomcat5.5

Y ya esta lista toda la configuración; sólo queda ponerlo en marcha para comprobar que todo ha sido correcto. Para ello ejecute el programa *Monitor Tomcat*, desde el cual se puede arrancar Tomcat (*start service*) y detenerlo (*stop service*).

Una vez arrancado, acceda a la dirección web http://localhost:8080/ desde su navegador favorito. Si todo ha ido bien, deberá ver la página principal de Tomcat (Figura 2.4). Si inicialmente no se conecta al puerto 8080, darle a recargar en el navegador y probar de nuevo).

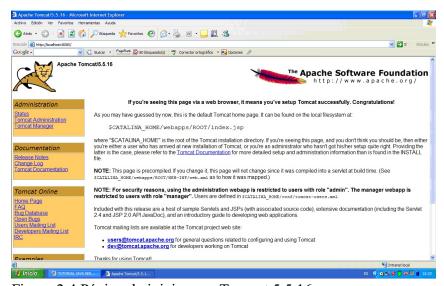


Figura 2.4 Página de inicio para Tomcat 5.5.16

Recuerde que **localhost** equivale a la dirección **127.0.0.1**, por lo que obtendría el mismo efecto, accediendo mediante http://127.0.0.1:8080/.

2.1.3 Instalación de JavaServer Faces

Puede descargarla de la dirección:

http://java.sun.com/javaee/javaserverfaces/download.html

Simplemente descomprima el archivo donde le parezca mejor, por ejemplo: c:\ArchivosDePrograma\Java\jsf-1 1

Dentro de dicha carpeta podrá encontrar el directorio *lib*, con las librerías necesarias para las aplicaciones JSF.

2.2 Entornos de desarrollo

En el ejemplo básico del capítulo anterior las páginas JSF y archivos de configuración se crearon mediante un editor de texto. No obstante, existe la posibilidad de usar entornos de desarrollo gráficos, lo que resulta recomendable dada la complejidad de esta operación y lo repetitiva que es.

Se pueden encontrar diferentes entornos, como por ejemplo:

- Java Studio Creator
- NetBeans
- Eclipse
- Exadel Studio Pro y Exadel Studio
- FacesIDE
- IBM's WebSphere Application Developer (WSAD)
- Oracle's Jdeveloper
- Borland Jbuilder
- IntelliJ IDEA

Veremos someramente algunos de ellos.

2.2.1 Java Studio Creator de Sun Microsystems

Sun Microsystems ofrece la herramienta de desarrollo Sun Java Studio Creator, (Figura 2.5), el cual se puede descargar de aquí:

http://developers.sun.com/prodtech/javatools/jscreator/downloads

Java Studio Creator es una solución completa de desarrollo, depuración y despliegue de aplicaciones, que incorpora la tecnología de producción JavaServer Faces (JSF), capaz de proporcionar un mayor rendimiento y eficiencia en la codificación.

El producto también incorpora:

- El runtime de Java Enterprise System, que permite a los desarrolladores evaluar las aplicaciones en preproducción de forma rápida.
- Incluye soporte para Sun Java System Application Server, Platform Edition.

- PointBase, un servidor de bases de datos SQL (incluido en Java System Application Server).
- El kit de desarrollo (SDK) para Java 2, Standard Edition (J2SE).

Elementos de ayuda para desarrolladores, tales como ejemplos, tutoriales, componentes visuales para la tecnología JavaServer Faces, etc.

Al integrar de forma transparente herramientas, runtimes y componentes, Java Studio Creator permite realizar codificación en Java de forma visual y eficiente. De esta forma, los desarrolladores pueden centrarse en añadir valor a las aplicaciones con elementos como la interacción de usuario y la lógica de negocio, en lugar de tener que dedicar la mayor parte de su tiempo a la "fontanería" de la infraestructura de sistemas y los servicios web.

La herramienta Java Studio Creator está totalmente basada en la plataforma Java y en los estándares desarrollados mediante el programa Java Community

Process (JCP); ello asegura la portabilidad entre los entornos de despliegue y herramientas de desarrollo tales como Sun Java Studio Enterprise.



Figura 2.5 Aspecto del IDE Sun Java Studio Creator 2

2.2.2 NetBeans

El IDE NetBeans (Figura 2.6), lo puede descargar junto con la Máquina virtual de java, desde aquí:

http://java.sun.com/j2se/1.5.0/download.jsp

es un entorno de desarrollo, de código abierto, una herramienta para programadores para escribir, compilar, corregir errores y para ejecutar programas. Está escrito en Java - pero puede servir de soporte a cualquier otro lenguaje de programación. Existe también un número enorme de módulos para extender el NetBeans IDE. El NetBeans IDE es un producto libre y gratuito sin restricciones de utilización.

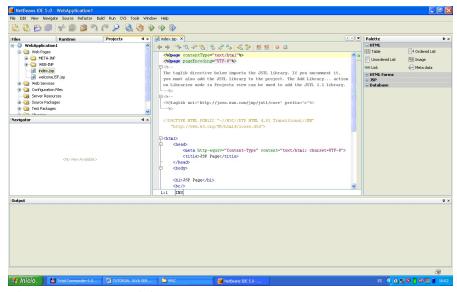


Figura 2.6 Aspecto del IDE NetBeans

2.2.3 Eclipse-SDK

Eclipse, (Figura 2.7), lo puede descargar desde aquí:

http://www.eclipse.org/downloads/

es una IDE multiplataforma libre para crear aplicaciones clientes de cualquier tipo. La primera y más importante aplicación que ha sido realizada con este entorno es la afamado IDE Java llamado Java Development Toolkit (JDT) y el compilador incluido en Eclipse, que se usaron para desarrollar el propio Eclipse.

El proyecto Eclipse se divide en tres subproyectos:

- El Core de la aplicación, que incluye el subsistema de ayuda, la plataforma para trabajo colaborativo, el Workbench (construido sobre SWT y JFace) y el Workspace para gestionar proyectos.
- Java Development Toolkit (JDT), donde la contribución de Erich Gamma ha sido fundamental.
- Plug-in Development Enviroment (PDE), que proporciona las herramientas para el desarrollo de nuevos módulos.

La versión actual de Eclipse dispone de las siguientes características:

- Editor de texto
- Resaltado de sintaxis

- Compilación en tiempo real
- Pruebas unitarias con JUnit
- Control de versiones con CVS
- Integración con Ant
- Asistentes (wizards): para creación de proyectos, clases, tests, etc.
- Refactorización

Asimismo, a través de "plugins" libremente disponibles es posible añadir:

- Control de versiones con Subversion, via Subclipse.
- Integración con Hibernate, via Hibernate Tools.

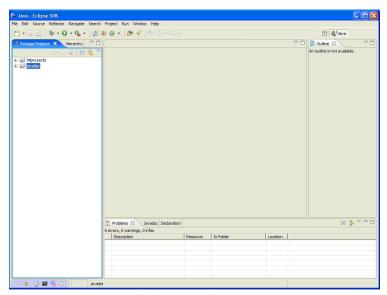


Figura 2.7 Aspecto del IDE Eclipse

Capitulo 3 Modelo Vista Controlador en JSF

3.1 Introducción

El patrón MVC (Modelo Vista Controlador), ver Figura 3.1, nos permite separar la lógica de control (qué cosas hay que hacer pero no cómo), la lógica de negocio (cómo se hacen las cosas) y la lógica de presentación (cómo interaccionar con el usuario).

Utilizando este tipo de patrón es posible conseguir más calidad, un mantenimiento más fácil, perder el miedo al folio en blanco (existe un patrón de partida por el que empezar un proyecto), etc. al margen de todo esto, una de las cosas más importantes que permite el uso de este patrón consiste en normalizar y estandarizar el desarrollo de Software.

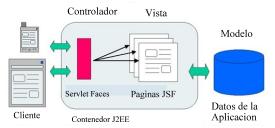


Figura 3.1 Arquitectura MVC

Además, este modelo de arquitectura presenta otras importantes ventajas:

- Hay una clara separación entre los componentes de un programa; lo cual nos permite implementarlos por separado.
- Hay una API muy bien definida; cualquiera que use la API, podrá reemplazar el modelo, la vista o el controlador, sin demasiada dificultad.
- La conexión entre el modelo y sus vistas (ya que puede haber varias) es dinámica: se produce en tiempo de ejecución, no en tiempo de compilación.

3.2 Modelo

Todas las aplicaciones *software* dejan a los usuarios manipular ciertos datos que proceden de una realidad sobre la que se pretende actuar, como supermercados, itinerarios de viaje, o cualquier dato requerido en un dominio problemático particular. A estos datos en estado puro, que representan el estado de la realidad se les llama <u>modelo</u>: modelan la parte de la realidad sobre la que se desea actuar.

El modelo, pues, es el objeto que representa y trabaja directamente con los datos del programa: gestiona los datos y controla todas sus transformaciones. El modelo no tiene conocimiento específico de los diferentes controladores y/o vistas, ni siquiera contiene referencias a ellos. Es el propio sistema el que tiene encomendada la responsabilidad de mantener enlaces entre el modelo y sus vistas, y notificar a las vistas cuándo deben reflejar un cambio en el modelo.

En nuestro ejemplo básico, lo primero que hay que hacer es definir el modelo, que se ve reflejado en el fichero **UsuarioBean.java**, donde se aprecia la clase **UsuarioBean**, que contiene los elementos necesarios para manejar los datos de la aplicación: es necesario un

campo que almacene el nombre del usuario que entró, y otro para su correspondiente contraseña, junto con sus respectivos métodos de asignación y recuperación:

```
public class UsuarioBean {
   private String nombre;
   private String password;

   // ATRIBUTO: nombre
   public String getNombre() { return nombre; }
   public void setNombre(String nuevoValor) { nombre = nuevoValor; }

   // ATRIBUTO: password
   public String getPassword() { return password; }
   public void setPassword(String nuevoValor) { password = nuevoValor; }
```

Este modelo a utilizar en la aplicación se le comunica al sistema JSF mediante el fichero **faces-config.xml**, donde se detalla la parte de **managed-bean**, donde se aprecia un *bean* denominado **usuario**, que está recogido en la clase **UsuarioBean**, y con un ámbito de sesión:

```
<managed-bean>
<managed-bean-name>usuario</managed-bean-name>
<managed-bean-class>UsuarioBean</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

3.3 Vista

La vista es el objeto que maneja la presentación visual de los datos gestionados por el Modelo. Genera una representación visual del modelo y muestra los datos al usuario. Interacciona con el modelo a través de una referencia al propio modelo.

En el ejemplo básico, la vista está manipulada a través de las páginas JSF, es decir, mediante las páginas **index.jsp** y **hola.jsp**. <u>JSF conecta la vista y el modelo</u>. Como ya se ha visto, un componente de la vista puede ligarse a un atributo de un *bean* del modelo, como:

<h:inputText value="#{usuario.nombre}"/>

donde se ve como se declara un campo de texto de entrada (*inputText*) en la vista, ese campo de texto recoge su valor de entrada en el atributo **nombre** de un *bean* denominado **usuario**.

De esta manera se establece el vínculo de enlace en vista y modelo.

3.4 Controlador

El controlador es el objeto que proporciona significado a las órdenes del usuario, actuando sobre los datos representados por el modelo. Entra en acción cuando se realiza alguna operación, ya sea un cambio en la información del modelo o una interacción sobre la Vista. Se comunica con el modelo y la vista a través de una referencia al propio modelo.

Además, JSF opera como un gestor que reacciona ante los eventos provocados por el usuario, procesa sus acciones y los valores de estos eventos, y ejecuta código para actualizar el modelo o la vista.

Retomando el ejemplo básico, una parte del controlador la recogen las lineas de código del fichero **index.jsp** que capturan los datos del nombre de usuario, su contraseña, y el botón de **Aceptar**:

```
<h:inputText value="#{usuario.nombre}"/>
<h:inputSecret value="#{usuario.password}"/>
<h:commandButton value="Aceptar" action="login"/>
y la parte del dichero hola.jsp que muestra el nombre por pantalla:
<h:outputText value="#{usuario.nombre}"/>
```

Por otro lado, está el control para las reglas de navegación, contenido en el fichero **faces-config.xml**, donde por ejemplo, se indica que estando **index.jsp**, si ocurre una acción denominada *login*, navegaremos a la página **hola.jsp**, esta acción comentada, es un string que se declara en la vista como un atributo del botón de aceptar que aparece en el formulario del ejemplo básico. El fichero **faces-config** sería el siguiente:

Por último, termina de definirse el controlador de la aplicación con el *servlet faces*, definido en el fichero **web.xml**:

Con la directiva *<servlet>* se establece el único servlet de nuestra aplicación es el propio del framework JSF.

La directiva *<servlet-mapping>* indica la ruta (url) para acceder a servlet definido anteriormente. Cualquier página JSP que pretendamos visualizar, si contiene la extensión .faces, se visualizará a traves de JSF.

Por último con <*welcome-file-list*> se establece como página de arranque de la aplicación **index.html**. Posteriormente, esta página se redirecciona a **index.faces**, esto se hace así para que el framework JSF, a través de su servlet principal tome el control. Si no se hiciera de esta manera, el servidor de aplicaciones mostraría una simple página JSP como tal, y la compilación de dicha página, fuera del marco JSF, provocaría errores.

3.5 Ciclo de vida de una página JavaServer Faces

El ciclo de vida de una página JavaServer Faces page es similar al de una página JSP: El cliente hace una petición HTTP (*Hiper Text Transfer Protocol*) 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 dependiendo 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.

3.5.1 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; la idea es que en una aplicación JSF se pueden mezclar páginas JSF y JSP (no-JSF) y, según se pidan y/o se devuelvan una u otras, tendremos diferentes respuestas y/o 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 generada por el *servlet* en la que no se ha ejecutado la fase Renderizar la Respuesta. Un ejemplo es una página JSP que no incorpora componentes JavaServer Faces.
- Petición Faces:
 - Una petición al *servlet* que fue enviada desde una Respuesta *Faces* previamente generada. Un ejemplo es un formulario enviado desde un componente de interfaz 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 al *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 construir 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 (p.ej. una imagen sencilla) 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 <u>aplicar valores</u> 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.

3.5.2 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 por ejemplo una casilla de verificación (*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.

La **figura 3.2** ilustra los pasos del ciclo de vida petición-respuesta JavaServer Faces.

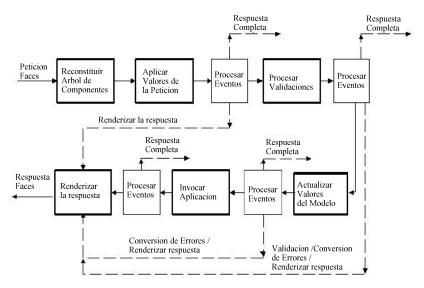


Figura 3.2 Ciclo de vida petición-respuesta de una página JSF

3.5.2.1 Reconstituir el árbol de componentes

Cuando se hace una petición de una página JavaServer Faces, o cuando se produce un evento como pulsar sobre un enlace o un botón, el sistema JavaServer Faces entra en 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 y los validadores y graba el estado en el **FacesContext**.

3.5.2.2 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**. A continuación, el valor se almacena localmente en el componente. Si la conversión del valor falla, 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 <u>renderizar la respuesta</u>, junto con cualquier error de validación resultante de la fase procesar validaciones.

Si durante esta fase se produce algún evento, la implementación JavaServer Faces comunica estos 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 este momento, se han puesto los nuevos valores en los componentes y los mensajes y eventos se han puesto en sus colas.

3.5.2.3 Procesar validaciones

Durante esta fase, el sistema JavaServer Faces procesa todas las validaciones registradas con los componentes del árbol. Examina los atributos del componente especificados por las reglas de validación y evalúa las reglas con los valores de dichos

atributos. Si un valor incumple una regla, 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.

3.5.2.4 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 actualizar los valores del modelo con los nuevos valores pasados en la petición. Sólo se actualizarán los componentes que tengan expresiones **valueRef**. Si el dato local no se puede convertir a los tipos especificados por los atributos del objeto del modelo, el ciclo de vida avanza directamente a la fase <u>renderizar las respuesta</u>, durante la que se dibujará de nuevo la página mostrando los errores, similar a lo que sucede con los errores de validación.

3.5.2.5 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.

Posteriormente, la implementación JavaServer Faces configura el árbol de componentes de la respuesta a esa nueva página y, por último, transfiere el control a la fase Renderizar la Respuesta.

3.5.2.6 Renderizar la Respuesta

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

Si se encontraron errores durante las fases <u>aplicar los valores a la petición</u>, <u>procesar validaciones</u> o <u>actualizar los valores del modelo</u>, 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.

3.6 Una sencilla aplicación JavaServer Faces en detalle

Vamos a comentar los pasos para construir la aplicación vista en el primer capítulo y luego se comentará en detalle.

1. La estructura inicial de directorios debe ser la siguiente (en cualquier carpeta de trabajo):



2. Debe compilarse el archivo **UsuarioBean.java** para generar **UsuarioBean.class**. una vez generado lo coloca en la carpeta /**classes**, en cuanto al fichero .java lo puede dejar en la misma carpeta o crear una propia, por ejemplo /**src** que recoja sus ficheros fuentes, es decir:

```
WEB-INF/
faces-config.xml
web.xml
src/
UsuarioBean.java
classes/
UsuarioBean.class
```

3. Debe crearse la carpeta **lib** dentro de **WEB-INF** y dentro de ésta, deben copiarse las bibliotecas necesarias, representadas por los siguientes ficheros (la ubicación original de éstos dependerá de donde haya instalado la implementación JSF y Tomcat):

```
jsf-1_1/lib/jsf-api.jar
jsf-1_1/lib/jsf-impl.jar
jsf-1_1/lib/commons-digester.jar
jsf-1_1/lib/commons-beanutils.jar
jsf-1_1/lib/commons-collections.jar
jsf-1_1/lib/commons-logging.jar
tomcat 5.5/webapps/jsp-examples/WEB-INF/lib/jstl.jar
tomcat 5.5/webapps/jsp-examples/WEB-INF/lib/standard.jar
```

4. Por último debe construirse el fichero .war (Web Archive). Para ello, dentro del directorio /ejempoBasico, debe ejecutar:

jar cvf ejemploBasico.war.

y el fichero **ejemploBasico.war** generado hay que copiarlo al directorio:

c:\ArchivosDePrograma\Tomcat5.5

lo que producirá el despliegue inmediato de la aplicación en el servidor Tomcat.

5. Por último, sólo queda probar que todo funciona bien. Desde su navegador, acceda a la dirección:

http://localhost:8080/ejemploBasico

y si todo es correcto, la aplicación se iniciará correctamente.

Por otro lado, para desplegar la aplicación existe una alternativa en caso de no querer usar archivos .war: se puede copiar directamente la carpeta ejemploBasico, dentro de c:\ArchivosDePrograma\Tomcat5.5\webapps

y acceder a la aplicación mediante la dirección

http://localhost:8080/ejemploBasico/index.faces

3.7 Beans y páginas JSF

Las aplicaciones web correctamente planificadas tienen dos partes: la parte de presentación y la lógica de negocio.

La parte de presentación afecta a la apariencia de la aplicación, y en el contexto de una aplicación basada en navegadores, la apariencia esta determinada por las etiquetas HTML, esto comprende marcos, tipos de caracteres, imágenes, etc. La lógica de negocio se implementa en Java y determina el comportamiento de la aplicación.

En el contexto de JSF, la lógica de negocio está contenida en los *beans*, y el diseño esta contenido en las paginas web Empezaremos nuestro estudio por los *beans*.

3.7.1 **Beans**

Un *bean* es una clase Java que contiene atributos. Un atributo es un valor identificado por un nombre, pertenece a un tipo determinado y puede ser leído y/o escrito sólo a través de métodos a tal efecto llamados métodos *getter* y *setter*:

- Para conocer el valor de un campo llamado xxx se utiliza la función getXxx (o isXxx si es de tipo boolean)
- Para asignarle valor a un campo llamado xxx se usa la función setXxx

Por ejemplo, **UsuarioBean.java** (Figura 3.3) tiene dos atributos, **nombre** y **password**, ambos de la clase **String**:

```
public class UsuarioBean {
    private String nombre;
    private String password;

    // ATRIBUTO: nombre
    public String getNombre() { return nombre; }
    public void setNombre(String nuevoValor) { nombre = nuevoValor; }

    // ATRIBUTO: password
    public String getPassword() { return password; }
    public void setPassword(String nuevoValor) { password = nuevoValor; }
}
```

Figura 3.3 Implementación de UsuarioBean.java

En una aplicación JSF, se deben usar *beans* para todos los datos accedidos por una página. Los *beans* son los conductos entre la interfaz de usuario y la trastienda de la aplicación.

Junto a las reglas de navegación del fichero faces-cofig.xml:

```
<faces-config> <navigation-rule>
```

nos encontramos con las definiciones de los *beans*. Así, puede usar el nombre del *bean* **usuario**, en los atributos de los componentes de la página. Por ejemplo, **index.jsp** contiene la etiqueta:

<h:inputText value="#{usuario.nombre}"/>

La etiqueta **managed-bean-class**, simplemente especifica la clase a que pertenece el *bean*, en nuestro caso **UsuarioBean.class**.

Y finalmente tenemos la etiqueta **managed-bean-scope**, que en nuestro caso tiene el valor *session*, con lo que estamos indicando que el *bean* estará disponible para un cliente durante todo el tiempo que esté conectado al sitio web y, por tanto, mantendrá su valor a través de múltiples páginas, es decir, se esta indicando en ámbito del *bean*.

3.7.2 Paginas JSF

Se necesita una pagina JSF por cada pantalla de presentación. Dependiendo de lo que se quiera hacer, las páginas típicas son .jsp o .jsf.

Siguiendo con nuestro ejemplo, la página **index.jsp** comienza con las declaraciones de las etiquetas, en la que se aprecia la importación de la librería de etiquetas *core*, se usan, entre otras aplicaciones, para manejo de eventos, atributos, conversión de datos, validadores, recursos y definición de la página, las cuales usan el prefijo **f**, por ejemplo **f:view**, y la librería de etiquetas *html_basic*, su utilización es básicamente para la construcción de formularios y demás elementos de interfaz de usuario, que usa el prefijo **h**, por ejemplo **h:form**:

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

A continuación, los valores de los campos de entrada son volcados del atributo del *bean* con el nombre usuario:

```
<h:inputText value="#{usuario.nombre}"/>
```

en el momento de presentar la página, el método **getNombre** es invocado para obtener el valor del atributo **nombre**. Y cuando la página se acepta pulsando el botón **Aceptar**, el método **setNombre** es invocado estableciendo el valor de usuario que ha entrado.

La etiqueta h:commandButton, ubicada en el fichero index.jsp:

Fichero: ejemploBasico/index.jsp

```
< ht.ml>
  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
      <head>
         <title>Una simple aplicacion JavaServer Faces</title>
      </head>
      <body>
         <h:form>
     <h3>Por favor, introduzca su nombre y password.</h3>
            Nombre:
                     <h:inputText value="#{usuario.nombre}"/>
                  Password:
                     <h:inputSecret value="#{usuario.password}"/>
                  >
               <h:commandButton value="Aceptar" action="login"/>
         </body>
   </f:view>
</html>
```

tiene un atributo denominado *action*, cuyo valor (un *String*) es usado para activar una regla de navegación, la cual está recogida en el fichero **faces-config.xml**, de manera que al producirse esta acción al pulsar el botón, navegaremos a una nueva página

En cuanto a la páginas hola.jsp:

```
<h+m1>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <f:view>
       <head>
           <title>Una simple aplicacion JavaServer Faces</title>
       </head>
       <body>
           <h:form>
              <h3>
                  Bienvenido a JavaServer Faces,
                  <h:outputText value="#{usuario.nombre}"/>!
              </h3>
           </h:form>
       </body>
   </f:view>
</html>
```

En este caso, sólo cabe mencionar la etiqueta **h:outputText**, la cual muestra el nombre del usuario que entró, almacenada previamente en el *bean*.

3.8 Navegación

Otra posibilidad que tiene el desarrollador de la aplicación es definir la navegación de páginas por la aplicación, como qué página va después de que el usuario pulse un botón para enviar un formulario.

El desarrollador de la aplicación define la navegación por la aplicación mediante el fichero de configuración, **faces-config.xml**, el mismo fichero en el que se declararon los *managed beans*.

Cada regla de navegación define cómo ir de una página (especificada en el elemento from-view-id) a otras páginas de la aplicación. El elemento navigation-rule puede contener cualquier número de elemento navigation-case, cada uno de los cuales define la página que se abrirá luego (definida por to-view-id) basándose en una salida lógica (definida mediante from-outcome).

La salida se puede definir mediante el atributo action del componente UICommand que envía el formulario.

Ahora vamos con las reglas de navegación, de esta manera, indicamos a que página ir tras otra página. En este caso la navegación es simple, tras pulsar el botón **Aceptar**, navegamos desde **index.jsp** hasta **hola.jsp**. Estas reglas de navegación se especifican en el fichero **faces-config.xml**, como se muestra a continuación:

El valor de **from-outcome** (**login**), indica la acción que se debe producir en la página **from-view-id** (**index.jsp**) para navegar al destino representado por **to-view-id** (**hola.jsp**). La acción es un mero nombrecito que se da a los botones de un formulario; en nuestro caso, la página **index.jsp** incluía la etiqueta:

<h:commandButton value="Aceptar" action="login"/>

3.9 Configuración Servlet

Una aplicación JSF requiere un servlet, llamado FacesServlet, el cual actual como controlador.

Esta configuración esta recogida en el fichero **web.xml**, el cual se muestra a continuación, y afortunadamente, se puede usar el mismo fichero para múltiples aplicaciones JSF. De esta manera se establece el único servlet de nuestra aplicación es el propio del framework JSF.

Lo importante aquí es el mapeo del *servlet*. Todas las páginas JSF son procesadas por un *servlet* especial que forma parte del código de implementación de JSF.

Para asegurarnos de que el *servlet* está correctamente activado cuando una pagina JSF es demandada, la URL posee un formato especial. En nuestra configuración, es el formato .faces. Por ejemplo, en su navegador no se puede poner simplemente :

http://localhost:8080/login/index.jsp sino que la dirección ha de ser: http://localhost:8080/login/index.faces

El contenedor *servlet* usa el la regla del mapeado *servlet* para activar el *servlet* JSF, quien elimina el sufijo **faces** y carga la pagina **index.jsp**

Esto se hace así para que el framework JSF, a través de su servlet principal tome el control. Si no se hiciera de esta manera, el servidor de aplicaciones mostraría una simple página JSP como tal, y la compilación de dicha página, fuera del marco JSF, provocaría errores.

Capitulo 4

Managed Beans, navegación y etiquetas básicas JSF

4.1 Managed Beans

Un apartado importante en el diseño de aplicaciones web es la separación de la presentación y la lógica de negocio. JSF usa *beans* para lograr esta separación. Las páginas JSF se refieren a las propiedades del *bean*, y la lógica de programa esta contenida en el código de implementación del *bean*. Los beans son fundamentales para programar JSF.

4.1.1 Concepto

Según la especificación de los JavaBeans, un Java *bean* es "un componente reutilizable del *software*, que puede ser manipulado". Esta es una definición bastante imprecisa y, ciertamente, como se verá en este capítulo, los *beans* sirven para una gran variedad de propósitos.

A primera vista, un *bean* parece ser similar a cualquier otro objeto. Sin embargo, los *beans* se manejan de una forma más concreta. Cualquier objeto se crea y se manipula dentro de un programa Java llamando a los constructores e invocando a los métodos. Sin embargo, los *beans* pueden ser configurados y manipulados sin programar, a través de entornos de trabajo (*frameworks*) o entornos de desarrollo integrados (IDE-*Integrated Development Environment*), que los utilizan mediante técnicas de introspección.

En el contexto de JavaServer Faces, los *beans* no se utilizan para nada relacionado con la interfaz de usuario: los *beans* se utilizan cuando se necesita conectar las clases Java con páginas web o archivos de configuración.

Una vez que un *bean* ha sido definido, puede ser accedido a través de etiquetas. JSF. Por ejemplo, la siguiente etiqueta lee y actualiza el atributo **password** del *bean* **usuario**:

<h:inputSecret value="#{usuario.password}"/>

4.1.1.1 Atributos

Las características más importantes de un *bean* son los atributos que posee, también llamados propiedades. Cada uno de éstos tiene:

- Un nombre.
- Un tipo.
- Métodos para obtener y establecer los valores de atributo.

La especificación de los JavaBeans impone una sola exigencia en una clase *bean*: debe tener un constructor predeterminado, sin parámetros. Además, para que los entornos de trabajo o de desarrollo puedan acceder a sus atributos mediante introspección, una clase *bean* debe declarar métodos *get* y/o *set* para cada uno de ellos, o debe definir descriptores utilizando la clave **java.beans.BeanDescriptor**.

Consideremos los dos siguientes métodos:

T getCampo1()

void setCampo1(T nuevoValor)

Éstos se corresponden con funciones que permiten leer y escribir en un campo de nombre **campo1** y tipo **T**. Si sólo tuviésemos el primer método, estaríamos tratando con un

atributo de solo lectura, y si tuviésemos sólo el segundo método, estaríamos ante uno de solo escritura. Nótese que el nombre del atributo forma parte del nombre del método que sigue al prefijo **get** o **set**.

Un método **get...** no posee parámetros mientras que un método **set...** posee un parámetro y no devuelve ningún valor. Por supuesto, una clase *bean* puede tener otros métodos además de los *getter* y *setter*.

En el caso de que un atributo sea de tipo **boolean**, podemos elegir entre dos prefijos en el método de lectura: **get** o **is**, es decir:

boolean isCampo1()
boolean getCampo1()

4.1.1.2 Expresiones para valores inmediatos y directos

Muchos componentes de interfaz de usuario JSF tienen un valor de atributo que permite especificar ya sea un valor inmediato o un valor directo obtenido del atributo de un *bean*. Por ejemplo, se puede especificar un valor inmediato de la forma:

<h:outputText value="Hola mundo!"/>
o se puede especificar un valor directo:
<h:inputText value="#{usuario.nombre}"/>

4.1.1.3 Agrupación de mensajes:

Cuando se implementa una aplicación web, es una buena idea coleccionar todas las cadenas de mensajes en un mismo sitio. Este proceso da facilidades para conservar mensajes consistentes y cruciales, y facilita las traducciones a otros idiomas.

JSF simplifica este proceso mediante los ficheros de propiedades. Primero, hay que aglutinar todos los literales de tip **String** en un archivo de extensión **.properties**:

saludo=¡Bienvenido a JSF!

pregunta=Escoja un número, por favor.

despedida=Que tenga un buen día.

Luego se guarda este fichero junto a los ficheros de clases .java, por ejemplo en:

WEB-INF/classes/mensajes.properties

aunque puede elegirse cualquier ruta y nombre de archivo, siempre que sea de extensión .**properties**.

A continuación se añade la etiqueta **f:loadBundle** a tu pagina JSF, tal que así:

<f:loadBundle basename="mensajes" var="msjs"/>

Y por último se puede usar una expresión de valor directo para acceder a la cadena del mensaje:

<h:outputText value="#{msjs.saludo}"/>

4.1.2 Ámbitos de los beans

Para comodidad del programador aplicaciones web, un contenedor de *servlets* suministra diferentes ámbitos, de petición, de sesión y de aplicación.

Estos ámbitos normalmente mantienen *beans* y otros objetos que necesitan estar disponibles en diferentes componentes de una aplicación web.

🖎 Ámbito de tipo petición:

Es el de vida más corta. Empieza cuando una petición HTTP comienza a tramitarse y acaba cuando la respuesta se envía al cliente.

Por ejemplo, el la siguiente línea de código:

<f:loadBundle basename="mensajes" var="msjs"/>

la etiqueta **f:loadBundle** hace que la variable **bundle** solo exista mientras dura la petición. Un objeto debe tener un ámbito de este tipo sólo si lo que se quiere es reenviarlo a otra fase de procesado.

Ámbito de tipo sesión:

El navegador envía una petición al servidor, el servidor devuelve una respuesta, y entonces ni el navegador ni el servidor tiene cualquier obligación para conservar cualquier memoria de la transacción. Este acomodamiento simple marcha bien para recuperar información básica, pero es poco satisfactorio para aplicaciones del lado del servidor. Por ejemplo, en una aplicación de un carrito compras, necesita al servidor para recordar los contenidos del carrito de compras.

Por esa razón, los contenedores *servlet* amplían el protocolo de HTTP para seguir la pista a una sesión, esto se consigue repitiendo conexiones para el mismo cliente. Hay diversos métodos para el rastreo de sesión . El método más simple es usar *cookies*: La pareja nombre /valor la envía el servidor a un cliente, esperando que regresen en subsiguientes peticiones.

Mientras el cliente no desactive las *cookies*, el servidor recibe un identificador de sesión por cada petición siguiente. Los servidores de aplicación usan estrategias de retirada, algo semejante como al URL *rewriting*, para tratar con esos clientes que no devuelven *cookies*. El URL *rewriting* añade un identificador de sesión a la URL, con lo cual se parece algo a esto:

http://ejemploBasico/index.jsp;jsessionid=64C28D1FC...D28

La sesión el rastreo con *cookies* es completamente transparente al desarrollador web, y las etiquetas estándar JSF automáticamente reescriben URL si un cliente no usa *cookies*.

El ámbito de sesión permanece desde que la sesión es establecida hasta que esta termina. Una sesión termina si la aplicación web invoca el método invalidate en el objeto *HttpSession* o si su tiempo expira.

Las aplicaciones Web típicamente colocan la mayor parte de sus *bean* dentro de un ámbito de sesión.

Por ejemplo, un bean UsuarioBean puede contener información acerca de usuarios que son accesibles a lo largo de la sesión entera. Un bean CarritoCompraBean puede irse llenando gradualmente durante las demandas que levantan una sesión.

Ámbito de tipo aplicación:

Persiste durante toda la aplicación web. Este ámbito compartido entre todas las peticiones y sesiones.

Existe la posibilidad de anidar *beans*, para conseguir objetivos mas complicados. Considere el siguiente *bean* como ejemplo:

```
<managed-bean>
  <managed-bean-name>examen</managed-bean-name>
  <managed-bean-class>ExamenBackingBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
     property-name>problemas
     list-entries>
        <value-class>ProblemasBean</value-class>
        <value>#{problema1}</value>
        <value>#{problema2}</value>
        <value>#{problema3}</value>
        <value>#{problema4}</value>
        <value>#{problema5}</value>
     </list-entries>
  </managed-property>
</managed-bean>
```

Ahora definimos objetos *beans* con nombres desde problema1 hasta problema5, tal que así:

Cuando el *bean* **examen** se referencia por vez primera, se dispara automáticamente la creación de los *beans* **problema1** .. **problema5**, sin que haya que preocuparse por el orden en el cual se han especificado. Note que estos beans (**problema1** .. **problema5**) no tienen ningún ámbito (none), puesto que nunca son demandados desde una página JSP.

Cuando junte ámbitos de *beans*, hay que asegurarse de que son compatibles, tal y como se muestra en el siguiente cuadro:

Cuando defina un bean de ámbito	Puede usar otro ámbito de tipo
none	none
application	none, application
session	none, application, session
request	none, application, session, request

Cuadro 4.1 Compatibilidades de ámbitos de beans

4.1.3 Configuración de un bean a través de XML

Vamos a describir como configurar un *bean* mediante un archivo de configuración. Los detalles son mas bien técnicos. El archivo de configuración más comúnmente usado es **WEB-INF/faces-config.xml**.

Esta configuración comprende las características básicas del propio *bean*, y los valores que se pueden establecer para los atributos del *bean*, ya sean valores simples, o diversos valores para un mismo atributo del *bean*.

Un *bean* se define con una etiqueta **managed-bean** al comienzo del fichero **faces-config.xml**. Básicamente se debe especificar nombre del bean con la etiqueta <**managed-bean-name**>, clase donde está recogida dicho *bean*, con la etiqueta <**managed-bean-scope**>, por ejemplo, para definir un *bean* llamado **usuario**, que está en la clase **UsuarioBean** y con un ámbito de sesión, sería:

El ámbito puede ser de sesión, de aplicación, de petición o ninguno (**session**, **application**, **request**, o **none**).

Establecer valores de atributos:

Puede inicializar cualquier variable de un managed bean. Como mínimo, deberá indicar el nombre del atributo, y un valor. Dicho valor puede ser nulo, un *String* o una expresion directa. Cada atributo es inicializado con el elemento <managed-property> (anidado dentro de <managed-bean>), el nombre del atributo se inicializa con la directiva

roperty-name>, y para los valores está el elemento <value> o <null-value> para valoers
nulos.

Echemos un vistazo al siguiente ejemplo:

```
<managed-bean>
  <managed-bean-name>usuario</managed-bean-name>
  <managed-bean-class>UsuarioBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>nombre</property-name>
    <value>yo</value>
  </managed-property>
    <managed-property>
    <property-name>password</property-name>
    <value>rojo70
</managed-property>
</managed-property>
</managed-property>
</managed-bean>
```

Observe el en el ejemplo que está declarado un *bean* llamado **usuario**, contenido en una clase **UsuarioBean**, Este *bean*, posee un atributo **nombre** y que se ha inicalizado con el valor **yo**, así el atributo nombrado, **password**, se ha inicializado al valor **rojo70.** Para inicializar un atributo a un valor nulo, debe usarse la etiqueta **<null-value>**, por ejemplo:

```
<managed-property>
  property-name>password<null-value />
</managed-property>
```

Inicializar listas y mapas:

En el caso de tenga un atributo de un bean, con un array o una lista de valores, puede usar el elemento **list-entries**>, el cual se anida dentro del elemento **<managed-property**>

Veamos un ejemplo para un atributo llamado **nombres** con una lista de valores, para apreciar la sintaxis:

Una lista puede contener la etiqueta **<value>** para un valor individual y etiquetas **null-value**, para valores nulos. Por defecto, los valores de una lista deben ser cadenas de caracteres (**String**), pero si quiere utilizar otro tipo, tiene la etiqueta **value-class**, que es opcional, por ejemplo:

Un mapa es más complejo. Al igual que las listas, se usa para atributos de beans que posees varios valores, pero en este caso, cada valor se identifica con una clave y con el propio valor. El identificativo para mapas, es <map-entries>, dentro se anidan, una clave <key> y un valor <value> (o <null-value> para valores nulos). Por ejemplo:

```
<managed-property>
  property-name> nombres
     <map-entries>
           <key-class>java.lang.Integer</key-class>
           <map-entry>
                 <key>1</key>
                 <value>Manuel Ruiz</value>
           </map-entry>
           <map-entry>
                 <key>3</key>
                 <value>Cristina Lopez</value>
           </map-entry>
           <map-entry>
                 <key>16</key>
                 <value>Eduardo Robles
           </map-entry>
           <map-entry>
                 <key>26</key>
                 <value>Ana Muñoz</value>
           </map-entry>
     </map-entries>
</managed-property>
```

Al igual que en las listas, podemos establecer el tipo de los valores de las claves y los valores con las etiquetas **key-class** y **value-class** (de nuevo, por defecto es **java.lang.String**). Veámoslo con un ejemplo:

