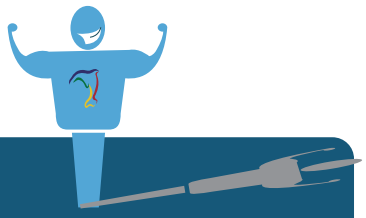
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

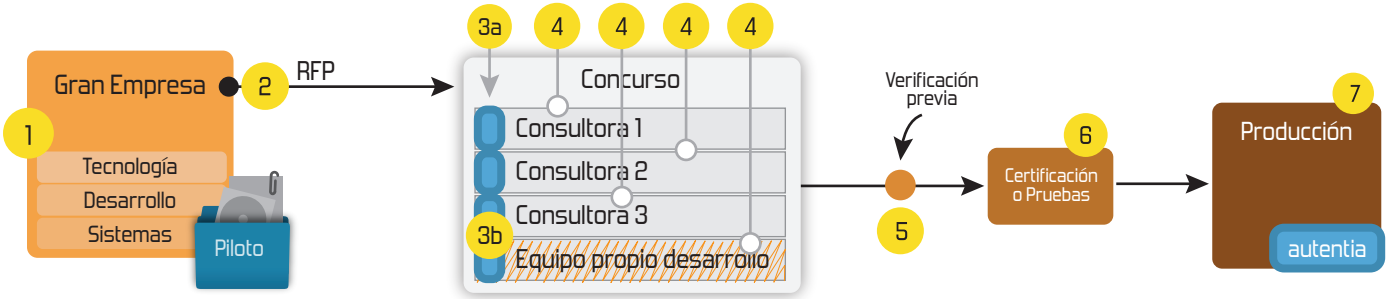
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery

Gestor portales (Liferay)
Gestor de contenidos (Alfresco)
Aplicaciones híbridas

Control de autenticación y acceso (Spring Security)
UDDI
Web Services
Rest Services
Social SSO
SSO (Cas)

JPA-Hibernate, MyBatis
Motor de búsqueda empresarial (Solr)
ETL (Talend)

Dirección de Proyectos Informáticos.
Metodologías ágiles
Patrones de diseño
TDD

Tareas programadas (Quartz)
Gestor documental (Alfresco)
Inversión de control (Spring)

BPM (jBPM o Bonita)
Generación de informes (JasperReport)
ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

Tutorial desarrollado por: Iván Zaera Avellón

Puedes encontrarme en Autentia
Somos expertos en Java/J2EE
Contacta en izaera@autentia.com

autentia REAL BUSINESS SOLUTIONS

Guía de referencia de JSF

Indice

1.Introducción

Esta guía-tutorial pretende dar a conocer todos los conceptos básicos de JSF así como servir de guía de referencia a los desarrolladores.

En ningún caso esta pensada para aprender JSF desde cero, y además presupone que el lector ha cacharreado algo con JSF y tiene conocimientos solidos de JSP, frameworks de desarrollo de aplicaciones visuales (p.e.: Swing, MFCs, ...), así como de aplicaciones web.

Se dan por sentadas muchas cosas que no se explican y que se dejan a la "intuición" del lector. Esta "intuición", sin duda alguna, fallará en sus suposiciones si no se poseen los conocimientos mencionados en el párrafo anterior.

En palabras sencillas: "a mi no me echéis la culpa si no os enteráis; compraos un buen libro de JSF" ;-P. Un buen libro de JSF es, por ejemplo, "Mastering Java Server Faces", de la editorial Wiley (autores: Bill Dudney, Jonathan Lehr, Bill Willis, y LeRoy Mattingly).

2.Componentes

Son los controles que permiten componer la interfaz de usuario (listados, botones, etc.) Todos implementan la interfaz UIComponent, que permite moverse por el árbol de componentes, añadir facetas, validadores, atributos, etc.

Los componentes tienen dos partes: el componente en si (que implementa la funcionalidad) y el productor (renderer). Así, por ejemplo, un campo de entrada normal o uno de contraseña sólo se distinguen por el productor, siendo el componente igual para ambos (UIInput).

2.1.Facetas

Son componentes hijos de otro componente pero que cumplen una función especial. Son relaciones que pueden ser ortogonales y existir fuera del árbol de componentes. Por ejemplo: las etiquetas de título del control de pestañas, las cabeceras de las tablas, etc.

2.2.Atributos y propiedades

En JSF se pueden asignar atributos y/o propiedades a los componentes, beans, validadores, etc. Los atributos se guardan en un mapa y las propiedades tienen setters y getters.

3.Modelo

Dos tipos de modelo: el de componentes visualizados y el de negocio de la aplicación (beans gestionados).

El modelo de componentes visualizados se debe guardar entre petición y petición o bien en sesión, o bien en el cliente (en campo hidden). Esto se controla mediante el parámetro javax.faces.STATE_SAVING_METHOD del servlet de JSF. Puede tomar los valores "client" o "server".

4.Validadores

Existe un atributo en todos los tags de entrada llamado "required" que es de tipo boolean y que define si se debe asignar un validador que obligue a rellenar el campo. Aparte de este atributo, existen algunos validadores estándar de JSF como, p.e., validadores de rango.

Pueden recibir parámetros Siempre que se implementen como clases aparte deben cumplir la interfaz Validator. Hay cuatro formas de uso:

- Tag f:validator:

```
<h:loQueSea>
      <f:validator type="tipo.definido.en.faces.config.xml">
            <f:attribute name="atributo1" value="valor1"/>
                  .
                  .
      </f:validator>
</h:loQueSea>
```

- Tag personalizado (ver más abajo como se implementa un tag personalizado):

```
<h:loQueSea>
<miLibreria:miValidador atributo1="valor1" .../>
</h:loQueSea>
```

- Atributo validator:

```
<h:loQueSea validator="#{bean.funcion}"/>
```

En este caso el validador es una función de un bean con la signatura:

```
public void validate(FacesContext context, UIInput component,
Object value) throws ValidatorException;
```

- Especializar componente y sobreescribir el método validate()

Se pueden implementar etiquetas JSP de usuario para nuevos validadores de usuario. Por supuesto, dichas etiquetas hay que declararlas en un fichero .tld aparte de nuestra creación

Para implementar una etiqueta de usuario de validador hay que heredar de ValidatorTag y llamar a super.setValidatorId() en el constructor. Además, hay que sobreescribir el método createValidator() en el cual llamaremos a super.createValidator() para que nos devuelva el validador referenciado por el id dado en el constructor. Después, inicializamos los atributos de dicho validador basándonos en los atributos pasados al tag.

5.Convertidores

Todos los componentes que heredan de UIOutput pueden tener un convertidor asociado. Sirven para convertir de objeto a cadena y de cadena a objeto. Implementan la interfaz Converter. Pueden tener parámetros, que se pueden fijar usando el tag f:attribute o a través de setters y getters (proveyendo además un tag personalizado para su uso, al estilo de los validadores).

Los tags personalizados deben añadir los convertidores por medio del método setConverter del interfaz UIComponent.

Se declaran en el fichero de configuración de JSF. Pueden tener un tipo asociado, en cuyo caso, no es necesario declararlos explícitamente en los JSPs, sino que JSF los invoca automáticamente en función del tipo de campo de los beans. Para definir un convertidor para un tipo explícito hay que usar <converter-for-class> en el fichero de configuración

6.Eventos y oyentes (Listeners)

Todos los eventos heredan de FacesEvent. Se recomienda que todas las clases de eventos acaben con el sufijo Event. Se pueden crear eventos personalizados.

Cada evento debe definir una interfaz oyente personalizada. Dicha interfaz debe heredar de FacesListener. Igualmente, se suele definir otra interfaz para los componentes que generan eventos de un determinado tipo. Por ejemplo: se define ActionSource para componentes que generan ActionEvents.

JSF define sólo dos tipos de evento y sus correspondientes oyentes:

- ActionEvent/ActionListener: lanzado por componentes UICommand.
- ValueChangedEvent/ValueChangeListener: lanzado por componentes UIInput.

Cada componente debe tener un método addLoQueSeaListener y otro removeLoQueSeaListener por cada interfaz oyente que pueda notificar. Para registrar un oyente hay que usar su correspondiente tag personalizado. Por ejemplo:

```
<h:loQueSea>

<f:valueChangeListener

type="oyente.configurado.en.faces.config"/>

</h:loQueSea>
```

Toda implementación de oyente debe devolver en el método getPhaseId() la fase del ciclo de vida en la que desea recibir el evento. Los eventos se van encolando y procesando en el FacesContext a medida que el ciclo de vida va avanzando.