```
<managed-property>
  property-name> numeros/property-name>
      <map-entries>
      <key-class>java.lang.Integer</key-class>
            <map-entry>
                  <key>1</key>
                  <value>Manuel Ruiz</value>
            </map-entry>
            <map-entry>
                  <key>3</key>
                  <value>Cristina Lopez</value>
            </map-entry>
            <map-entry>
                  <key>16</key>
                  <value>Eduardo Robles
            </map-entry>
            <map-entry>
                  <key>26</key>
                  <value>Ana Muñoz</value>
            </map-entry>
      </map-entries>
</managed-property>
```

4.2 Navegación

Las aplicaciones JavaServer Faces usan las reglas de navegación para controlar las navegación entre páginas. Cada regla de navegación especifica cómo ir de una página a las demás dentro de la aplicación. En la arquitectura MVC, la navegación de la página es una de las responsabilidades del controlador. Las reglas de navegación de las aplicaciones JSF están contenidas en el archivo faces-config.xml bajo el directorio WEB-INF.

4.2.1 Concepto

En este apartado veremos cómo configurar la navegación de su aplicación web, de manera que se pase de una página a otra cuando se pulsa un botón o se realiza cualquier otra acción por parte del usuario.

Para empezar, existen dos tipos diferenciados de navegación: navegación estática y dinámica.

4.2.2 Navegación estática

Considere el caso en el que un usuario rellena un formulario de una página web. El usuario puede escribir en campos del texto, puede hacer clic sobre enlaces, pulsar botones o seleccionar elementos de una lista, entre otras muchas cosas.

Todas estas acciones ocurren dentro del navegador del cliente. Cuando, por ejemplo, el usuario pulsa un botón, envía los datos del formulario y éstos son gestionados por el servidor.

Al mismo tiempo, el servidor JSF analiza la entrada del usuario y debe decidir a que página ir para dar la respuesta.

En una aplicación web simple, la navegación es estática. Es decir, pulsar sobre un botón suele redirigir al navegador a una misma página para dar la respuesta. En este caso, simplemente, a cada botón se le da un valor para su atributo de acción (**action**), por ejemplo:

<h:commandButton label="Aceptar" action="login"/>

Esta acción desencadenante, debe concordar con la etiqueta **outcome** del fichero **faces-config.xml**, dentro de sus reglas de navegación.

En esta simple regla de navegación, se indica que tras la acción **login**, se navegará a la página **hola.jsp**, si esta acción ocurre dentro de la página **index.jsp**.

Tenga cuidado con no olvidar la /, en la lineas **from-view-id** y **to-view-id** Puede mezclar reglas de navegación con el mismo **from-view-id**, por ejemplo:

4.2.3 Navegación dinámica

En la mayoría de aplicaciones web, la navegación no es estática. El flujo de la página no depende de qué botón se pulsa, sino que también depende de los datos que el cliente introduce en un formulario. Por ejemplo, una página de entrada al sistema puede tener dos resultados: El éxito o el fracaso.

El resultado depende de una computación, sea cual sea el nombre y la contraseña es legítima. Para implementar una navegación dinámica, el botón de aceptar debe tener un método referencia, por ejemplo:

En este caso, **loginControlador**, referencia un *bean*, y éste debe tener un método denominado **verificarUsuario**.

Un método de referencia, en un atributo de acción, no tiene parámetros de entrada y devuelve una cadena de caracteres, que sera una usada para activar una regla de navegación, por ejemplo, el método **verificarUsuario** debería parecerse a algo así:

```
String verificarUsuario() {
    if (...)
        return "exito";
    else
        return "fracaso";
}
```

El método devuelve un **String**, "éxito" o "fracaso". El manejador de navegación usa el *string* devuelto para buscar una regla de navegación que haga juego. De manera que en las reglas de navegación, podría encontrarse algo así:

4.2.4 Navegación avanzada

En este apartado, se describen las reglas restantes para los elementos de navegación que pueden aparecer en el archivo de **faces-config.xml**. En la Figura 4.1 se puede ver el diagrama de sintaxis de etiquetas válidas.

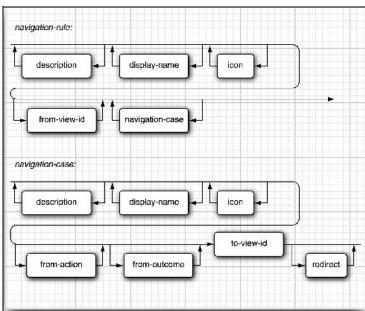


Figura 4.1 Diagrama de sintaxis de los elementos de navegación

Redirección

Si se añade una etiqueta **redirect**, después de **to-view-id**, el contenedor de JSP termina la petición actual y envía una redirección HTTP al cliente.

Con <**redirect**> se le dice a JSF que envíe una redireccion HTTP al usuario de una vista nueva. En el mundo de JSP, esto quiere decir que el usuario ve el URL de la página que él actualmente mira, en contra de la dirección de la página previa.

Por ejemplo:

La respuesta redireccionada dice al cliente que URL usar para la siguiente página. Sin redirección, la dirección original (<u>localhost:8080/ejemplo/index.faces</u>) es la misma cuando el usuario se muda de la página **index.jsp** a **exito.jsp**. Con redirección, el navegador establece la dirección nueva (<u>localhost:8080/ejemplo/exito.faces</u>).

© Comodines

Se pueden usar los comodines, en las etiquetas **from-view-id** de las reglas de navegación; por ejemplo:

Esta regla se aplicará a todas aquellas páginas que empiecen con el prefijo **aplicacion**. Solo se permite un *, y debe estar al final de la cadena del **from-view-id**.

Etiqueta from-action

Junto con la etiqueta **from-outcome**, está también la etiqueta **from-acción**. Esto puede ser útil si se tienen dos acciones separadas con la misma cadena de acción, o dos métodos de referencia de acción que devuelven la misma cadena. Por ejemplo, suponga que tiene dos métodos, **accionRespuesta** y **nuevoExamen**, y ambos devuelven la misma cadena, "repetir", pues, para diferenciar ambos casos de navegación, se usa el elemento *from-action*:

4.3 Etiquetas básicas

Componentes de interfaz 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

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 *actions* 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.
- 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...
- 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, selececcionándolo 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.

La implementación de referencia de JavaServer Faces proporciona una librería de etiquetas personalizadas para renderizar componentes en HTML. A continuación se muestra un lista de etiquetas básicas, cuyo contenido y atributos se explicará en detalle en el siguiente capítulo:

UIForm

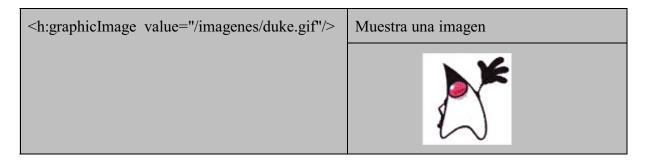
form	
<h:form </h:form 	Representa un formulario

UICommand

commandButton	
<h:commandbutton value="Aceptar" action="siguiente"/></h:commandbutton 	Un botón con una acción asociada
	Aceptar

UIGraphic

Tutorial de JavaServer Faces



<u>UIInput</u>

inputText	
<h:inputtext value="#{formulario.nombre}" /></h:inputtext 	Permite al usuario introducir un string
	Nombre:
inputSecret	
<h:inputsecret value="#{formulario.password}" /></h:inputsecret 	Permite al usuario introducir un string sin que aparezca el string real en el campo
	Password:
inputTextArea	
<h:inputtextarea cols="15" rows="3" value="Escribir aqui"></h:inputtextarea>	Permite al usuario introducir un texto multi-líneas
	Escribir aqui

Tutorial de JavaServer Faces

UIOutput

outputText	
<h:outputtext value="Hola Mundo!"/></h:outputtext 	Muestra una línea de texto
outputLink	
<h:outputlink value="http://www.hoylodejo.com/"></h:outputlink>	Muestra un enlace Pinche aqui

UISelectBoolean

selectBooleanCheckbox	
<pre><h:selectbooleancheckbox value="#{cuestionario.recibirInformacion}"></h:selectbooleancheckbox></pre>	Permite al usuario cambiar el valor de una elección booleana
	✓ Deseo recibir informacion

UISelectMany

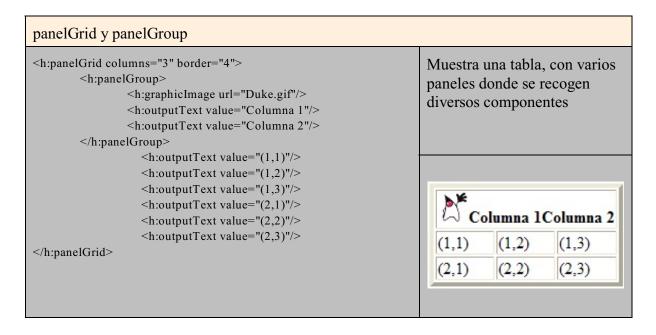
selectManyCheckbox

<h:selectmanycheckbox value="#{cuestionario.aficiones}"> <f:selectitem itemlabel="Musica" itemvalue="musica"></f:selectitem> <f:selectitem itemlabel="Cine" itemvalue="cine"></f:selectitem> <f:selectitem itemlabel="Pesca" itemvalue="pesca"></f:selectitem></h:selectmanycheckbox>	Muestra un conjunto de checkbox, en los que el usuario puede seleccionar varios
<f:selectitem itemlabel="Deporte" itemvalue="deporte"></f:selectitem> <f:selectitem itemlabel="Lectura" itemvalue="lectura"></f:selectitem>	✓ Musica □ Cine ✓ Pesca
selectManyListbox	
<pre><h:selectmanylistbox value="#{cuestionario.lenguajes}"> <f:selectitem itemlabel="c" itemvalue="c"></f:selectitem> <f:selectitem itemlabel="c++" itemvalue="c++"></f:selectitem> <f:selectitem itemlabel="Java" itemvalue="java"></f:selectitem> <f:selectitem itemlabel="Visual basic" itemvalue="Visual basic"></f:selectitem> <f:selectitem itemlabel="Modula 2" itemvalue="modula2"></f:selectitem> </h:selectmanylistbox></pre>	Permite al usuario seleccionar varios elementos de una lista de elementos.
	c c++ Java Visual basic Modula 2
selectManyMenu	
<pre><h:selectmanymenu value="#{cuestionario.comidas}"> <f:selectitem itemlabel="Carnes" itemvalue="carnes"></f:selectitem> <f:selectitem itemlabel="Pescados" itemvalue="pescados"></f:selectitem> <f:selectitem itemlabel="Legumbres" itemvalue="legumbres"></f:selectitem> <f:selectitem itemlabel="Pastas" itemvalue="pastas"></f:selectitem></h:selectmanymenu></pre>	Permite al usuario seleccionar varios elementos de una lista de elementos.
<f:selectitem itemlabel="Sopas" itemvalue="sopas"></f:selectitem>	Carnes

UISelectOne

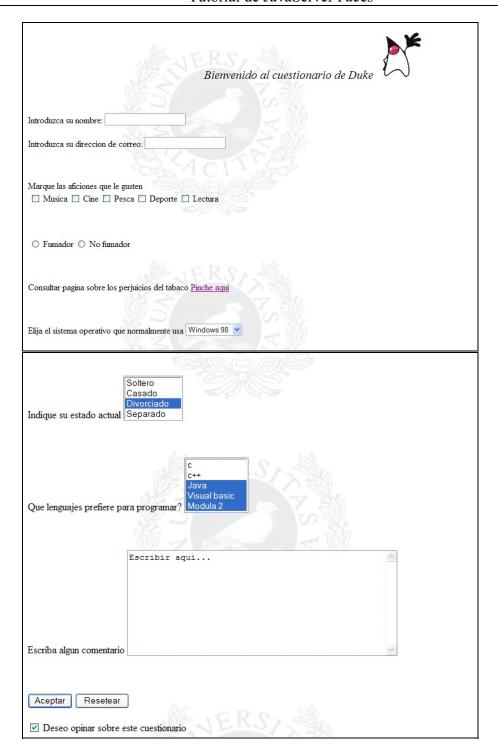
selectOneRadio	
<h:selectoneradio value="#{cuestionario.fumador}"> <f:selectitem itemlabel="Fumador" itemvalue="si"></f:selectitem> <f:selectitem itemlabel="No fumador" itemvalue="no"></f:selectitem></h:selectoneradio>	Permite al usuario seleccionar un elemento de un grupo de elementos
	O Fumador O No fumador
selectOneMenu	
<pre><h:selectonemenu value="#{cuestionario.sistema}"> <f:selectitem itemlabel="Windows 98" itemvalue="windows 98"></f:selectitem> <f:selectitem itemlabel="Windows XP" itemvalue="windows xp"></f:selectitem> <f:selectitem itemlabel="Suse" itemvalue="suse"></f:selectitem> <f:selectitem itemlabel="Red Hat" itemvalue="red hat"></f:selectitem> <f:selectitem itemlabel="Ubuntu" itemvalue="ubuntu"></f:selectitem> </h:selectonemenu></pre>	Permite al usuario seleccionar un elemento de un grupo de elementos
	Windows 98 Windows 98 Windows XP Suse Red Hat Ubuntu
selectOneListbox	
<pre><h:selectonelistbox value="#{cuestionario.estado}"> <f:selectitem itemlabel="Soltero" itemvalue="soltero"></f:selectitem> <f:selectitem itemlabel="Casado" itemvalue="casado"></f:selectitem> <f:selectitem itemlabel="Divorciado" itemvalue="divorciado"></f:selectitem> <f:selectitem itemlabel="Separado" itemvalue="separado"></f:selectitem> </h:selectonelistbox></pre>	Permite al usuario seleccionar un elemento de una lista de elementos
	Soltero Casado Divorciado Separado

UIPanel

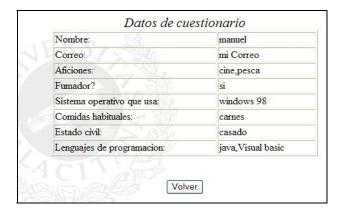


4.4 Un ejemplo completo de etiquetas básicas con JSF

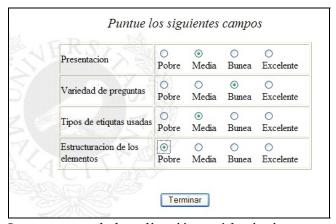
Con este ejemplo práctico se intenta ilustrar el uso de las diversas etiquetas que se pueden utilizar en un formulario. El ejemplo consiste en un cuestionario donde se nos pide rellenar cierta información. Una vez que se ha rellenado se aceptara el formulario mediante un botón, y navegaremos a una nueva página donde se muestran los datos introducidos, con la salvedad de que se puede optar por ir a una página intermedia (navegación dinámica) donde se puede opinar sobre el cuestionario. La apariencia del cuestionario es la siguiente:



El pagina que muestra los datos tiene el siguiente aspecto:



Y por último, la página opcional para opinar sobre la aplicación es:



La estructura de la aplicación será la siguiente:

```
Cuestionario
       index.html
       index.jsp
       mostrarInformacion.jsp
       test.jsp
       duke.gif
       paloma.gif
       estilos.css (Hoja de estilos)
WEB-INF/
       faces-config.xml
       web.xml
          classes/
               Cuetionario.class
        src/
                  (Opcional)
               Cuetionario.java
        lib/
               common-beanutils.jar
               common-collections.jar
               common-digester.jar
               common-logging.jar
               jsf-api.jar
               jsf-impl.jar
```

jstl.jar standard.jar

Esta aplicación parte de la página **index.jsp**, que muestra el cuestionario con las diversas etiquetas JSF, y cuyo código se expone a continuación:

```
< ht.ml>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <f:view>
          <head>
          <link href="estilos.css" rel="stylesheet" type="text/css"/>
          <f:loadBundle basename="mensajes" var="msjs"/>
             <h:outputText value="#{msjs.titulo}"/>
          </title>
      </head>
      <body>
          \langle \bar{h}: form \rangle
                      <h:outputText value="#{msjs.bienvenida}" styleClass="enfasis"/>
                      <h:graphicImage value="/duke.gif"/>
                      </center>
                      <br><br><
                       <h:outputText value="#{msjs.nombre}"/>
                       <h:inputText value="#{cuestionario.nombre}"/>
                       <h:outputText value="#{msjs.correo}"/>
                        <h:inputText value="#{cuestionario.correo}"/>
                  <br><br>>
                        <h:outputText value="#{msjs.aficiones}"/>
                        <h:selectManyCheckbox value="#{cuestionario.aficiones}">
                           <f:selectItem itemValue="musica" itemLabel="Musica"/>
                           <f:selectItem itemValue="cine" itemLabel="Cine"/>
                           <f:selectItem itemValue="pesca" itemLabel="Pesca"/>
                           <f:selectItem itemValue="deporte" itemLabel="Deporte"/>
<f:selectItem itemValue="lectura" itemLabel="Lectura"/>
                      </h:selectManyCheckbox>
                  <br><br>>
                          <h:selectOneRadio value="#{cuestionario.fumador}">
                               <f:selectItem itemValue="si" itemLabel="Fumador"/>
                               <f:selectItem itemValue="no" itemLabel="No fumador"/>
                      </h:selectOneRadio>
                  <br><br>>
                        <h:outputText value="#{msjs.tabaco}"/>
                           <h:outputLink value="http://www.hoylodejo.com/">
                           <h:outputText value="#{msjs.pinche}"/>
                      </h:outputLink>
                    <br><br>>
                        <h:outputText value="#{msjs.sistema}"/>
                        <h:selectOneMenu value="#{cuestionario.sistema}">
                          <f:selectItem itemValue="windows 98" itemLabel="Windows 98"/>
<f:selectItem itemValue="windows xp" itemLabel="Windows XP"/>
                          <f:selectItem itemValue="suse" itemLabel="Suse"/>
                          <f:selectItem itemValue="red hat" itemLabel="Red Hat"/>
<f:selectItem itemValue="ubuntu" itemLabel="Ubuntu"/>
                      </h:selectOneMenu>
                  <br><br>>
                        <h:outputText value="#{msjs.comidas}"/>
                        <h:selectManyMenu value="#{cuestionario.comidas}">
                            <f:selectItem itemValue="carnes" itemLabel="Carnes"/>
                           <f:selectItem itemValue="pescados" itemLabel="Pescados"/>
<f:selectItem itemValue="legumbres" itemLabel="Legumbres"/>
                           <f:selectItem itemValue="pastas" itemLabel="Pastas"/>
                           <f:selectItem itemValue="sopas" itemLabel="Sopas"/>
```

```
</h:selectManyMenu>
                   <br><br>>
                      <h:outputText value="#{msjs.estado}"/>
                      <h:selectOneListbox value="#{cuestionario.estado}">
                        <f:selectItem itemValue="soltero" itemLabel="Soltero"/>
                        <f:selectItem itemValue="casado" itemLabel="Casado"/>
                        <f:selectItem itemValue="divorciado" itemLabel="Divorciado"/>
                        <f:selectItem itemValue="separado" itemLabel="Separado"/>
                     </h:selectOneListbox>
                   <br><br>>
                      <h:outputText value="#{msjs.lenguajes}"/>
                      <h:selectManyListbox value="#{cuestionario.lenguajes}">
                        <f:selectItem itemValue="c" itemLabel="c"/>
                        <f:selectItem itemValue="c++" itemLabel="c++"/>
<f:selectItem itemValue="java" itemLabel="Java"/>
                       <f:selectItem itemValue="Visual basic" itemLabel="Visual
                        basic"/>
                        <f:selectItem itemValue="modula2" itemLabel="Modula 2"/>
                    </h:selectManyListbox>
                <br><br>>
                     <h:outputText value="#{msjs.comentario}"/>
                      <h:inputTextarea rows="10" cols="50"
                                                               value="Escribir
                     aqui..."/>
                <br><br>>
                    <h:commandButton value="#{msjs.aceptar}"
                    action="#{cuestionario.opinar}"/>
                    <h:commandButton value="#{msjs.resetear}"
                    type="reset"/><br><br>
                    <h:selectBooleanCheckbox value="#{cuestionario.quieroOpinar}"/>
                   <h:outputText value="#{msjs.opinar}"/>
         </h:form>
      </body>
   </f:view>
</ht.ml>
```

Como características a destacar, tras la importación de las librería *core* y *html_basic*, se hace uso de una hoja de estilo, mediante:

```
<link href="estilos.css" rel="stylesheet" type="text/css"/>
```

donde se indica que los estilos de letras que se utilizaran durante la aplicación, se recogen en el fichero **estilos.css**, y cuyo contenido se muestra a continuación:

```
body {
   background-image: url(paloma.gif);
}
.enfasis {
   font-style: italic;
   font-size: 1.3em;
}
.colores1 {
   font-size: 1.5em;
   font-style: italic;
   color: Blue;
   background: Yellow;
```

```
}
.colores2 {
  font-size: 0.8em;
  font-style: italic;
  color: Yellow;
  background: Blue;
}
```

se ha declarado un fondo de imagen (paloma.gif), un tipo de letra con estilo itálica y un cierto tamaño (*enfasis*) y dos tipos de letra mas con colores. De tal manera que se recurre a un tipo de letra de este fichero con el atributo *styleClass* como es el caso de las salida por pantalla del texto de bienvenida:

```
<h:outputText value="#{msjs.bienvenida}" styleClass="enfasis"/>
que usa el tipo de letra enfasis.
```

Todos los mensajes que aparecen por pantalla se han recogido en el dichero **mensajes.properties**, de manera que facilita la labor si se quieres mostrar la aplicación en otro idioma con solo cambiar este fichero. Al igual que con la hoja de estilos, al comienzo del fichero index.jsp, se carga este fichero de la siguiente manera:

```
<f:loadBundle basename="mensajes" var="msjs"/>
```

donde se indica que el fichero se llama mensajes (la extension debe ser .properties), y como identificar para acceder a los mensajes es msjs. Por lo tanto para mostrar el mensaje de bienvenida, sería: value="#{msjs.bienvenida}"

El contenido de **mensajes.properties** es el siguiente:

```
titulo=Aplicacion cuestionario
bienvenida=Bienvenido al cuestionario de Duke
nombre=Introduzca su nombre:
correo=Introduzca su direccion de correo:
aficiones=Marque las aficiones que le gusten
tabaco=Consultar pagina sobre los perjuicios del tabaco
comentario=Escriba algun comentario
sistema=Elija el sistema operativo que normalmente usa
aceptar=Aceptar
opinar=Deseo opinar sobre este cuestionario
comidas=Elija sus comida mas habituales
estado=Indique su estado actual
lenguajes=Que lenguajes prefiere para programar?
resetear=Resetear
pinche=Pinche aqui
datos=Datos de cuestionario
sal_nombre=Nombre:
sal_correo=Correo:
sal_fumador=Fumador?
   aficiones=Aficiones:
sal sistema=Sistema operativo que usa:
sal comidas=Comidas habituales:
```

```
sal_lenguajes=Lenguajes de programacion:
sal_estado=Estado civil:
volver=Volver

puntue=Puntue los siguientes campos
presentacion=Presentacion
variedad=Variedad de preguntas
tipos=Tipos de etiqutas usadas
estructura=Estructuracion de los elementos
terminar=Terminar
```

Donde se encuentran todos los mensajes de todas las páginas de la aplicación.

Al termino del formulario se encuentran dos botones, unos que acepta y tramina el formulario y otro que resetea los datos introducidos.

Por úlimo, se haya un elemento checkbox, que en el caso de que se marque, optaremos por ir a una pagina nueva para rellenar un breve formulario, opinando sobre la aplicación

Los datos de la aplicación están recogidos en el bean *Cuestionario*, contenido en las clase *Cuestionario*. El contenido de **Cuestionario.java** es el siguiente:

```
public class Cuestionario {
  private String nombre;
  private String correo;
  private String[] aficiones;
  private boolean quieroOpinar;
  private String fumador;
  private String sistema;
  private String[] comidas;
  private String estado;
  private String[] lenguajes;
   // ATRIBUTO: nombre
  public String getNombre() {
      return nombre;
  public void setNombre(String nuevoValor) {
     nombre = nuevoValor;
   // ATRIBUTO: correo
   public String getCorreo() {
     return correo;
  public void setCorreo(String nuevoValor) {
     correo = nuevoValor;
   // ATRIBUTO: aficiones
   public String[] getAficiones() {
       return aficiones;
  public void setAficiones(String[] nuevoValor) {
       aficiones = nuevoValor;
    // ATRIBUTO: quieroOpinar
   public boolean getQuieroOpinar() {
```

```
return quieroOpinar;
public void setQuieroOpinar(boolean nuevoValor) {
  quieroOpinar = nuevoValor;
// ATRIBUTO: listaAficiones
public String getListaAficiones() {
  return concatenate(aficiones);
// ATRIBUTO: fumador
public String getFumador() {
  return fumador;
public void setFumador(String nuevoValor) {
  fumador = nuevoValor;
 // ATRIBUTO: sistema
public String getSistema() {
  return sistema;
public void setSistema(String nuevoValor) {
  sistema = nuevoValor;
// ATRIBUTO: comidas
public String[] getComidas() {
    return comidas;
public void setComidas(String[] nuevoValor) {
    comidas = nuevoValor;
// ATRIBUTO: listaComidas
public String getListaComidas() {
  return concatenate(comidas);
// ATRIBUTO: estado
public String getEstado() {
  return estado;
public void setEstado(String nuevoValor) {
  estado = nuevoValor;
// ATRIBUTO: lenguajes
public String[] getLenguajes() {
    return lenguajes;
}
public void setLenguajes(String[] nuevoValor) {
    lenguajes = nuevoValor;
// ATRIBUTO: listaLenguajes
public String getListaLenguajes() {
  return concatenate(lenguajes);
```

```
// ATRIBUTO: opinar
  public String opinar(){
           if (quieroOpinar) {
               return "test";
           }else{
              return "mostrarInformacion";
       }
   // metodo para concetenar los elementos de las etiquetas de seleccion
   private static String concatenate(Object[] valores) {
      if (valores == null)
         return "";
      StringBuffer r = new StringBuffer();
      for (int i = 0; i < valores.length; ++i) {</pre>
         if (i > 0)
            r.append(',');
         r.append(valores[i].toString());
     return r.toString();
  }
}
```

Los datos utilizados en el formulario, tienen en **Cuestionario.java** sus metodos *get* y *set* correspondientes, para establecer y devolver sus valores, pero con una novedad, para las etiquetas multi selección, se ha utilizado una array de cadenas de caracteres, por lo que a la hora de devolver este valor para mostrarlo por pantalla, se ha implementado unos métodos *get*, que devuelven todos los valores seleccionados concatenados, de esta manera solo se devuelve una cadena de caracteres. Como método auxiliar, esta *concatenate*, que se encarga de concatenar una lista cualquiera, devolviendo un *String*.

El método opinar, se encarga de comprobar si la casilla de *selectCheckbox* de la página **index.jsp**, esta marcada, en cuyo caso, este método devolvera una cadena de caracteres con un valor u otro. De esta manera, esas cadenas de caracteres devueltas (test o mostrarInformacion) son usadas como acciones para activar una regla de navegación, así, se está haciendo uso de la navegación dinámica. Recuerde que estas reglas de navegación se recogen en el fichero **faces-config.xml**.

El test opcional para opinar sobre la aplicación, tras rellenar el formulario, está contenido en el fichero **test.jsp**, y contiene lo siguiente:

```
<h:outputText value="#{msjs.puntue}" styleClass="enfasis"/><br><br>
              <h:panelGrid columns="2" cellpadding="1" border="1" width="40%">
                       <h:outputText value="#{msjs.presentacion}"/>
                           <h:selectOneRadio>
                                 <f:selectItem itemValue="Pobre" itemLabel="Pobre"/>
<f:selectItem itemValue="Media" itemLabel="Media"/>
                                 <f:selectItem itemValue="Bunea" itemLabel="Bunea"/>
                                 <f:selectItem itemValue="Excelente"
itemLabel="Excelente"/>
                       </h:selectOneRadio>
                       <h:outputText value="#{msjs.variedad}"/>
                           <h:selectOneRadio>
                                 <f:selectItem itemValue="Pobre" itemLabel="Pobre"/>
                                 <f:selectItem itemValue="Media" itemLabel="Media"/>
                                 <f:selectItem itemValue="Bunea" itemLabel="Bunea"/>
                                 <f:selectItem itemValue="Excelente"
itemLabel="Excelente"/>
                       </h:selectOneRadio>
                       <h:outputText value="#{msjs.tipos}"/>
                           <h:selectOneRadio>
                                 <f:selectItem itemValue="Pobre" itemLabel="Pobre"/>
<f:selectItem itemValue="Media" itemLabel="Media"/>
                                 <f:selectItem itemValue="Bunea" itemLabel="Bunea"/>
                                 <f:selectItem itemValue="Excelente"
itemLabel="Excelente"/>
                       </h:selectOneRadio>
                       <h:outputText value="#{msjs.estructura}"/>
                           <h:selectOneRadio>
                                 <f:selectItem itemValue="Pobre" itemLabel="Pobre"/>
<f:selectItem itemValue="Media" itemLabel="Media"/>
<f:selectItem itemValue="Bunea" itemLabel="Bunea"/>
                                 <f:selectItem itemValue="Excelente"
itemLabel="Excelente"/>
                       </h:selectOneRadio>
              </h:panelGrid>
              <br><br><br>>
               <h:commandButton value="#{msjs.terminar}" action="mostrarInformacion"/>
          </h:form>
          </center>
       </body>
   </f:view>
</html>
```

Este formulario (el recogido en **test.jsp**) no efectúa ninguna operación con sus valores introducidos, ya que el cometido de esta página de navegación era ilustrar la navegación dinámica.

Para mostrar la información por pantalla, está el fichero **mostrarIntormacion.jsp**, el cual posee una botón al termino del formulario para regresar a la página inicial de la aplicación :

```
<html>
  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
  <f:view>
         <head>
         <link href="estilos.css" rel="stylesheet" type="text/css"/>
         <f:loadBundle basename="mensajes" var="msjs"/>
     </head>
     <body>
         <center>
         <h:panelGroup>
        <h:outputText value="#{msjs.datos}" styleClass="enfasis"/>
        </h:panelGroup>
         <h:panelGrid columns="2" cellpadding="1" border="1" width="40%">
            <h:outputText value="#{msjs.sal_nombre}"/>
               <h:outputText value="#{cuestionario.nombre}"/>
            <h:outputText value="#{msjs.sal correo}"/>
               <h:outputText value="#{cuestionario.correo}"/>
            <h:outputText value="#{msjs.sal_aficiones}"/>
               <h:outputText value="#{cuestionario.listaAficiones}"/>
            <h:outputText value="#{msjs.sal_fumador}"/>
               <h:outputText value="#{cuestionario.fumador}"/>
            <h:outputText value="#{msjs.sal sistema}"/>
               <h:outputText value="#{cuestionario.sistema}"/>
            <h:outputText value="#{msjs.sal comidas}"/>
               <h:outputText value="#{cuestionario.listaComidas}"/>
            <h:outputText value="#{msjs.sal_estado}"/>
               <h:outputText value="#{cuestionario.estado}"/>
            <h:outputText value="#{msjs.sal lenguajes}"/>
               <h:outputText value="#{cuestionario.listaLenguajes}"/>
         </h:panelGrid><br><br>
         <h:form>
             <h:commandButton value="#{msjs.volver}" action="volver"/>
         </h:form>
         </center>
     </body>
   </f:view>
</ht.ml>
```

Y por último está el fichero encargado de la navegación de la aplicación, es decir, **faces-confug.xml**:

```
<navigation-case>
            <from-outcome>volver</from-outcome>
            <to-view-id>/index.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
            <from-view-id>/test.jsp</from-view-id>
            <from-outcome>mostrarInformacion</from-outcome>
            <to-view-id>/mostrarInformacion.jsp</to-view-id>
      </navigation-case>
 </navigation-rule>
 <managed-bean>
      <managed-bean-name>cuestionario</managed-bean-name>
      <managed-bean-class>Cuestionario</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
 </managed-bean>
</faces-config>
```

Capitulo 5 Etiquetas JSF estándares

5.1 Introducción

Las librerías de etiquetas *core* y HTML forman un total de 42 etiquetas. Para usar las librerías de etiquetas JSF, debe importarlas mediante las directivas *taglib*. La convención establecida es usar el prefijo **f** para la librería *core* y el prefijo **h** para HTML. Las páginas JSF tienen una estructura similar a esto:

en donde se aprecia la importación de la librería de etiquetas **core**, se usan, entre otras aplicaciones, para manejo de eventos, atributos, conversión de datos, validadores, recursos y definición de la página, las cuales usan el prefijo **f**, y la librería de etiquetas **html_basic**, su utilización es básicamente para la construcción de formularios y demás elementos de interfaz de usuario, que usa el prefijo **h**.

5.2 Etiquetas Core

En la siguiente tabla 5.1 vemos las etiquetas **core**:

Etiqueta	Descripción
view	Crea una vista
subview	Crea un subvista
facet	Añade una faceta a un componente
attribute	Añade un atributo (clave/valor) a un componente
param	Añade un parámetro a un componente
actionListener	Añade una acción a un componente
valueChangeListener	Añade un nuevo valor al oyente de un componente
convertDateTime	Añade fecha a un componente
convertNumber	Añade una conversión de número a un componente
validator	Añade un validador a un componente
validateDoubleRange	Valida un rango de tipo double para valores de componentes

Etiqueta	Descripción
validateLength	Valida la longitud del valor de un componente
validateLongRange	Valida un rango de tipo <i>long</i> para valores de componentes
loadBundle	Carga el origen de un elemento Bundle
selectitems	Especifica elementos para seleccionar uno o varios elementos
selectitem	Especifica un elemento para seleccionar uno o varios elementos
verbatim	Añade una marca a una página JSF

5.3 Etiquetas HTML

En la siguiente tabla vemos las etiquetas **HTML**:

Etiqueta	Descripción
form	Formulario HTML
inputText	Simple linea de texto de entrada
inputTextarea	Múltiples lineas de texto de entrada
inputSecret	Contraseña de entrada
inputHidden	Campo oculto
outputLabel	Etiqueta para otro componente
outputLink	Enlace HTML
outputFormat	Formatea un texto de salida
outputText	Simple linea de texto de salida
commandButton	Botón: submit, reset o pushbutton
commandLink	Enlace asociado a un botón pushbutton
message	Muestra el mensaje mas reciente para un componente
messages	Muestra todos los mensajes
graphicImage	Muestra una imagen
selecOneListbox	Selección simple para lista desplegable
selectOneMenu	Selección simple para menu

Etiqueta	Descripción
selectOneRadio	Conjunto de botones radio
selectBooleanCheckbox	Checkbox
selectManyCheckbox	Conjunto de checkboxes
selectManyListbox	Selección múltiple de lista desplegable
selectManyMenu	Selección múltiple de menu
panelGrid	Tabla HTML
panelGroup	Dos o más componentes que son mostrados como uno
dataTable	Tabla de datos
column	Columna de un dataTable

Se pueden agrupar las etiquetas HTML bajo las siguientes categorías:

- Entrada(*Input*...)
- Salida (*Output*...)
- Comandos (commandButton y commandLink)
- Selecciones (checkbox, listbox, menu, radio)
- Paneles (panelGrid)
- Tablas de datos (dataTable)
- Errores y mensajes (message, messages)

5.3.1 Atributos comunes

Existen tres tipos de atributos de etiquetas HTML:

- Básicos
- HTML 4.0
- Eventos DHTML

5.3.1.1 Atributos básicos

Atributo	Tipo de componente	Descripción
id	A(25)	Identificador para un componente

Atributo	Tipo de componente	Descripción
binding	A(25)	Referencia a un componente que puede ser usado en un backing bean
rendered	A(25)	Un boolean, false sustituye el renderizado
styleClass	A(23)	Nombre de la clase del stylesheet (CSS)
value	I,O,C(19)	Valor de un componente
converter	I,O(11)	Convierte el nombre de una clase
validator	I(11)	Nombre de la clase de un validador
required	I(11)	Un boolean, si es true se requiere introducir un valor en un campo
A: Cualquier etiqueta I:etiquetas de entrada O:etiquetas de salida C:comandos (n):número de etiquetas		

5.3.1.2 Atributos HTML 4.0

Atributo	Descripción
accesskey (14)	Una clave, típicamente combinada con un sistema definido de meta claves, esto da una referencia a un elemento
accept (1)	lista separada por comas de los tipos contenidos en un formulario
accept-charset (1)	lista separada por comas o espacios de caracteres codificados de un formulario
alt (4)	Texto alternativo para elementos que no son texo, como imágenes o <i>applets</i>
border (4)	Valor en pixel, para el grosor de un elemento border
charset (3)	Carácter codificado para un enlace
coords (2)	Coordenadas de un elemento, tal como un círculo, un rectángulo o un polígono
dir (18)	Dirección de un texto. Los valores válidos son ltr (de izquierda a la derecha) y rtl (de derecha a izquierda)
disabled (11)	Desactiva un elemento de entrada o un botón
hreflang (2)	Lenguaje especificado para lo especificado con el atributo <i>href</i>
lang (20)	Lenguaje base para atributos de elementos y textos

Tutorial de JavaServer Faces

Atributo	Descripción
maxlength (2)	Número máximo de caracteres para un campo texto
readonly (11)	Estado de solo lectura, para un campo de entrada, un texto podría ser seleccionado pero no editado
rel (2)	Relación entre el documento actual y el enlace especificado con el atributo <i>href</i>
rev (2)	Enlace de regreso para el valor especificado en un href en un documento actual
rows (1)	Número de columnas de un área de texto
shape (2(Forma de una región. Valores válidos: default, rect, circle, poly.
size (4)	Tamaño de un campo de entrada
style (23)	Información de un estilo
tabindex (14)	Valor numérico que especifica in índice
target (3)	Nombre de un <i>frame</i> , en el que un documento es abierto
tittle (22)	Título que describe un elemento
type (4)	Tipo de un enlace
width (3)	Anchura de un elemento
(n): número de etiquetas	

5.3.1.3 Atributos de eventos DHTML

Atributo	Descripción
onblur (14)	Orígenes de elementos perdidos
onchange (11)	Valor de elementos cambiados
onclick (17)	El botón del ratón es pulsado sobre un elemento
ondbclick (18)	El botón de ratón es pulsado dos veces sobre un elemento
onfocus (14)	Un elemento recibe un origen
onkeydown (18)	Tecla presionada
onkeypress (18)	Tecla presionada y seguidamente soltada
onkeyup (18)	Tecla soltada
onmousedown (18)	El botón del ratón es presionado sobre un elemento

Atributo	Descripción
onmousemove (18)	El ratón se mueve sobre un elemento
onmouseout (18)	El ratón abandona el área de un elemento
onmouseover (18)	El ratón se mueve dentro de un elemento
onmouseup (18)	Botón del ratón soltado
onreset (1)	El formulario es reseteado
onselect (11)	Texto seleccionado en un texto de entrada
onsubmit (1)	Formulario aceptado
(n): número de etiquetas	

5.4 Formularios

La etiqueta **h:form** genera un formulario, el cual puede contener cualquier etiqueta estándar, como botones, menu, listas, textos, etc...teniendo una estructura tal que así:

```
<h:form>
...
(* codigo *)
...
</h:form>
```

Por ejemplo se puede generar un formulario básico donde se pida el nombre, un password, como se vio en el ejemplo básico del primer capítulo.

Atributos para h:form

Atributos	Descripción
binding, id, rendered, styleClass	Atributos básicos
accept, acceptcharset, dir, enctype, lang, style, target, title	Atributos HTML 4.0

Atributos	Descripción
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onreset, onsubmit	Atributos de eventos DHTML

Esta etiqueta genera un elemento HTML llamado *form*, por ejemplo, si una página JSF llamada /*index.jsp* contiene la etiqueta **h:form** sin atributos, el renderizado genera HTML tal que así:

<form id="_id0" method="post" action="/forms/faces/index.jsp"
enctype="application/x-www-form-urlencoded">
A continuación se listan los diversos elementos que puede contener un formulario.

5.5 Campos de texto y áreas de texto

Las entradas de texto son el soporte principal de las aplicaciones web, JSF soporta tres variantes, representadas por las siguientes etiquetas:

- h:inputText (permite al usuario introducir un string)
- h:inputSecret (permite al usuario introducir un string sin que aparezca el string real en el campo)
- h:inputTextarea (permite al usuario introducir un texto multi-líneas)

Las tres etiquetas usan atributos similares que se muestran a continuación:

Atributos	Descripción
cols	(solo para h:inputTextarea), número de columnas
inmediate	Procesa una validación antes de la cuenta en el ciclo de vida
redisplay	(solo para h:inputSecret), cuando es <i>true</i> , se retiene el valor del campo al recargar la página
required	Requiere la entrada de un componente cuando un formulario es tramitado
rows	(solo para h:inputTextarea), número de filas
binding, converter, id, rendered, required, styleClass, value, validator	Atributos básicos

Tutorial de JavaServer Faces

Atributos	Descripción
accesskey, alt, dir, disabled, lang, maxlength, readonly, size, style, tabindex, title	Atributos HTML 4.0, (alt, maxlength, y size no son aplicables para h:inputTextarea)
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onselect	Atributos de eventos DHTML

Para ver el uso de estas etiquetas junto con algunos atributos, se muestran los siguientes ejemplos:

Ejemplo	Resulado
<h:inputtext maxlength="6" size="10" value="1234567890"></h:inputtext>	123456
<h:inputtext size="4" value="abcdefgh"></h:inputtext>	abcdef
<h:inputtext readonly="true" value="miBean.texto"></h:inputtext>	Solo lectura
<pre><h:inputtext style="color: Yellow; background: Teal;" value="Con colores"></h:inputtext></pre>	Con colores
<h:inputsecret redisplay="true" value="clave"></h:inputsecret>	••••
	(Tras recargar la página)
<h:inputsecret redisplay="false" value="clave"></h:inputsecret>	
	(Tras recargar la página)
<h:inputtextarea cols="10" rows="3" value="Escribir"></h:inputtextarea>	Escribir.

En el primer ejemplo, se ilustra el uso del atributo *maxlength*, el cual indica el número de caracteres máximos que se puedes introducir en el campo de entrada, este atributo es preciso, a diferencia de *size*, el cual muestra el numero máximo de caracteres que se muestran

por pantalla, pero debido a que las fuentes de caracteres varían en anchura, este atributo no es preciso, como se aprecia con el segundo ejemplo, donde a pesar de que se indica que se van a mostrar como máximo, cuatro caracteres, se ven seis de ellos.

El tercer ejemplo, muestra un valor que es solo de lectura, por lo tanto, en el *bean* donde se encuentre implementado(en este caso, *miBean*) el atributo texto, solo precisará del método *get*, es decir:

```
private String texto = "Solo lectura";
public String getTexto() {
    return texto;
}
```

Con el ejemplo de h:inputSecret, se ve el uso del atributo redisplay, se aprecia como tras recargar la página, en el campo de entrada se retiene el valor introducido, quinto ejemplo, y como se pierde el valor tras recargar, sexto ejemplo.

Por último, h:inputTextarea, usa los atributos para definir el número de filas(*rows*), y número de columnas(*cols*). El atributo *cols* es algo impreciso, al igual que *size*.

Luego están las etiquetas para mostrar texto y gráficos:

- h:outputText (muestra una línea de texto)
- h:outputFormat (formatea una texto con los parámetros definidos en su cuerpo)
- h:graphicImage (muestra una imagen)

Cuyos atributos se muestran a continuación por separado.

Atributos de **h:outputText** y **h:outputFormat**

Atributos	Descripción
escape	Si se pone a <i>true</i> , los caracteres <,> y & se convierten a <, > y & respectivamente, por defecto es false
binding, converter, id, rendered, styleClass, value	Atributos básicos
style, title	Atributos HTML 4.0

Atributos de h:graphicmage

Atributos	Descripción
binding, id, rendered, styleClass, value	Atributos básicos
alt, dir, height, ismap, lang, longdesc, style, title, url, usemap, width	Atributos HTML 4.0

Tutorial de JavaServer Faces

Atributos	Descripción
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	Atributos de eventos DHTML

Para ver el uso de estas etiquetas junto con algunos atributos, se muestran los siguientes ejemplos:

<h:outputtext value="Hola Mundo!"></h:outputtext>	Hola Mundo!
<h:outputtext value="Nombre
#{formulario.nombre}"></h:outputtext>	Nombre Manuel
<pre><h:outputformat value="{0} es estudiante de la Universidad de {1}"></h:outputformat></pre>	Manuel es estudiante de la Universidad de Malaga
<h:graphicimage value="duke.gif"></h:graphicimage>	
<h:graphicimage style="border: thin solid black" value="duke.gif"></h:graphicimage>	

JSF también provee una etiqueta para campos ocultos, h:inputHidden.

• h:inputHidden

Atributos de h:inputHidden

Atributos	Descripción
binding, converter, id, immediate, required, validator,	Atributos básicos
value,	
valueChangeListener	

5.6 Botones y enlaces

JSF provee las siguientes etiquetas para botones y enlaces:

- h:commandButton
- h:commandLink
- h:outputLink

Tanto *commandButton* como *commandLink*, representan componentes de comandos, con ellos, se disparan e invocan acciones que llevan asociados, de esta manera, con una acción se puede activar una regla de navegación, por ejemplo.

En cuanto a outputLink genera un enlace HTML que puede tener una imagen o una página web como destino.

Atributos de h:commandButton y h:commandLink

Atributos	Descripción
action	Si es un String: directamente especifica una acción que se usa para activar una regla de navegación Si es una funcion: tendrá la signatura String funcion(); y el string devuelto será la acción que active la regla de navegación
actionListener	
charset	
image	(solo para commandButton) una imagen se depliega sobre el botón.
immediate	Un boleano, si es false(por defecto), las acciones son invocadas al término de la petición del ciclo de vida, si es true, se invocan al principio del ciclo

Tutorial de JavaServer Faces

Atributos	Descripción
type	Para h:commandButton, el tipo de elemento generado, submit, reset o button.
	Para h:commandLink, el tipo del destino del enlace, text/html, image/gif, o audio/basic
value	Etiqueta dispuesta sobre el botón o el enlace. Puede ser tanto un String como una referencia
accesskey, alt, binding, id, lang, rendered, styleClass, value	Atributos básicos
coords (h:commandLink), dir,disabled, hreflang (h:commandLink), lang, readonly, rel (h:commandLink), rev (h:commandLink), shape (h:commandLink), style, tabindex, target (h:commandLink), title, type	Atributos HTML 4.0
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	Atributos de eventos DHTML

Atributos de h:outputLink

Atributos	Descripción
accesskey, alt, binding, id, lang, rendered, styleClass, value	Atributos básicos
charset, coords, dir, hreflang, lang, rel, rev, shape, style, tabindex, target, title, type	Atributos HTML 4.0
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	Atributos de eventos DHTML

Para ver el uso de estas etiquetas junto con algunos atributos, se muestran los siguientes ejemplos:

Tutorial de JavaServer Faces

<h:commandbutton type="submit" value="Aceptar"></h:commandbutton>	Aceptar
<h:commandbutton type="reset" value="resetear"></h:commandbutton>	resetear
<h:commandbutton disabled="#{not formulario.botonActivo}" value="disabled"></h:commandbutton>	desactivado
<h:commandbutton type="reset" value="#{formulario.textoBoton}"></h:commandbutton>	Boton
<h:commandlink></h:commandlink>	registrar
<h:commandlink style="font-style: italic"></h:commandlink>	Pinche aqui
<pre><h:commandlink></h:commandlink></pre>	Ir a la pagina de Duke
<h:outputlink value="http://java.net"></h:outputlink>	java.net
<pre><h:outputlink value="introduccion"></h:outputlink></pre>	<u>Introduccion</u>

<pre><h:outputlink title="Ir a la conclusion" value="conclusion"></h:outputlink></pre>	Conclusion Ir a la conclusion
<pre><h:outputlink title="Ir a la tabla de contenidos" value="tabla"></h:outputlink></pre>	Tabla de Contenidos Ir a la tabla de contenidos

Los dos primeros ejemplos muestran dos botones típicos de un formulario, el primero de tipo *submit*, para tramitar el formulario una vez rellenado, y el segundo de tipo *reset*, para resetear los valores.

El tercer ejemplo se basa el una variable booleana (botonActivo) contenida en el *bean formulario*, para desactivar o no el botón.

En cuanto a las etiquetas **outputLink**, usan el atributo *title*, con el que se muestra un titulo al pasar el putero del ratón sobre los enlace.

5.7 Etiquetas de selección

Para la selección de diversos elementos, ya sean selecciones únicas o múltiples, se dispone de las siguientes etiquetas:

- h:selectBooleanCheckbox
- h:selectManyCheckbox
- h:selectOneRadio
- h:selectOneListbox
- h:selectManyListbox
- h:selectOneMenu
- h:selectManyMenu

selectBooleanCheckbox es la etiqueta de selección mas simple, lo puede asociar con un atributo booleano de un *bean*. Si se quiere un conjunto de *checkbox*, está **selectManyCheckbox**.

Las etiquetas que comienzan con *selectOne*, le permiten elegir un elemento de entre un conjunto de elementos, mientra que las que comiencen por *selectMany* se permiten seleccionar mas de uno.

Todas las etiquetas de selección presentan idénticos atributos, que se muestran a continuación:

Atributos	Descripción
disabledClass	clase CSS para elementos deshabilitados, solo para h:selectOneRadio y h:selectManyCheckbox
enabledClass	clase CSS para elementos habilitados, solo para h:selectOneRadio y h:selectManyCheckbox
layout	Especifica como se colocan los elementos, lineDirection (horizontal) o pageDirection (vertical), solo para h:selectOneRadio y h:selectManyCheckbox
binding, converter, id, immediate, styleClass, required, rendered, validator, value, valueChangeListener	Atributos básicos
accesskey, border, dir, disabled, lang, readonly, style, size, tabindex, title	Atributos HTML 4.0 (border solo es aplicable a h:selectOneRadio y h:selectManyCheckbox size solo es aplicable a h:selectOneListbox y h:selectManyListbox)
onblur, onchange, onclick, ondblclick, onfocus, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup, onselect	Atributos de eventos DHTML

5.7.1 selectBooleanCheckbox

Representa un simple *checkbox*, que puede asociar a un atributo booleano de un *bean*, por ejemplo, si en su página JSF tiene algo como esto:

<h:selectBooleanCheckbox value="#{cuestionario.recibirInformacion}"/>

en el bean que declare, debe tener los siguientes métodos de lectura/escritura:

```
private boolean recibirInformacion;

// ATRIBUTO: recibirInformacion
public boolean getRecibirInformacion() {
    return recibirInformacion;
}
public void setRecibirInformacion (boolean nuevoValor) {
    recibirInformacion = nuevoValor;
```

}

Quedando como resultado en el navegador:

✓ Deseo recibir informacion

5.7.2 selectManyCheckbox

Puede crear un grupo de *checkboxes* con la etiqueta *selectManyCheckbox*, como su nombre indica, puede seleccionar uno o mas elementos de un grupo de *checkboxes*. Dentro del cuerpo de **h:selectManyCheckbox**, se declara cada elemento con la etiqueta **f:selectItem**, usando el atributo *itemValue*, para establecer su valor de selección y la etiqueta *itemLabel*, para establecer la etiqueta que aparece junto a la casilla en el navegador, de manera que puede tener un aspecto como el siguiente:

```
<h:selectManyCheckbox value="#{cuestionario.aficiones}">
  <f:selectItem itemValue="musica" itemLabel="Musica"/>
  <f:selectItem itemValue="cine" itemLabel="Cine"/>
  <f:selectItem itemValue="pesca" itemLabel="Pesca"/>
  </h:selectManyCheckbox>
```

Para este ejemplo, los valores de las casillas marcadas, se recogen en el atributo *aficiones*, dentro del bean *cuestionario*.

Quedando en el navegador:



5.7.3 selectOneRadio

Con esta etiqueta se selecciona un elemento de entre una lista de ellos, y un vez mas, cada uno de los elementos se declara en el cuerpo de **h:selectOneRadio** con la etiqueta **f:selectItem**. Un ejemplo puede ser:

```
<h:selectOneRadio value="#{cuestionario.fumador}">
  <f:selectItem itemValue="si" itemLabel="Fumador"/>
  <f:selectItem itemValue="no" itemLabel="No fumador"/>
</h:selectOneRadio>
```

Para este ejemplo, los valores de las casillas marcadas, se recogen en el atributo *fumador*, dentro del bean *cuestionario*.

Quedando en el navegador: O Fumador O No fumador

Tanto para selectOneRadio como para selectManyMenu, se disponen de algunos atributos en común:

- border
- style
- layout

El atributo **border**, determina la anchura del borde que rodea a las etiquetas, por ejemplo si establece **border="2"** para *selectManyCheckbox*, quedaría algo como:

☐ Musica ☐ Cine ☐ Pesca ☐ Deporte ☐ Lectura				
con style podra decorar a su gusto la etiqueta, por ejemplo, con:				
style="font-style: italic;color: blue; background: Yellow"				
se quedaría en el navegador:				

☐ Musica ☐ Cine ☐ Pesca ☐ Deporte

Por último, el atributo layout, indica la disposición de la etiqueta, ya sea horizontal, **layout="lineDirection"** o vertical, **layout="pageDirection"**, por ejemplo:

Lectura



5.7.4 selectOneListbox

Para desplegar una lista de elementos, y seleccionar uno de ellos, tiene un aspecto similar al siguiente:

<h:selectOneListbox value="#{cuestionario.estado}">

```
<f:selectItem itemValue="soltero" itemLabel="Soltero"/>
<f:selectItem itemValue="casado" itemLabel="Casado"/>
```

<f:selectItem itemValue="divorciado" itemLabel="Divorciado"/>

<f:selectItem itemValue="separado" itemLabel="Separado"/>

</h:selectOneListbox>

Para este ejemplo, los valores de las casillas marcadas, se recogen en el atributo *estado*, dentro del bean *cuestionario*.

Quedando en el navegador:



5.7.5 selectManyListbox

Similar a la anterior etiqueta, pero permitiéndole seleccionar mas de un elemento, un ejemplo de su uso puede ser el siguiente:

<h:selectManyListbox value="#{cuestionario.lenguajes}">

<f:selectItem itemValue="c" itemLabel="c"/>

<f:selectItem itemValue="c++" itemLabel="c++"/>

<f:selectItem itemValue="java" itemLabel="Java"/>

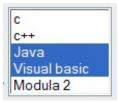
<f:selectItem itemValue="Visual basic" itemLabel="Visual basic"/>

<f:selectItem itemValue="modula2" itemLabel="Modula 2"/>

</h:selectManyListbox>

Para este ejemplo, los valores de las casillas marcadas, se recogen en el atributo *lenguajes*, dentro del bean *cuestionario*.

Quedando en el navegador:



5.7.6 selectOneMenu

Se permite elegir un elemento de un menu de elementos, su código se refleja en el siguiente ejemplo:

<h:selectOneMenu value="#{cuestionario.sistema}">

```
<f:selectItem itemValue="windows 98" itemLabel="Windows 98"/>
```

- <f:selectItem itemValue="windows xp" itemLabel="Windows XP"/>
- <f:selectItem itemValue="suse" itemLabel="Suse"/>
- <f:selectItem itemValue="red hat" itemLabel="Red Hat"/>
- <f:selectItem itemValue="ubuntu" itemLabel="Ubuntu"/>
- </h:selectOneMenu>

Para este ejemplo, los valores de las casillas marcadas, se recogen en el atributo *sistema*, dentro del bean *cuestionario*.

Quedando en el navegador:



5.7.7 selectManyMenu

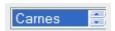
Igual que su predecesor pero puede seleccionar mas de un elemento, un ejemplo puede ser:

```
<h:selectManyMenu value="#{cuestionario.comidas}">
```

- <f:selectItem itemValue="carnes" itemLabel="Carnes"/>
- <f:selectItem itemValue="pescados" itemLabel="Pescados"/>
- <f:selectItem itemValue="legumbres" itemLabel="Legumbres"/>
- <f:selectItem itemValue="pastas" itemLabel="Pastas"/>
- <f:selectItem itemValue="sopas" itemLabel="Sopas"/>
- </h:selectManyMenu>

Para este ejemplo, los valores de las casillas marcadas, se recogen en el atributo *comidas*, dentro del bean *cuestionario*.

Quedando en el navegador:



5.8 Mensajes

Típicamente, los mensajes son asociados a componentes particulares, mostrando por pantalla errores de conversión o validación. Aunque el mensaje mas usado, se el mensaje de error, los mensajes se pueden clasificar en cuatro variedades:

Información

- Alertas
- Errores
- Fatales

Todos los mensajes pueden contener un sumario y un detalle, por ejemplo, una sumario puede ser *Invalid Entry*(entrada no válida), y un detalle, "el número introducido es mayor de lo permitido".

Las aplicaciones JSF, usan dos etiquetas en sus páginas JSF, para mostrar por pantalla los mensajes, h:messages y h:message

Mientras que **h:messages** atañe a todos los mensajes en general, que puedan ser mostrados, **h:message** muestra un mensaje para un único componente

Ambas etiquetas tienes los atributos que se muestran a continuación:

Atributos	Descripción
errorClass	Clase CSS aplicada a mensajes de error
errorStyle	Estilo CSS aplicado a mensajes de error
fatalClass	Clase CSS aplicada a mensajes fatales
fatalStyle	Estilo CSS aplicado a mensajes fatales
globalOnly	(solo para h:messages). Para mostrar solo los mensajes globales por pantalla, por defeco false
infoClass	Clase CSS aplicada mensajes de informacion
infoStyle	Estilo CSS aplicado mensajes de informacion
layout	(solo para h:messages). Especifica la estructura del mesaje, tabla (table) o lista (list)
showDetail	Un booleano que determina si el detalle de un mensaje es mostrado. Por defecto es false para h:messages y true para h:message
showSummary	Un booleano que determina si el sumario de un mensaje es mostrado. Por defecto es <i>true</i> para h:messages y false para h:message
warnClass	Clase CSS para mensajes de alerta
warnStyle	Estilo CSS para mensajes de alerta
binding, id, rendered, styleClass	Atributos básicos
style, title	Atributos HTML 4.0

Por ejemplo, suponga que desea emitir mensajes de error tras rellenar un formulario que contiene un campo nombre. Este formulario queda recogido en un fichero .jsp como se ha visto hasta ahora. La etiqueta que nos pida el nombre podría ser:

```
<h:inputText id="nombre" value="#{usuario.nombre}" required="true"/>
```

nótese que se le ha asignado un identificador *id*, con el valor *nombre*, y con el atributo *required*, se indica que no se puede dejar vacío dicho campo. Después de esta etiqueta puede emitir un mensaje de error en el caso de que la validación se errónea, es decir, que se deje el campo sin rellenar, de la siguiente manera:

```
<h:message for="nombre" errorClass="errores"/>
```

donde se usa la etiqueta **h:message** para el componente con identificador *nombre*, y hace uso de la clase CSS para mensajes de errores, es decir, al inicio del fichero .jsp que contiene estas etiquetas, se debe invocar la ruta del fichero de estilos que nos formatea el texto de salida, algo tal que así:

```
k href="estilos.css" rel="stylesheet" type="text/css"/>
```

en la ruta se indica que los estilos de letra están recogidos en el fichero estilos.css, el cual alberga los estilos para los mensajes de errores, por ejemplo:

```
.errores {
  font-style: italic;
  color: red;
}
```

con lo que los mensajes se mostrarán en itálica y en color rojo.

5.9 Paneles

Hasta ahora se ha podido ver como los elementos de un formulario se organizaban mediante tablas HTML, pero este trabajo es tedioso y propendo a errores. Para facilitar las cosas está la etiqueta **h:panelGrid**, la cual genera tablas HTML. Con el atributo *column* (por defecto 1), especifica el número de columnas que tendrá su tabla, por ejemplo:

```
<h:panelGrid column="3">
...
</h:panelGrid>
```

Los elementos serán ubicados dentro de la tabla, de izquierda a derecha y de arriba hacia abajo. A continuación se listan los atributos para la etiqueta **h:panelGrid**

Tutorial de JavaServer Faces

Atributos	Descripción
bgcolor	Color del fondo de la tabla
border	Anchura del borde de la tabla
cellpadding	Espacio para las celdas de la tabla
cellspacing	Espacio entre las celdas de la tabla
columnClasses	Lista separada por comas, de las clases CSS para las columnas de la tabla
columns	Número de columnas
footerClass	Clase CSS para el pie de la tabla
headerClass	Clase CSS para la cabecera de la tabla
rowClasses	Lista separada por comas, de las clases CSS para las filas de la tabla
binding, id, rendered, style- Class, value	Atributos básicos
dir, lang, style, title, width	Atributos HTML 4.0
onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	Atributos de eventos DHTML

En conjunción con esta etiqueta, está **h:panelGroup**, esta agrupa dos o mas componentes que son tratados como uno solo, por ejemplo, puede agrupa un campo de entrada de texto y un mensaje de error como un solo elemento de la siguiente manera:

Esta es una etiqueta simple con unos pocos atributos que se muestran a continuación:

Atributos	Descripción	
binding, id, rendered, styleClass, value	Atributos básicos	
style	Atributos HTML 4.0	

5.10 Data Table

Las tablas HTML son muy usadas a la hora de estructurar y mostrar por pantalla los componentes de una aplicación web. JSF le permite organizar dichos elementos mediante la etiqueta **h:dataTable**

5.10.1 La etiqueta h:dataTable

La etiqueta **h:dataTable** itera sobre los datos contenidos en la tabla, para formar la tabla HTML. A continuación se muestra un ejemplo de su uso:

El atributo *value* representa los datos sobre los que *h:dataTable* itera; estos datos pueden ser:

- Un array
- Una instancia de java.util.List
- Una instancia de java.sql.ResultSet
- Una instancia de javax.servlet.jsp.jstl.sql.Result
- Una instancia de javax.faces.model.DataModel

El identificador que llama a cada elemento es especificado con el atributo *var*.

Cada columna puede contener un número ilimitado de componentes como elementos adicionales.

Atributos para la etiqueta h:dataTable

Atributos	Descripción		
bgcolor	Color de fondo de la tabla		
border	Anchura del borde de la tabla		
cellpadding	Espacio de las celdas de la tabla		
cellspacing	Espacio entre las celdas de la tabla		
columnClasses	Lista separada por comas de las clases CSS para las columnas		
first	Índice de la primera fila mostrada en la tabla		
footerClass	Clase CSS para el pie de tabla		
headerClass	Clase CSS para la cabecera de tabla		
rowClasses	Lista separada por comas de las clases CSS para las filas		
rules	Especificación para las lineas dibujadas entre las celdas. Los valores permitidos son: groups, rows, columns, all		
var	Nombre de la variable creada por la tabla que representa el valor del elemento actual		
binding, id, rendered, style- Class, value	Atributos básicos		
dir, lang, style, title, width	Atributos HTML 4.0		
onclick, ondblclick, onkeydown, onkeypress, onkeyup, onmousedown, onmousemove, onmouseout, onmouseover, onmouseup	Atributos de eventos DHTML		

5.10.2 Cabeceras y pie de tabla

En el caso de que desee decorar mas su tabla o aclarar el contenido de la misma, puede añadir los elementos de cabecera (*header*) y pie (*footer*) de tabla. Estos elementos son facetas adicionales a la tabla y cuyo uso se muestra a continuación:

<h:dataTable>
...
<h:column>

5.10.3 Componentes JSF incluidos en celdas tabla

Se puede ubicar cualquier componente JSF dentro de una columna de tabla. Su uso no difiere del visto hasta ahora, declarando los componentes dentro de las columnas de la tabla.

5.10.4 Editando celdas

Para editar la celda de una tabla, basta con colocar un componente de entrada en las celdas que quiere que sean editadas.

5.10.5 Estilos para filas y columnas

h:dataTable posee atributos para las clases CSS sobre los siguientes elementos:

- La tabla como un todo(*styleClass*)
- Cabeceras y pie de tablas (headerClass y footerClass)
- columnas individuales (columnClasses)
- filas individuales (rowClasses)

5.10.6 Técnicas de scroll

Usar una barra de scroll es una técnica muy simple. Para ello se usa la directiva **div**. Su sintaxis es la siguiente:

Como verá, se ha usado la directiva **div** con algunos atributos, como son el estilo (*style*), el ancho de la tabla (*width*) y la longitud de la tabla (*heigth*).

5.10.7 Un breve ejemplo de la etiqueta h:dataTable

Para ver el uso de esta etiqueta, se va a mostrar un sencillo ejemplo que muestra por pantalla dos tablas. La primera muestra un listado de nombres en tres columnas con atributos diferentes, estos nombres están recogidos en el bean *dataTable*. La segunda tabla introduce algunos componentes JSF en sus celdas. Esta segunda tabla hace uso de unos valores numéricos recogidos en el bean *listaNumeros*, y que se encuentra inicializado el la propia declaración del bean, es decir, en el fichero **faces-config.xml**. En la siguiente figura se muestra su resultado por pantalla:

Una simple tabla con nombres

Jose,	Jose	<u>Jose</u>
Manuel,	Manuel	<u>Manuel</u>
Andres,	Andres	<u>Andres</u>
Carlos,	Carlos	Carlos

Una simple tabla con componentes JSF

Numeros	Campo Texo	Boton	Checkbox	Enlaces	radio boton	listbox
1	1	1		1	⊙ Si ○ no	siempre a menudo alguna vez v
2	2	2		2	○ Si ⊙ no	siempre a menudo alguna vez
3	3	3		3	○ Si ⊙ no	siempre a menudo alguna vez
4	4	4		4	⊙ Si ○ no	a menudo 🔥 alguna vez nunca 💌
5	5	5		<u>5</u>	⊙ Si ○ no	siempre a menudo alguna vez

Como estructura de directorios del ejemplo, se tiene lo siguiente:

```
dataTable
        index.html
        index.jsp
        estilos.css
        WEB-INF/
                faces-config.xml
                web.xml
                   classes/
                         DataTable.class
                        DataTable.java
                  lib/
                         common-beanutils.jar
                         common-collections.jar
                         common-digester.jar
                        common-logging.jar
                        jsf-api.jar
                        jsf-impl.jar
                        jstl.jar
                        standard.jar
```

El ejemplo parte de la página *index.jsp* que se aprecia a continuación:

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/core"</pre>
                                                   prefix="f" %>
                                                   prefix="h" %>
   <%@ taglib uri="http://java.sun.com/jsf/html"</pre>
   <f:view>
      <head>
           <link href="estilos.css" rel="stylesheet" type="text/css"/>
         <title>
            <h:outputText value="Ejemplo data table"/>
         </title>
      </head>
      <body>
         <h:outputText value="Una simple tabla con nombres"/>
         <h:form>
            <h:dataTable border="2" value="#{dataTable.nombres}" var="nombre">
               <h:column>
                  <h:outputText value="#{nombre}"/>
                  <f:verbatim>,</f:verbatim>
               </h:column>
              <h:column>
                  <h:outputText value="#{nombre}"styleClass="enfasis"/>
               </h:column>
               <h:column>
                  <h:outputText value="#{nombre}"styleClass="color1"/>
               </h:column>
            </h:dataTable>
            >
            <\n.outputText value="Una simple tabla con componentes JSF"/>
            <h:dataTable border="3" value="#{listaNumeros}" var="numero">
```

<h:column>

```
<f:facet name="header">
                     <h:outputText value="Numeros"/>
                  </f:facet>
                  <h:outputText value="#{numero}"/>
               </h:column>
               <h:column>
                  <f:facet name="header">
                     <h:outputText value="Campo Texo"/>
                  </f:facet>
                  <h:inputText value="#{numero}" size="3"/>
               </h:column>
               <h:column>
                  <f:facet name="header">
                     <h:outputText value="Boton"/>
                  </f:facet>
                  <h:commandButton value="#{numero}"/>
               </h:column>
               <h:column>
                  <f:facet name="header">
                     <h:outputText value="Checkbox"/>
                  </f:facet>
                  <h:selectBooleanCheckbox value="false"/>
               </h:column>
               <h:column>
                  <f:facet name="header">
                     <h:outputText value="Enlaces"/>
                  </f:facet>
                  <h:commandLink>
                     <h:outputText value="#{numero}"/>
                  </h:commandLink>
               </h:column>
               <h:column>
                  <f:facet name="header">
                     <h:outputText value="radio boton"/>
                  </f:facet>
                  </h:selectOneRadio>
               </h:column>
               <h:column>
                  <f:facet name="header">
                     <h:outputText value="listbox"/>
                  </f:facet>
                  <h:selectOneListbox size="3">
                     <f:selectItem itemValue="siempre" itemLabel="siempre"/>
<f:selectItem itemValue="amenudo" itemLabel="a menudo"/>
                     <f:selectItem itemValue="algunavez" itemLabel="alguna vez" />
                     <f:selectItem itemValue="nunca" itemLabel="nunca" />
                  </h:selectOneListbox>
               </h:column>
            </h:dataTable>
         </h:form>
      </body>
  </f:view>
</html>
```

Al principio de *index.jsp*, se ha importado la hoja de estilos que se usará para darle un formato al texto de la aplicación de ejemplo. Esta hoja de estilos está recogida en el fichero *estilos.css*, cuyo contenido es el siguiente:

```
.enfasis {
   font-style: italic;
   font-size: 1.3em;
}
.color1 {
   font-size: 1.5em;
   font-style: italic;
   color: Blue;
   background: Yellow;
}
```

Se aprecian dos tipos de letra, un tipo de letra itálica con un tamaño de 1.3em., el segundo es de color amarillo para el fondo, itálica y color azul para la letra.

La primera tabla utilizada en el ejemplo, muestra tres columnas con tres nombres cada una, y con estilos de letra diferentes. Estos nombres, están recogidos en el bean *dataTable* (de la clase DataTable), donde se ha establecido un array de *String*, con los nombres que se mostrarán por pantalla. El bean se encuentra definido en el fichero *faces-config.xml*

La clase DataTable sería la siguiente:

```
public class DataTable {
    private static final String[] nombres = new String[] {
        new String("Jose"),
        new String("Manuel"),
        new String("Andres"),
        new String("Carlos"),
    };

    public String[] getNombres() { return nombres;}
}
```

Posee el array inicializado con los valores de los nombres, y con un método *get*, que obtiene los valores.

Este ejemplo no posee navegación de páginas alguna, así que el fichero *faces-config.xml*, solo contiene la declaración de los dos bean que posee el ejemplo.

El segundo bean que aparece en *faces-config.xml* (listaNumeros), es el utilizado en la segunda tabla del ejemplo, a diferencia del anterior, este bean, inicializa sus valores mediante una lista, haciendo uso de la directiva *list-entries*, con los valores:1,2,3,4,5. y que pertenece a la clase **java.util.ArrayList**. Ambos bean poseen un ámbito de sesión.

Capitulo 6 Conversión y Validación

6 Conversión y validación

En este capítulo se describe como los datos de un formulario son convertidos a objetos java y como los resultados de la conversión son chequeados para su validación. Se abarcarán conceptos previos a la conversión y a la validación, y se comentarán las etiquetas estándares que JSF provee para la conversión y la validación.

6.1 El proceso de conversión y validación

Veamos paso a paso como un usuario introduce un valor, y este viaja desde el formulario del navegador hasta el bean.

Primero, un usuario rellena un campo de un formulario web. Cuando el usuario pulsa el botón para tramitar el formulario, el navegador envía el dato del formulario al servidor, este dato se denomina, valor demandado.

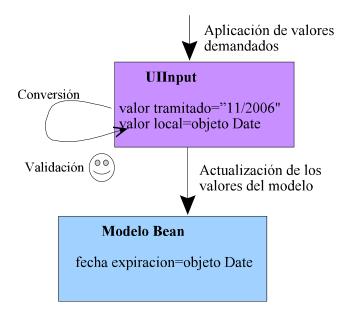
En la fase, "aplicación de valores demandados", los valores demandados son cargados en objetos (recuerde que cada etiqueta de entrada de la página JSF tiene su correspondiente objeto componente). El valor que es cargado en el objeto, se denomina valor tramitado.

Por supuesto, todos los valores demandados son cadenas de caracteres (*Strings*). Por otra parte, la aplicación web se ocupa de tipos arbitrarios, como enteros (*int*), fechas (*Date*) o tipos mas sofisticados. Un proceso de conversión transforma estas cadenas de caracteres en estos tipo. En la próxima sección se comenta la conversión en detalle.

Los valores convertidos no son tramitados inmediatamente al bean, sino que son primeramente cargados en objetos como valores locales. Después de la conversión, los valores locales son validados. Los diseñadores de páginas pueden establecer condiciones de validación, por ejemplo, que ciertos campos deban tener una longitud máxima o una longitud mínima. Después de que todos los valores locales hayan sido validados, empieza la fase "Actualización de los valores del modelo", y los valores locales son cargados en el bean.

JSF efectúa dos pasos para dar facilidades para conservar la integridad del modelo. Como todos los programadores conocen bien, los usuarios introducirán información equivocada con cierta regularidad. Suponga que algunos de los valores del modelo se han actualizado antes de que el primer error del usuario fuese detectado. El modelo podría entrar en un estado inconsistente, y sería tedioso regresarlo al estado anterior. Por esta razón, JSF primero convierte y valida todos lo valores de entrada del usuario. Si se encuentran errores, la página es recargada, a fin de que el usuario lo intente de nuevo. La fase "actualización de los valores del modelo" comienza solo si todas las validaciones ocurrieron sin problemas.

La siguiente figura muestra el recorrido de un valor introducido en el navegador, hacia un objeto en el lado del servidor y finalmente hasta el modelo bean.



6.2 Usando conversores estándares

En las siguientes secciones, se habla sobre los conversores y validadores que son parte de la librería de JSF.

6.2.1 Conversión de números y fechas

Una aplicación web carga datos de muchos tipos, pero el interfaz de usuario, trata exclusivamente con cadenas de caracteres. Por ejemplo, suponga que un usuario necesita editar un objeto fecha (*Date*). Primero, el objeto fecha es convertido a cadena de caracteres (*string*), que es enviado al navegador del cliente para ser mostrado dentro del campo de texto. El usuario luego edita el campo de texto. La cadena de caracteres resultante es retornada al servidor y debe ser convertida de nuevo a un objeto fecha. Lo mismo ocurre para tipos primitivos con *int*, *double* o *boolean*. El usuario de la aplicación web, edita cadenas de caracteres y el contenedor JSF necesita convertir esas cadenas al tipo requerido por la aplicación.

Para ver un uso típico de un conversor incorporado, imagine una aplicación web que se usa para tramitar pagos. Los datos de pago son:

- cantidad a ser cobrada
- número de la tarjeta de crédito
- fecha de caducidad de la tarjeta de crédito

Quedando una pantalla principal como la siguiente:

Introduzca inf	formacion del pago
Cantidad	0,00
Tarjeta de credito	
Fecha de expiracion (mes/año)	05/2006
Acepta	

Se usará un convesor para el campo de texto de la cantidad, indicándole que formatee el valor actual con al menos dos dígitos después de la coma:

El conversor **f:converNumber** es uno de los conversores estándares proporcionados por la implementación JSF.

El segundo campo del ejemplo no usa conversor. El tercer campo usa el conversor **f:datetime**, quien establece un patrón para la cadena de caracteres mm/yyyy:

En la página que muestra los resultados que se introdujeron, se utiliza un conversor diferente para en campo de la cantidad de pago:

Este conversor proporciona automáticamente un símbolo de moneda y un separador decimal.

Obteniendo el siguiente resultado por pantalla:

Informacion del pago

Cantidad 365,00 €

Tarjeta de credito 12324538123

Decha de expiracion 05/2006

Volver

6.2.2 Conversores y atributos

La siguiente table recoge los atributos para la etiqueta f:convetNumber

Atributo	Tipo	Valoridación
type	String	number (por defecto), currency , o percent
pattern	String	Formato de patrón
maxFractionDigits	int	Máximo número de dígitos en la parte fraccional
minFractionDigits	int	Mínimo número de dígitos en la parte fraccional
maxIntegerDigits	int	Máximo número de dígitos en la parte entera
minIntegerDigits	int	Mínimo número de dígitos en la parte entera
integerOnly	boolean	Vale true si es solo un entero (por defecto, false)
groupingUsed	boolean	Vale true si se usa un grupo de separadores (por defecto, false)
currencyCode	String	ISO 4217 - Código de moneda usado para la conversión a valores monetarios
currencySymbol	String	Símbolo de moneda usado para la conversión a valores monetarios

La siguiente table recoge los atributos para la etiqueta f:convetDateTime

Atributo	Tipo	Valoridación
type	String	date (por defecto), time, o ambos
dateStyle	String	default, short, medium, long, o full
timeStyle	String	default, short, medium, long, o full
pattern	String	Formato de patrón
timeZone	java.util.TimeZone	Zona horaria a usar

6.2.3 Mensajes de error

Es importante que el usuario pueda ver los mensajes de error que son causados por los errores de conversión y validación. Debería añadir etiquetas **h:message** cuando use conversores y validadores.

Lo lógico es mostrar los mensajes de error junto a los componentes que causan dichos errores. Por los tanto, asígnele un valor identificativo a un componente (con el atributo **id**) y referéncielo con la etiqueta **h:message**, por ejemplo:

```
<h:inputText id="cantidad" value="#{pago.cantidad}"> <h:message for="cantidad"/>
```

Un mensaje posee dos partes, sumario y detalle. Por defecto, la etiqueta **h:message** muestra el detalle y esconce el sumario. Si quiere mostrar el sumario, use los siguientes atributos:

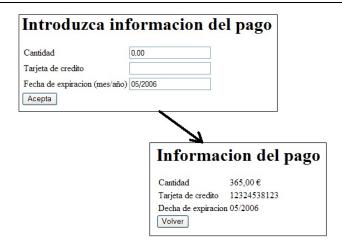
```
<h:message for="cantidad" showSummary="true" showDetail="false"/>
```

También puede usar los atributos *styleClass* o *style* para cambiar la apariencia del mensaje:

```
<h:messages styleClass="mensajeError"/>
o
<h:message for="cantidad" style="color:red"/>
```

6.2.4 Cambiar el texto de los mensajes de error estándares

Los mensajes de error por defecto para la conversión son "Conversion error ocurred", el cual lo puede modificar para que sea mas claro o familiar. Para ello basta con modificar el fichero que recoge los mensajes de la aplicación, es decir, el fichero con extensión .properties, y se reemplaza el la clave javax.faces.component.UIInput.CONVERSION, por el texto que desee, por ejemplo:



javax.faces.component.UIInput.CONVERSION=Por favor, introduzca el valor correctamente

y por último, establezca el nombre de la base que recoge los mensajes en el fichero de configuracion (faces-config.xml), algo como:

en el caso de que el fichero que contenga los mensajes se llame, mensajes.properties

6.2.5 Un ejemplo completo usando conversores

Ya está listo para abordar el siguiente ejemplo, donde se presenta un pequeño formulario que le pide al usuario unos datos sobre un pago, como la cantidad a pagar, el número de la tarjeta de crédito y la fecha, tras aceptar la información por pantalla, esta es mostrada por pantalla. A lo largo del capítulo se han ido mostrando fragmentos de código que mostraban el uso de la conversión, estos fragmentos pertenecen a la aplicación de este ejemplo, por lo que visualizar su código al completo debería serle muy fácil de entender.