7.Producción (rendering)

Dos formas de producción:

- Directa: sobreescribiendo los métodos: decode, encodeBegin, encodeChildren y encodeEnd, así como getRendererType() para que devuelva null.
- Delegada: se basa en clases que heredan de la clase abstracta Renderer. Para delegar en un productor hay que hacer que getRendererType() devuelva el tipo de productor a usar. JSF le pedirá ese tipo de productor al RenderKit en uso (ver siguiente capitulo).

7.1.Kits de producción (render kits)

Heredan de RenderKit. Suponen una colección de productores especializados para cierto tipo de dispositivos. Por ejemplo: podemos tener un kit para HTML y otro para WML.

Los kits de definen en faces-config:

```
<render-kit>

...

<renderer>

<renderer-type>tipo.logico.de.renderer</renderer-type>

<renderer-class>clase.renderer</renderer-class>

</renderer>

...

</render-kit>
```

8.Contexto JSF

Se obtiene con FacesContext.getCurrentInstance(). Esta guardado en el thread que atiende la petición mediante TLS (thread local storage). Esto implica que no se deben crear threads extra (a menos que sea por una buena razón) en JSF porque se pierde el contexto.

Permite acceder a los objetos HTTP (petición, sesión, etc.) a través de interfaces de JSF que los encapsulan. También se puede consultar el locale del usuario.

También almacena una lista de mensajes globales así como sendas sublistas por cada componente. Estos mensajes se muestran con <h:messages> y <h:message> respectivamente.

Igualmente, almacena una cola de eventos globales y sendas colas por cada componente.

9.Ciclo de vida JSF

El ciclo de vida completo es:

1. Restaurar vista
2. Aplicar valores de la petición
3. Procesar validaciones
4. Actualizar modelo
5. Invocar aplicación
6. Producir respuesta

Tres escenarios posibles. Cada escenario pasa por distintas fases del ciclo de vida:

- Petición JSF genera respuesta JSF:
  - Ciclo de vida completo
- Petición no-JSF genera respuesta JSF:
  - Restaurar vista
  - Producir respuesta
- Petición JSF genera respuesta no-JSF (ejemplo: generar XML)
  - Restaurar vista
  - Aplicar valores de la petición
  - Procesar validaciones
  - Actualizar modelo
  - Invocar aplicación
  - Desvío a productor no-JSF

La navegación se suele definir en faces-config.xml, pero se puede navegar programáticamente con el siguiente código:

```
FacesContext context = FacesContext.getCurrentInstance();

context.getApplication().getViewHandler().createView(context,

"/pagina.jsp");
```

Hay dos métodos en FacesContext para controlar saltos en el ciclo de vida: renderResponse() salta directamente a "Producir respuesta", responseComplete() acaba directamente el ciclo de vida.

Se pueden generar eventos durante todo el ciclo de vida hasta la fase "Invocar aplicación" inclusive. Los eventos se despachan al final de cada fase.

9.1.Fase del ciclo de vida: Restaurar vista

En esta fase se crea el árbol de componentes. Se puede crear a partir de información existente (en sesión o en un campo hidden, según se haya configurado) o de cero si no había información

Si no se encuentran datos POST o "query string", se pasa directamente a "Producir respuesta".

9.2.Fase del ciclo de vida: Aplicar valores de la petición

En esta fase se almacenan los valores enviados por el cliente en sus respectivos componentes. Se llama a processDecodes() recursivamente (profundidad primero) en cada componente para que actualice sus datos.

Dentro de processDecodes() primero se llama a processDecodes() en los hijos y luego a decode() en uno mismo. El método decode() invoca al convertidor asociado al componente si lo hay. Si la decodificación falla se llama a setValid() y a addMessage() para marcar el componente como erróneo

Si cualquier conversión falla o si algún componente llama a renderResponse() el ciclo de vida salta directamente a "Producir respuesta".

En esta fase todos los valores devueltos se guardan en los componentes (no en el bean) como String (que es lo único que sabe devolver el método getParameter de la HttpServletRequest). Después se intentan convertir mediante Converters a un

tipo Java (p.e: las fechas de String a Date) y se vuelven a guardar en el componente. Esto se hace para que, si falla la conversión, el componente siga teniendo guardado (como String) el valor que tecleo el usuario y, en la fase de producir respuesta, se pueda repintar otra vez, junto con el error de conversión

9.3.Fase del ciclo de vida: Procesar validaciones

En esta fase se llama a processValidators() recursivamente (profundidad primero) en cada componente para que actualice sus datos.