Como estructura de directorios del ejemplo, se tiene lo siguiente:

```
dataTable
index.html
index.jsp
resultado.jsp
```

```
estilos.css
  WEB-INF/
        faces-config.xml
        web.xml
           lasses/
                 BeanPago.class
            src/
                 BeanPago.java
           lib/
                 common-beanutils.jar
                 common-collections.jar
                 common-digester.jar
                 common-logging.jar
                 jsf-api.jar
                 jsf-impl.jar
                 jstl.jar
                 standard.jar
```

La página principal, index.jsp se muestra a continuación:

```
<html>
  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
  <f:view>
      <head>
         <link href="estilos.css" rel="stylesheet" type="text/css"/>
         <title><h:outputText value="Conversion"/></title>
     </head>
     <body>
         <h:form>
            <h1><h:outputText value="Introduzca informacion del pago"/></h1>
            <h:panelGrid columns="3">
               <h:outputText value="Cantidad"/>
               <h:inputText id="cantidad" value="#{pago.cantidad}">
                  <f:convertNumber minFractionDigits="2"/>
               </h:inputText>
               <h:message for="cantidad" styleClass="mensajeError"/>
               <h:outputText value="Tarjeta de credito"/>
               <h:inputText id="tarjeta" value="#{pago.tarjeta}"/>
               <h:panelGroup/>
               <h:outputText value="Fecha de expiracion (mes/año)"/>
               <h:inputText id="fecha" value="#{pago.fecha}">
                  <f:convertDateTime pattern="MM/yyyy"/>
               </h:inputText>
               <h:message for="fecha" styleClass="mensajeError"/>
            </h:panelGrid>
            <h:commandButton value="Acepta" action="procesar"/>
         </h:form>
     </body>
  </f:view>
</html>
```

La siguiente página de la navegación, tras aceptar el formulario es resultado.jsp:

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

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <f:view>
     <head>
         <link href="estilos.css" rel="stylesheet" type="text/css"/>
         <title><h:outputText value="Conversion"/></title>
     </head>
     <body>
         <h:form>
            <h1><h:outputText value="Informacion del pago"/></h1>
            <h:panelGrid columns="2">
               <h:outputText value="Cantidad"/>
               <h:outputText value="#{pago.cantidad}">
                  <f:convertNumber type="currency"/>
               </h:outputText>
               <h:outputText value="Tarjeta de credito"/>
               <h:outputText value="#{pago.tarjeta}"/>
               <h:outputText value="Decha de expiracion"/>
               <h:outputText value="#{pago.fecha}">
                  <f:convertDateTime pattern="MM/yyyy"/>
               </h:outputText>
            </h:panelGrid>
            <h:commandButton value="Volver" action="volver"/>
         </h:form>
     </body>
  </f:view>
</html>
```

y por último el contenido del fichero BeanPago.java:

```
import java.util.Date;

public class BeanPago {
    private double cantidad;
    private String tarjeta = "";
    private Date fecha = new Date();

    // ATRIBUTO: cantidad
    public void setCantidad(double nuevoValor) { cantidad = nuevoValor; }
    public double getCantidad() { return cantidad; }

    // ATRIBUTO: tarjeta
    public void setTarjeta(String nuevoValor) { tarjeta = nuevoValor; }
    public String getTarjeta() { return tarjeta; }

    // ATRIBUTO: fecha
    public void setFecha(Date nuevoValor) { fecha = nuevoValor; }
    public Date getFecha() { return fecha; }
}
```

6.3 Usando validadores estándares

Es difícil imaginar una aplicación Web que no realice una buena de validación de datos. La validación debería ser fácil de usar y extender. JavaServer Faces provee una buena cantidad de validadores estándares y ofreciéndole un mecanismo sencillo para implementar sus validadores.

6.3.1 Validando longitudes de cadenas de caracteres y rangos numéricos

Es muy fácil usar validadores JSF dentro de páginas JSF. Simplemente añada etiquetas validadoras al cuerpo de un componente etiqueta, algo tal que así:

Este fragmento de código, añade un validador al campo de texto de entrada; cuando el texto del campo de entrada es aceptado, el validador se asegura de que el texto introducido tiene como máximo 12 caracteres. Cuando el validador falla (en este caso, cuando el número de caracteres es mayor de 12), genera un mensajes de error asociado con el componente responsable. Estos mensajes de error pueden ser mostrados en una página JSF por las etiquetas h:message o h:messages.

JavaServer Faces se basa en mecanismos que le permiten llevar a cabo las siguientes validaciones:

- Chequear la longitud de una cadena de caracteres
- Chequear límites para un valor numérico (por ejemplo, >0 o <=100)
- Chequear que un valor ha sido proporcionado

En el fragmento de código anterior, se vio el uso de la validación de longitudes de cadenas de caracteres; para validar entradas numéricas se unas validadores de rangos. Por ejemplo:

```
<h:inputText value="#{regalo.cantidad}">
 <f:validateLongRange minimum="10" maximum="10000"/>
</h:inputText>
```

El validador chequea que el valor proporcionado es ≥ 10 y ≤ 10000 .

La siguiente tabla muestra los validadores estándares que JSF provee:

etiqueta JSP	Clase de validador	Atributos	Validación
f:validateDoubleRange	DoubleRangeValidator	minimum, maximum	valida un valor double dentro de un rango
f:validateLongRange	LongRangeValidator	minimum, maximum	valida un valor long dentro de un rango

etiqueta JSP	Clase de validador	Atributos	Validación
f:validateLength	LengthValidator	minimum, maximum	valida un String dentro de un número mínimo y un número máximo de caracteres

Todas las etiquetas de validadores estándares poseen los atributos **minimum** y **maximum**. Debe aportar al menos uno de estos atributos.

6.3.2 Chequeando valores requeridos

Para chequear si un valor es aportado, no necesita anidar un validador dentro de la etiqueta de entrada. En su lugar, utilice el atributo, required="true":

```
<h:inputText value="#{informe.fecha}" required="true"/>
```

Todas las etiquetas JSF de entrada, soportan el atributo required. Puede combinar un atributo required con un validador anidado:

6.3.3 Mostrando errores de validación

Los errores de validación se manejan de igual manera que los errores de conversión. Un mensaje se añade al componente que falla la validación, el componente es invalidado, y la página actual es recargada inmediatamente tras completarse la fase "proceso de validaciones". Use las etiquetas h:message y h:messages para mostrar los errores de validación.

6.3.4 Un completo ejemplo de validación

Para completar el ejemplo de conversión visto anteriormente, se le ha añadido elementos de validación, de manera que la cantidad introducida esté comprendida dentro de un rango de valores, que el numero de la tarjeta de crédito tenga un valor mínimo y que la fecha sea requerida, por ejemplo:

Introduzca informacion del pago				
Cantidad	0,00	Error de Validación: PENDING(localize): 10 - 10.000.		
Tarjeta de credito	12	Error de Validación: Valor is menos de valor de minimo permitido: '13'.		
Fecha de expiracion (me	s/año)	Error de Validación: Valor es necesario.		
Aceptar Cancelar				

Como estructura de directorios del ejemplo, se tiene lo siguiente:

```
dataTable
        index.html
        index.jsp
        resultado.jsp
        cancelado.jsp
        estilos.css
        WEB-INF/
                faces-config.xml
                web.xml
                    lasses/
                         BeanPago.class
                   src/
                        BeanPago.java
                   lib/
                         common-beanutils.jar
                        common-collections.jar
                         common-digester.jar
                        common-logging.jar
                        jsf-api.jar
                        jsf-impl.jar
                        jstl.jar
                         standard.jar
```

Con lo que solo cambiaria una par de aspectos con respecto al ejemplo, anterior, lo principal es la página index.jsp, donde se recogen tanto los conversores como los validadores:

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <f:view>
          <link href="estilos.css" rel="stylesheet" type="text/css"/>
<title><h:outputText value="Ejemplo Validacion"/></title>
      </head>
      <body>
          <h:form>
             <h1><h:outputText value="Introduzca informacion del pago"/></h1>
             <h:panelGrid columns="3">
                 <h:outputText value="Cantidad"/>
                <h:inputText id="cantidad" value="#{pago.cantidad}"
                    required="true">
                    <f:convertNumber minFractionDigits="2"/>
                    <f:validateDoubleRange minimum="10" maximum="10000"/>
                 </h:inputText>
                 <h:message for="cantidad" styleClass="mensajeError"/>
                <h:outputText value="Tarjeta de credito"/>
                <h:inputText id="tarjeta" value="#{pago.tarjeta}"
    required="true">
```

Tutorial de JavaServer Faces

y como extra, se haya un segundo botón junto al de aceptar formulario, para cancelar introducción de datos y navegar a una página (cancelado.jsp) donde se indica que la operación de pago se ha cancelado.

```
<html>
  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
  <f:view>
     <head>
         <link href="estilos.css" rel="stylesheet" type="text/css"/>
         <title><h:outputText value="Validacion"/></title>
     </head>
     <body>
         <h:form>
            <h:outputText value="Cancelado"/>
           <h:commandButton value="Volver" action="volver"/>
        </h:form>
     </f:view>
  </body>
</html>
```

Dejándonos la opción de regresar a la pantalla principal.

Capitulo 7 Manejo de Eventos

7 Manejo de eventos

Las aplicaciones de Web a menudo necesitan responder a los eventos del usuario, como los elementos seleccionados de un menú o dando un clic sobre un botón.

Típicamente, se registran manejadores de eventos con componentes; por ejemplo, podría registrar un oyente de cambio de valor(*valueChangeListener*) con una etiqueta menú en una página JSF de la siguiente manera:

<h:selectOneMenu valueChangeListener="#{formulario.ciudad}"...>
...
</h:selectOneMenu>

En el precedente código, el método directo #{formulario.ciudad} referencia al método ciudad de un bean denominado formulario. Este método es invocado por la implementación de JSF después de realizar la selección del menu. Justo cuando es invocado el método, es una de las claves a tratar en este capítulo. JSF soportar tres clases de eventos

- Eventos de cambio de valor
- Eventos de acción
- Eventos de fase

Los eventos de cambio de valor son disparados con los componentes de entrada, como *h:inputText, h:selectOneRadio*, y *h:selectManyMenu*, cuando el valor del componente cambia y el formulario es tramitado.

Los eventos de acción son disparados por componentes de comando, como *h:commandButton* y *h:commandLink*, cuando el botón o el enlace es activado.

Los eventos de fase son disparados por el ciclo de vida de JSF. Si quiere manejar eventos, necesita tener un conocimiento básico sobre el ciclo de vida, como se vio en el capítulo 3.

7.1 Eventos de cambio de valor

Los componentes de una aplicación web, a menudo dependen de otros componentes.

Por ejemplo, suponga que tiene que rellenar un formulario en el que debe introducir una cantidad económica, que dependiendo de su nacionalidad, ésta sera en euros, dólares, etc... Tal y como se muestra en el siguiente ejemplo:

Por favor, rellene el formul	ario
Nombre	
Direccion	
Telefono	
Euros a donar	€
Nacionalidad Espana	Por favor, rellene el formulario
aceptar	Nombre
	Direccion
	Telefono
	Dolares a donar \$
	Nacionalidad Estados Unidos 🔻
	aceptar

En la etiqueta de entrada que recoge la cantidad económica a donar, varia según indicemos en el menú de la nacionalidad, estableciendose como euros a donar, si somos españoles o como dólares a donar en el caso de estadounideneses. En esta pequeña aplicación, el menú que contiene las nacionalidades establece un oyente de cambio de valor(valueChangeListener), como se muestra en el siguiente trozo de código:

y que usa el atributo onchange para forzar la tramitación del formulario tras algún cambio en el menú.

Cuando una nacionalidad es seleccionada en el menú, la función JavaScript submit() es invocada para tramitar el formulario, el cual invoca seguidamente el ciclo de vida de JSF. Después de la fase del proceso de validación, la implementación de JSF invoca el método cambioNacionalidad del bean llamando formulario. Este método cambia la vista de la apliación, según el nuevo valor de la nacionalidad:

```
private static final String US = "Estados Unidos";
...
public void cambioNacionalidad(ValueChangeEvent evento) {
    FacesContext contexto = FacesContext.getCurrentInstance();

    if(US.equals((String) evento.getNewValue()))
        contexto.getViewRoot().setLocale(Locale.US);
    else
        contexto.getViewRoot().setLocale(Locale.getDefault());
}
```

El oyente es pasado a un evento de cambio de valor. El oyente utiliza este evento para acceder al nuevo valor del componente. La clase *ValueChangeEvent* extiende de *FacesEvent*, ambos del residen en el paquete *javax.faces.event*. Los métodos más comúnmente usados de estas clases se listan a continuación:

✓ javax.faces.event.ValueChangeEvent

- UIComponent getComponent()
 Devuelve el componente de entrada que disparo el evento.
- Object getNewValue()
 Devuelve el nuevo valor del componente, después de que el valor haya sido convertido y validado.
- Object getOldValue()
 Devuelve el valor previo del componente.

✓ javax.faces.event.FacesEvent

- void queue()
 - Cola de eventos a soltar al acabar la fase actual del ciclo de vida.
- PhaseId getPhaseId()
 - Devuelve el identificador de fase correspondiente a la fase durante el evento es soltado
- void setPhaseId(PhaseId)
 Conjunto de identificadores de fase correspondientes a la fase durante los eventos son soltados.

La estructura de directorios para la aplicación se muestra a continuación, así como el contenido de los ficheros de la misma:

```
cambioValor
index.html
index.jsp
gracias.jsp
estilos.css
WEB-INF/
faces-config.xml
```

```
web.xml
  classes/
        Formulario.class
        mensajes es.properties
        mensajes en.properties
   src/
        Formulario.java
   lib/
        common-beanutils.jar
        common-collections.jar
        common-digester.jar
        common-logging.jar
        jsf-api.jar
        jsf-impl.jar
        jstl.jar
        standard.jar
```

/cambioValor/Index.jsp:

```
<html>
   <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <f:view>
      <head>
          k href="estilos.css" rel="stylesheet" type="text/css"/>
<f:loadBundle basename="mensajes" var="msjs"/>
              <h:outputText value="#{msjs.titulo}"/>
          </title>
      </head>
      <body>
          <h:outputText value="#{msjs.cabecera}" styleClass="enfasis"/>
          <h:form>
             <h:panelGrid cellspacing="5" columns="2">
                 <h:outputText value="#{msjs.nombre}"/>
<h:inputText id="nombre" value="#{formulario.nombre}"/>
                 <h:outputText id="direccion" value="#{msjs.direccion}"/>
                 <h:inputText value="#{formulario.direccion}"/>
                 <h:outputText id="telefono" value="#{msjs.telefono}"/>
                 <h:inputText value="#{formulario.telefono}"/>
                 <h:outputText id="cantidad" value="#{msjs.cantidad}"/>
                 <h:panelGroup>
                          <h:inputText value="#{formulario.cantidad}"/>
                          <h:outputText id="signo" value="#{msjs.signo}"/>
                 </h:panelGroup>
                 <h:outputText value="#{msjs.nacionalidad}"/>
                 <h:selectOneMenu value="#{formulario.nacionalidad}"
                    onchange="submit()"
                    valueChangeListener="#{formulario.cambioNacionalidad}">
                    <f:selectItems value="#{formulario.nacionalidades}"/>
```

```
</h:selectOneMenu>
            </h:panelGrid>
            <h:commandButton action="aceptar" value="#{msjs.aceptar}"/>
         </h:form>
     </body>
   </f:view>
</html>
/cambioValor/gracias.jsp:
  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
  <f:view>
      <head>
        <link href="estilos.css" rel="stylesheet" type="text/css"/>
         <f:loadBundle basename="mensajes" var="msjs"/>
           <h:outputText value="#{msjs.titulo}"/>
         </title>
     </head>
      <body>
         <h:form>
               <h:outputText value="#{msjs.gracias}"/>
            </h3>
         </h:form>
     </body>
  </f:view>
</html>
```

/cambioValor/WEB-INF/src/Formulario.java:

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Locale;
import javax.faces.context.FacesContext;
import javax.faces.event.ValueChangeEvent;
import javax.faces.model.SelectItem;
public class Formulario {
  private String nombre;
  private String direccion;
  private String telefono;
  private String nacionalidad;
   private String cantidad;
  private static final String ES = "Espana";
  private static final String US = "Estados Unidos";
  private static final String[] NOMBRES_NAC = { ES, US };
  private static ArrayList nacItems = null;
   // ATRIBUTO: nacionalidades
   public Collection getNacionalidades() {
      if(nacItems == null) {
         nacItems = new ArrayList();
         for (int i = 0; i < NOMBRES NAC.length; ++i) {</pre>
            nacItems.add(new SelectItem(NOMBRES NAC[i]));
```

```
return nacItems;
         }
          // ATRIBUTO: nombre
         public void setNombre(String nuevoValor) { nombre = nuevoValor; }
         public String getNombre() { return nombre; }
          // ATRIBUTO: direccion
         public void setDireccion(String nuevoValor) { direccion = nuevoValor; }
         public String getDireccion() { return direccion; }
         // ATRIBUTO: telefono
         public void setTelefono(String nuevoValor) { telefono = nuevoValor; }
         public String getTelefono() { return telefono; }
         // ATRIBUTO: nacionalidad
         public void setNacionalidad(String nuevoValor) { nacionalidad =
nuevoValor; }
         public String getNacionalidad() { return nacionalidad; }
        // ATRIBUTO: cantidad
         public void setCantidad(String nuevoValor) { cantidad = nuevoValor; }
         public String getCantidad() { return cantidad; }
         public void cambioNacionalidad(ValueChangeEvent evento) {
            FacesContext contexto = FacesContext.getCurrentInstance();
            if (US.equals((String) evento.getNewValue()))
               contexto.getViewRoot().setLocale(Locale.US);
               contexto.getViewRoot().setLocale(Locale.getDefault());
         }
      }
```

cambioValor/WEB-INF/faces-config.xml:

```
<faces-config>
<navigation-rule>
<navigation-case>
<from-outcome>aceptar</from-outcome>
<to-view-id>/gracias.jsp</to-view-id>
</navigation-case>
</navigation-rule>

<managed-bean>
<managed-bean-name>formulario</managed-bean-name>
<managed-bean-class>Formulario</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>
</managed-bean>
</faces-config>
```

cambioValor/WEB-INF/classes/mensajes es.properties:

titulo=Ejemplo de eventos de cambio de valor

cabecera=Por favor, rellene el formulario

nombre=Nombre
direccion=Direccion
telefono=Telefono
cantidad=Euros a donar
signo=€
aceptar=aceptar
nacionalidad=Nacionalidad
gracias=Gracias por su donacion

cambioValor/WEB-INF/classes/mensajes en.properties:

titulo=Ejemplo de eventos de cambio de valor cabecera=Por favor, rellene el formulario

nombre=Nombre
direccion=Direccion
telefono=Telefono
cantidad=Dolares a donar
signo=\$
aceptar=aceptar
nacionalidad=Nacionalidad
gracias=Gracias por su donacion

7.2 Eventos de acción

Los eventos de acción son disparados por componentes de comando (botones, enlaces, ...) cuando el componente es activado. Los eventos de acción son disparados durante la fase del ciclo de vida de invocación de la aplicación, cerca del fin del ciclo de vida.

Normalmente los oyentes de acción se asocian a componentes de comando en una página JSF. Por ejemplo, puede añadir un componente de acción a una etiqueta de enlace (*link*), tal que así:

<h:commandLink actionListener="#{bean.linkActivado}"> ... </h:commandLink>

Los componentes de comando se tramitan cuando son activados, por lo que no necesitan el atributo *onchange* para forzar la tramitación del formulario, como era el caso de los eventos de cambio de valor. Cuando activa un comando o un enlace, el formulario es tramitado y la implementación JSF dispara el evento de acción.

Es importante distinguir entre oyentes de acción y acciones. En resumidas cuentas, las acciones son diseñadas para la lógica comercial y participan en el manejo de la navegación,

mientras que los oyentes de acción típicamente realizan lógica de interfaz de usuario y no participan de manejo de navegación.

Los oyentes de acción a menudo trabajan en conjunto con acciones cuando una acción necesita información acerca de la interfaz de usuario. Por ejemplo, la siguiente aplicación usa una acción y un oyente de acción para reaccionar a los clicks del ratón para navegar a una página JSF. Si da un clic sobre un signo del zodíaco, entonces la aplicación navega a una página JSF con información acerca de ese signo. Note que una acción aisladamente no puede implementar ese comportamiento — una acción puede navegar a una página, pero no puede determinar la página apropiada porque no sabe nada del botón de imagen en la interfaz de usuario o el click del ratón.

La aplicación a modo de ejemplo se vería de la siguiente manera:



La estructura de directorios para la aplicación se muestra a continuación, así como el contenido de los ficheros de la misma:

```
zodiaco
index.html
index.jsp
zodiaco.jpg
aries.jsp
acuario.jsp
cancer.jsp
capricornio.jsp
```

```
escorpio.jsp
geminis.jsp
leo.jsp
libra.jsp
piscis.jsp
sagitario.jsp
tauro.jsp
virgo.jsp
estilos.css
 WEB-INF/
        faces-config.xml
        web.xml
         classes/
                 Zodiaco.class
                 mensajes.properties
                 Zodiaco.java
          lib/
                 common-beanutils.jar
                 common-collections.jar
                 common-digester.jar
                 common-logging.jar
                 jsf-api.jar
                 jsf-impl.jar
                 jstl.jar
                 standard.jar
```

zodiaco/index.jsp:

```
<html>
    <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
    <f:view>
        <head>
            k href="estilos.css" rel="stylesheet" type="text/css"/>
<f:loadBundle basename="mensajes" var="msjs"/>
            <title>
                 <h:outputText value="#{msjs.titulo}"/>
             </title>
        </head>
        <body>
            <h:form>
                 <h:commandButton image="zodiaco.jpg"
                     actionListener="#{zodiaco.oyente}"
action="#{zodiaco.accion}"/>
            </h:form>
        </body>
    </f:view>
</html>
```

zodiaco/aries.jsp:(El resto de ficheros de signos del zodiaco son similares)

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
   <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
   <f:view>
      <head>
         <link href="estilos.css" rel="stylesheet" type="text/css"/>
         <f:loadBundle basename="mensajes" var="msjs"/>
            <h:outputText value="#{msjs.titulo}"/>
         </title>
      </head>
      <body>
         <h:form>
            <h:outputText value="#{msjs.tituloAries}"
              styleClass="tituloSigno"/>
            <\a>
            <h:outputText value="#{msjs.aries}"
              styleClass="signo"/>
         </h:form>
      </body>
   </f:view>
</html>
```

zodiaco/WEB-INF/src/Zodiaco.java:

```
import java.awt.Point;
       import java.awt.Rectangle;
       import java.util.Map;
       import javax.faces.context.FacesContext;
       import javax.faces.event.ActionEvent;
      public class Zodiaco {
         private String resultado = null;
          private Rectangle geminis = new Rectangle(110, 32, 30, 30);
private Rectangle cancer = new Rectangle(166, 32, 30, 30);
         private Rectangle leo = new Rectangle(200, 57, 45, 30);
         private Rectangle virgo = new Rectangle(235, 106, 30, 30);
         private Rectangle libra = new Rectangle(242, 154, 30, 40);
         private Rectangle escorpio = new Rectangle(210, 218, 30, 30);
          private Rectangle sagitario = new Rectangle(155, 243, 40, 30);
          private Rectangle capricornio = new Rectangle(103, 239, 40, 30);
          private Rectangle acuario = new Rectangle(59, 215, 30, 30);
          private Rectangle piscis = new Rectangle(31, 161, 30, 40);
         private Rectangle aries = new Rectangle(37, 105, 30, 40);
         private Rectangle tauro = new Rectangle(61, 58, 40, 40);
          public void oyente(ActionEvent e) {
             FacesContext context = FacesContext.getCurrentInstance();
             String clientId = e.getComponent().getClientId(context);
             Map requestParams =
context.getExternalContext().getRequestParameterMap();
             int x = new Integer((String) requestParams.get(clientId + ".x"))
                    .intValue();
             int y = new Integer((String) requestParams.get(clientId + ".y"))
                    .intValue();
```

```
resultado = null;
   if (aries.contains(new Point(x, y)))
      resultado = "aries";
   if (tauro.contains(new Point(x, y)))
      resultado = "tauro";
   if (geminis.contains(new Point(x, y)))
      resultado = "geminis";
   if (cancer.contains(new Point(x, y)))
      resultado = "cancer";
   if (leo.contains(new Point(x, y)))
      resultado = "leo";
   if (virgo.contains(new Point(x, y)))
      resultado = "virgo";
   if (libra.contains(new Point(x, y)))
      resultado = "libra";
   if (escorpio.contains(new Point(x, y)))
      resultado = "escorpio";
   if (sagitario.contains(new Point(x, y)))
      resultado = "sagitario";
   if (capricornio.contains(new Point(x, y)))
      resultado = "capricornio";
   if (acuario.contains(new Point(x, y)))
      resultado = "acuario";
   if (piscis.contains(new Point(x, y)))
      resultado = "piscis";
}
public String accion() {
   return resultado;
```

cambioValor/WEB-INF/faces-config.xml:

```
<faces-config>
<navigation-rule>
<navigation-case>
<from-outcome>aries</from-outcome>
<to-view-id>/aries.jsp</to-view-id>
</navigation-case>
<navigation-case>
<from-outcome>tauro</from-outcome>
<to-view-id>/tauro.jsp</to-view-id>
</navigation-case>
</navigation-case>
```

<navigation-case>

```
<from-outcome>geminis</from-outcome>
            <to-view-id>/geminis.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
            <from-outcome>cancer</from-outcome>
            <to-view-id>/cancer.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
            <from-outcome>leo</from-outcome>
            <to-view-id>/leo.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
            <from-outcome>virgo</from-outcome>
            <to-view-id>/virgo.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
            <from-outcome>libra</from-outcome>
            <to-view-id>/libra.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
            <from-outcome>escorpio</from-outcome>
            <to-view-id>/escorpio.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
            <from-outcome>sagitario/from-outcome>
            <to-view-id>/sagitario.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
            <from-outcome>capricorpnio</from-outcome>
            <to-view-id>/capricornio.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
            <from-outcome>acuario</from-outcome>
            <to-view-id>/acuario.jsp</to-view-id>
      </navigation-case>
      <navigation-case>
            <from-outcome>piscis</from-outcome>
            <to-view-id>/piscis.jsp</to-view-id>
      </navigation-case>
 </navigation-rule>
 <managed-bean>
      <managed-bean-name>zodiaco</managed-bean-name>
      <managed-bean-class>Zodiaco</managed-bean-class>
      <managed-bean-scope>session</managed-bean-scope>
 </managed-bean>
</faces-config>
```

cambioValor/WEB-INF/classes/mensajes.properties:

titulo=Signos de Zodiaco

tituloAries=Signo de Aries
tituloTauro=Signo de Tauro
tituloGeminis=Signo de Geminis
tituloCancer=Signo de Cancer
tituloLeo=Signo de Leo
tituloVirgo=Signo de Virgo
tituloLibra=Signo de Libra
tituloEscorpio=Signo de Escorpio
tituloSagitario=Signo de Sagitario
tituloCapricornio=Signo de Capricornio
tituloAcuario=Signo de Acuario
tituloPiscis=Signo de Piscis

aries=21 marzo - 20 abril. Aventureros y energéticos, los aries son pioneros y valientes. Son listos, dinámicos, seguros de si y suelen demostrar entusiasmo hacia las cosas.Pueden ser egoístas y tener genio. Los Aries son impulsivos y a veces tienen poca paciencia. Tienden a tomar demasiados riesgos.

7.3 Eventos de fase

La implementación JSF dispara eventos conocidos como eventos de fase, antes y después de cada fase del ciclo de vida. Estos eventos son manejados por oyentes de fase. Los oyentes de fase son especificados en archivo de configuración faces-config.xml, tal que así:

En el precedente fragmento de código se especifica un solo oyente, pero puede especificar tantos como necesite. Los oyentes son invocados en el orden en el que son especificados en el fichero de configuración. Usted implementa oyentes de fase por medio de la interfaz *PhaseListener* del paquete *javax.faces.event*. Esa interfaz define tres métodos:

- PhaseId getPhaseId()
- void afterPhase(PhaseEvent)
- void beforePhase(PhaseEvent)

El método *getPhaseId* dice a la implementación JSF cuando debe entregar los eventos de fase al oyente; por ejemplo, *getPhaseId* () podría devolver PhaseId.APPLY REQUEST VALUES. En ese caso, *beforePhase* () y *afterPhase* () serían

Tutorial de JavaServer Faces

llamados una vez por ciclo de vida: antes y después de la fase de aplicación de valores de petición. También podría especificar PhaseId.ANY_PHASE, lo cual realmente significa todas las fases — sus métodos de oyentes de fase beforePhase y afterPhase serán llamados que seis veces por ciclo de vida: Una vez por cada fase del ciclo de vida. Los oyentes de fase son útil para la depuración y corrección de errores.

Por ejemplo, si usa componentes JSF en otra aplicación en otro marco de trabajo como Struts, entonces podría querer actualizar la localidad de ese marco de trabajo después de la fase de aplicación de valores de petición, cuándo JSF internamente establece su vista local.

Capitulo 8 Subvistas

8 Subvistas

El interfaz de usuario es típicamente el aspecto mas volátil de una aplicación web durante su desarrollo, por lo tanto es crucial crear un interfaz flexible y extensible. Este capítulo muestra como conseguir esa flexibilidad mediante subvistas.

8.1 Organización común

Muchos sitios web coinciden en la organización de los elementos de sus páginas, distribuyendo cada página en una cabecera, un contenido y un menú, como muestra la siguiente figura:

Cabecera	
M E n u	Contenido

Puede usar frames HTML para lograr un aspecto como el de la figura anterior, pero no es una opción recomendable. En los próximos apartados se comenta la alternativa a dichos frames HTML.

8.2 Un visor de libro

Para mostrar las organizaciones de las páginas web, se va a realizar una aplicación a modo de ejemplo práctico, un visor de libro. La idea es estructurar la aplicación, de manera que quede todo bien organizado, como muestra la siguiente figura:



Esta estructura muestra una cabecera con una imagen, un marco izquierdo con los enlaces a los diferentes capítulos y un marco derecho con el contenido de cada capítulo que sea seleccionado.

El visor de libro es muy intuitivo, si se pulsa el enlace correspondiente a un capítulo, el contenido de este, se mostrará en la región de contenido de la pagina.

El visor de libro está implementado con un **panelGrid** con dos columnas. La región cabecera contiene una imagen, texto y una línea horizontal HTML. Además de la cabecera, se presenta una fila que contiene el menú y el contenido del capítulo actual. El menú se muestra en la columna izquierda de la pantalla y el capítulo actual en la columna derecha.

El menu se compone de enlaces a los diferentes capítulos.

Una página monolítica JSF es una elección pobre para el visor de libro porque la página JSF es difícil de modificar. También, dese cuenta de que nuestra página monolítica JSF representa dos cosas: la organización y el contenido.

La organización es implementada con una etiqueta **h:panelGrid** y el contenido es representado por varias etiquetas JSF, como *h:graphicImage*, *h:outputText*, *h:commandLink*, y los capítulos del libro. Si realiza esto con una página monolítica JSF, no podrá reutilizar el contenido o la organización.

En lugar de apretujar un montón de código dentro de una página monolítica JSF, es mejor incluir contenido común así puede reusar que el contenido en otras páginas JSF. Con JSP, tiene tres opciones para incluir el contenido:

```
<%@ include file="cabecera.jsp"% >
<jsp:include page="cabecera.jsp"/>
<c:import url="cabecera.jsp"/>
```

La primera elección, la directiva *include file*, incluye el archivo especificado antes de que la página JSF sea compilada para el *servlet*.. Sin embargo, padece de una limitación importante: Si el archivo incluido contiene cambios después de que la página que incluye fuese procesada, esos cambios no están reflejados en la página que incluye. Eso significa que usted debe actualizar manualmente las páginas.

Las dos últimas elecciones listadas incluyen el contenido de una página en tiempo de ejecución y mezcla el contenido incluido con la página incluida JSF. Como la inclusión ocurre en tiempo de ejecución, los cambios para páginas incluidas se reflejan siempre cuando la página que incluye es recargada. Por esa razón, *jsp:Include* y *c:import* es preferido normalmente a la directiva *include*.

8.2.1 Consideraciones para la especificación JSF

Tanto si hace uso de la directiva include, *jsp:include* o *c:import*, hay que tener en cuenta dos consideraciones cuando incluya contenido en una aplicación JSF:

- Debe incluir las etiquetas JSF dentro de una etiqueta **f:subview**.
- Las etiquetas JSF includas no pueden contener la etiqueta **f:view**

Por ejemplo, el visor de libro debería encapsular el encabezado en su propia página JSF a fin de que pueda reusar ese contenido:

```
<%-- cabecera .jsp --%>
  <%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
  <%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
  <h:panelGrid columns="1" styleClass="cabecera">
  <h:panelGrid columns="1" styleClass="cabecera">
  <h:graphicImage value="imagen.jpg"/>
  <h:outputText value="#{msjs.tituloLibro}"

styleClass="tituloLibro"/>
  ...
  </h:panelGrid>

Ahora puede incluir este contenido el la página original JSF:
```

```
<%-- Esta es la página JSF original --%>
<f:view>
...
<f:subview id="cabecera">
<c:import url="cabecera.jsp"/>
</f:subview>
...
</f:view>
```

Debe asignar una identificador ID a cada subvista.

8.2.2 Inclusión de contenido en el visor de libro.

Para incluir el contenido en el visor de libro, en lugar de una página monolítica JSF, se dispone de tres ficheros: /cabecera.jsp, /menu.jsp y contenido.jsp. Se incluirá la cabecera, el menú y el contenido en la página JSF original.

De esta manera, el código es mas limpio, fácil de entender y de mantener. Y lo que es mas importante, puede reusar la cabecera, menú y contenido en otras vistas.

Ejemplo: visor de libro

Como estructura de directorios del ejemplo, se tiene lo siguiente:

```
libro
        index.html
        libro.jsp
        cabeceraLibro.jsp
        menuLibro.jsp
        contenidoLibro.jsp
        capitulo1.html
        capitulo2.html
        capitulo3.html
        estilos.css
        Imagenes
               ecu P.gif
               ecu_P2.gif
               ecu\_R.gif
               ecu_R2.gif
               ecu\_TRC.gif
               ecu_VAN.gif
               ecu VAN2gif
               ecu VAN3gif
               ecu VAN4gif
               ecu VAN5gif
               ecu_VAN6gif
               ecu_VAN7gif
               ecu VAN7 2.gif
               Flecha.gif
               paloma.gif
          WEB-INF/
                faces-config.xml
```

```
web.xml
classes/
mensajes.properties
lib/
common-beanutils.jar
common-collections.jar
common-ligester.jar
common-logging.jar
jsf-api.jar
jsf-impl.jar
jstl.jar
standard.jar
```

Hasta ahora, la página que redireccionaba la aplicación a la página principal, ha sido index.html, cuyo contenido ha sido el mismo hasta ahora. Como novedad, la página index.html de esta aplicación, redirecciona a la página libro.jsp, junto con un parámetro, de nombre capitulo, que es el capítulo que abre en la región de contenido del visor de libro, y recibe el valor capitulo1; de esta manera el visor empezará mostrando el primer capítulo del libro. La página principal index.html sería:

Pagina libro.jsp contiene las tres partes ya mencionadas, cabecera, menu y contenido, estas partes bien diferenciadas, están recogidas en dicheros disintos (cabeceraLibro.jsp, menuLibro.jsp y contenidoLibro.jsp) y su importación en libro.jsp se ha hecho mediante la directva <c:import url="pagina.jsp"/>, para ello, no olvide importar al principio del fichero(junto a las librerías básicas html y librerías core) la librerías core de jstl. El contenido de libro.jsp sería:

```
columnClasses="columnaMenu, columnaCapitulo">
               <f:facet name="header">
                  <f:subview id="cabecera">
                     <c:import url="/cabeceraLibro.jsp"/>
                  </f:subview>
               </f:facet>
               <f:subview id="menu">
                  <c:import url="/menuLibro.jsp"/>
               </f:subview>
               <f:verbatim>
                  <c:import url="/contenidoLibro.jsp"/>
               </f:verbatim>
            </h:panelGrid>
         </h:form>
      </body>
   </f:view>
</html>
```

La cabecera del visor del libro está recogida en cabeceraLibro.jsp, aquí solo se ha incluido una imagen y un texto con el nombre del libro:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<h:panelGrid columns="1" styleClass="cabeceraLibro">
        <h:graphicImage value="/Imagenes/paloma.jpg"/>
        <h:outputText value="#{msjs.titulolibro}" styleClass="tituloLibro"/>
        <f:verbatim><hr></f:verbatim></h:panelGrid>
```

El menu del visor del libro está recogida en menuLibro.jsp, este menu recoge los diferentes enlaces a los capítulos del libro:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<h:panelGrid columns="1" styleClass="columnaMenu">
      <h:commandLink>
         <h:outputText value="#{msjs.capitulo1}"/>
         <f:param name="capitulo" value="capitulo1"/>
     </h:commandLink>
      <h:commandLink>
         <h:outputText value="#{msjs.capitulo2}"/>
         <f:param name="capitulo" value="capitulo2"/>
      </h:commandLink>
      <h:commandLink>
         <h:outputText value="#{msjs.capitulo3}"/>
         <f:param name="capitulo" value="capitulo3"/>
     </h:commandLink>
</h:panelGrid>
```

El contenido del visor del libro está recogida en contenidoLibro.jsp, este solo muestra el contenido de los capítulos (los capítulos se encuentran en formato html):

```
<%@ taglib uri="http://java.sun.com/jstl/core" prefix="c" %>
<c:import url="${param.capitulo}.html"/>
Y por último la hoja de estilos es la siguiente:
.cabeceraLibro {
  width: 100%;
  text-align: center;
  background-color: #eee;
  padding: 0 px;
  border: thick solid #ddd;
}
.tituloLibro {
  text-align: center;
  font-style: italic;
  font-size: 1.3em;
  font-family: Helvetica;
}
.libro {
  vertical-align: top;
  width: 100%;
  height: 100%;
}
.columnaMenu {
  vertical-align: top;
  background-color: #eee;
  width: 100px;
  border: thin solid blue;
}
.columnaCapitulo {
  vertical-align: top;
  text-align: left;
  width: *;
}
```