Dentro de processValidators() primero se llama a processValidators() en los hijos y luego a validate() en uno mismo. El método validate() invoca a los validadores definidos y, si alguno falla, llamara a setValid(false) y a addMessage() y el ciclo de vida saltará a "Producir respuesta".

9.4.Fase del ciclo de vida: Actualizar modelo

En esta fase se llama a processUpdates() recursivamente (profundidad primero) en cada componente para que actualice sus datos.

Dentro de processUpdates() primero se llama a processUpdates() en los hijos y luego a updateModel() en uno mismo. El método updateModel() actualiza los beans de modelo asociados al componente (normalmente mediante el atributo value del tag del componente).

En esta fase se pasan los valores almacenados en los componentes (y ya convertidos y validados) al bean.

9.5.Fase del ciclo de vida: Invocar aplicación

En esta fase se procesan todos los ActionEvents encolados.

9.6.Fase del ciclo de vida: Producir respuesta

En esta fase los componentes y sus Renderers se encargan de convertir el árbol de componentes en código visualizable (normalmente HTML).

La producción se realiza mediante la subclase de ViewHandler registrada en JSF. Todas las implementaciones de JSF deben tener un ViewHandler por defecto (normalmente un ViewHandler que procesara páginas JSP).

En esta fase se llama a encodeBegin() en cada componente para generar su contenido. Después, si el método rendersChildren() del componente devuelve true. se llama a encodeChildren() y, finalmente a encodeEnd().

Después de producir la respuesta JSF serializa el árbol de componentes para la siguiente petición Esta serialización se puede guardar en sesión o en un campo hidden (configurable).

10.Configuración

Mediante ficheros XML de configuración que pueden estar en:

- META-INF/faces-config.xml en los distintos JAR de la aplicación
- Los ficheros especificados en el parámetro javax.faces.application.CONFIG_FILES del servlet de JSF
- WEB-INF/faces-config.xml

10.1.Etiquetas XML posibles en la configuración

10.1.1.<application>

Permite modificar el action-listener por defecto (el que enruta los ActionEvents a los beans gestionados), los mensajes locales, manejadores de navegación y vista, el resolvedor de variables y propiedades, los locales soportados por la aplicación, y el locale por defecto.

10.1.2.<factory>

Factorías de aplicación, contexto de JSF, ciclo de vida y render-kit.

10.1.3.<attribute>

Atributo (dinámico) de un componente, renderer, convertidor o validador que permite definir la descripción, nombre descriptivo, icono, nombre programático y tipo de atributo, valor sugerido y valor de usuario (específico de la implementación).

10.1.4.<property>

Propiedad (estática) de un componente, renderer, convertidor o validador que permite definir lo mismo que <attribute>.

10.1.5.<component>

Permite definir un componente a medida y definir su descripción, nombre descriptivo, icono, nombre programático, clase que lo implementa, atributos, propiedades, y valor de usuario (especifico de la implementación).

10.1.6.<converter>

Permite declarar un convertidor y definir su descripción, nombre descriptivo, icono, id, clase implementadora, atributos y propiedades.

10.1.7.<managed-bean>

Permite declarar un bean gestionado (modelo de la aplicación) y definir su descripción, nombre descriptivo, icono, nombre programático, clase que lo implementa, ámbito (sesión, petición, aplicación, ...) y propiedades gestionadas (tag managed-property).

10.1.8.<managed-property>

Propiedad de un managed-bean permite definir descripción, nombre descriptivo, icono, nombre programático, clase de la propiedad, y valor inicial (<map-entries>, <list-entries>, <null-value>, <value>).

10.1.9.<referenced-bean>

Declara un bean referenciado (por oposición a gestionado) que no es creado por JSF, sino que debe existir cuando JSF quiera acceder a el; permite definir la descripción, nombre descriptivo, icono, nombre programático y clase que lo implementa.

10.1.10.<render-kit>

Define un kit de producción nuevo o añade productores al por defecto (según especifiquemos o no un id); permite definir la descripción, nombre descriptivo, icono, id, clase, y una lista de renderers.

10.1.11.<renderer>

Declara un productor dentro de un kit y permite definir la descripción, nombre descriptivo, icono, id programático, clase, atributos, lista de componentes soportados (referenciados por su clase o por su id programático) y un valor de usuario.

10.1.12.<validator>

Declara un validador; permite definir descripción, nombre descriptivo, icono, id, clase, atributos y propiedades.

11.Librerías de tags e integración JSP-JSF

Dos librerías estándar: core (prefijo habitual: f) y html (prefijo habitual: h).

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

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

La html tiene una etiqueta por cada combinación productor-componente, por ejemplo: <h:inputText> y <h:inputSecret> son dos productores distintos para el mismo componente (UIInput).

La core contiene <f:view> que implementa la vista raíz de JSF que contiene el árbol de componentes. La vista es la responsable de serializar el modelo de componentes en el cliente o el servidor. Además de la vista, la core contiene etiquetas multifunción para agregar validadores, conversores, etc. a los distintos componentes. Estas etiquetas llaman, por debajo, a los métodos en la interfaz UIComponent.

Las etiquetas sólo fijan los valores de los atributos si no están previamente fijados. Esto es útil para cambiar los atributos programáticamente y que los tags no nos estropeen la programación Los tags se deben ver como el diseño inicial de la interfaz al crearse la vista. Posteriormente la interfaz se puede modificar programáticamente mediante, p.e., oyentes, sin que los tags nos sobreescriban los atributos.

Los JSPs son JSPs estándar pelados. Los tags son los que realmente están integrando el JSP con JSF, al hacer (por debajo) llamadas al API de JSF manipulando el árbol de componentes y el contexto. Podría ser posible escribir una aplicación JSF entera en Java, sin usar JSPs, manipulando únicamente los componentes.

11.1.Restricciones que debemos cumplir al usar JSPs+JSF

1. Todos los tags de componentes JSF que se embeban dentro de etiquetas no JSF que producen contenido condicionalmente (p.e: c:if) deben tener id. Si no se especifica un id se generaría uno automático unas veces si y otras no, con lo que los ids del resto de controles bailarían, puesto que se generan con un contador secuencial, y la aplicación haría cosas raras.
2. No se pueden usar tags de JSF dentro de etiquetas no-JSF que iteren sobre su contenido (p.e.: c:forEach).
3. No se puede poner texto HTML dentro de etiquetas JSF que puedan tener hijos (no se pinta). Para eso usar <f:verbatim>.

11.2.Inclusión de JSPs en JSPs

Para incluir un JSP dentro de otro en JSF es necesario emplear los tags f:subview y jsp:include. Además, mediante el tag jsp:include podemos pasar parámetros al JSP contenido. Ejemplo:

- Página JSP contenedora:

```
<f:subview id="cabecera"/>

<jsp:include page="cabecera.jsp" flush="true">

<jsp:param name="titulo" value="Nomina"/>

</jsp:include>

</f:subview>
```

- Página JSP contenida (cabecera.jsp), que usa el parámetro enviado:

```
<h:outputText id="cabecera" styleClass="cabecera"

value="#{param.titulo}"/>
```

11.3.Etiquetas con parámetros

Se puede usar el tag f:parameter para pasar parámetros de localización a la etiqueta h:outputText. Ejemplo:

```
<h:outputMessage value="Mensaje con un parámetro: {0}">

<f:parameter value="texto del parámetro 0"/>

</h:outputMessage>
```

12.Tablas

Dos componentes para trabajar con tablas: UIData y UIPanel. El primero es para iterar sobre datos, mientras que el segundo permite crear paneles estáticos de forma libre.

Los tags para trabajar con UIData son h:dataTable y h:column mientras que para el UIPanel debemos usar h:panelGrid y h:panelGroup.

Se pueden especificar las cabeceras/pies del dataTable y de las columnas definiendo facetas con el nombre "header" y "footer" respectivamente.

Los panelGrids tienen un número de columnas especificado en las que van colocando los componentes de izquierda a derecha y de arriba a abajo. Para encapsular varios controles en una sola celda se puede usar panelGroup.

Los ids de los controles dentro de una dataTable llevan como prefijo el id de la tabla, un signo ":", el índice en la tabla, y otro signo ":". Por ejemplo, el componente con id "elComponente" dentro de a tabla con id "laTabla" sito en la fila 7 tendrá el id "laTabla:7:elComponente".

13.Formularios

Como en HTML de toda la vida. Los ids de los controles bajo un tag h:form llevan el id del formulario por delante, seguido de dos puntos. En el siguiente ejemplo:

```
<h:form id="elFormulario">

<h:inputText id="elInput"/>

</h:form>
```

El componente inputText tendría el id "elFormulario:elInput" en el HTML generado.

13.1.Listas, desplegables y botones de radio

Existen varios tags según el tipo de control a emplear. Todos se basan en el mismo componente y todos necesitan una etiqueta f:selectItems anidada para obtener la lista de opciones.

La etiqueta f:selectItems define en su atributo value un método de un bean que devuelve una lista de objetos SelectItem.

13.2.Checkboxes

Usan el tag h:selectBooleanCheckbox y el componente UISelectBoolean y se enganchan a una propiedad de tipo boolean del bean.

13.3.Atributo immediate

Todos los componentes tienen un atributo immediate de tipo boolean que permite, al ser fijado a true, que se salten las fases de conversión y validación al procesar su envío

Es útil para controles que necesitan hacer un submit temporal del formulario para recargar datos del formulario, sin que se procese la transacción real asociada al formulario. Por ejemplo: si tenemos un listado que recarga otro al cambiar de valor, es necesario usar immediate=true en el listado maestro.

13.4.Atributo action

Todos los componentes que lanzan eventos ActionEvent tienen un atributo action para registrar un ActionListener que llame a un método de un bean gestionado. Dicho método debe tener la signatura siguiente:

```
public String método();
```

La cadena devuelta sera un resultado que sera comparado con los nodos "outcome" de las reglas de navegación del faces-config.xml (si se devuelve null se queda en la vista actual). Por ejemplo:

```
<navigation-rule>

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

<navigation-case>

<from-outcome>resultado</from-outcome>

<to-tree-id>/resultado.jsp</to-tree-id>

</navigation-case>

</navigation-rule>
```

También es posible especificar una cadena en el action, en lugar de una llamada a un método En ese caso se usa la cadena como resultado.

13.5.Enlaces, botones y relación con la navegación

Los enlaces y los botones son equivalentes, y ambos tienen atributo action. Cuando un método es invocado a través de una propiedad action nos encontramos con que dicho método no puede recibir parámetros Esto representa un problema cuando producimos un listado de elementos con un enlace en cada uno para, por ejemplo, visualizar el detalle del elemento.

Existen dos formas de saber que elemento se ha seleccionado:

- Pasando un parámetro con <f:param> y recogiéndolo en el método action con la llamada:

```
FacesContext.getCurrentInstance().getExternalContext().

getRequestParameterMap().get("idDelParametro").
```

- Accediendo al control tabla en la parte servidor y obteniendo el objeto de negocio asociado a su fila seleccionada:

```
FacesContext facesContext = FacesContext.getCurrentInstance();

UIViewRoot root = facesContext.getViewRoot();

UIData table = (UIData)root.findComponent(“idDelFormulario”)

.findComponent(“idDeLaTabla”);

ObjetoNegocio seleccionado = (ObjetoNegocio)table.getRowData();
```

La segunda es más bonita por aquello de que no hay que andar cogiendo parámetros sino que se accede al componente, pero es una cuestión de gustos.

Para pasar el objeto seleccionado al bean asociado a la siguiente página podemos aplicar tres métodos Estos trozos de código habría que añadirlos en el método que procese el action que provoque el cambio de página. Justo después de ellos devolveríamos la cadena que hace que se navegue a la siguiente vista:

- Utilizar el mismo bean para las dos páginas y guardarlo en sesión, quedándonos una copia del objeto seleccionado dentro de un atributo del bean:

```
this.objetoActivo = seleccionado;
```

- Guardar en la petición el objeto seleccionado para que el siguiente bean lo recupere en un getter al inicializar la página:

```
HttpServletRequest request = (HttpServletRequest)

facesContext.getExternalContext().getRequest();

request.setAttribute("objetoActivo", seleccionado );
```

- Acceder desde el API JSF al siguiente bean y pasarle el objeto explícitamente:

```
ValueBinding binding = Util.getValueBinding(“#{siguienteBean}”);

ClaseSiguienteBean bean = (ClaseSiguienteBean)binding

.getValue(facesContext);

bean.setObjetoActivo( seleccionado );
```

Sin duda, el tercer método es el más elegante, desacoplado y mantenible (la hueva, vamos :-P). Es importante darse cuenta de que cuando se usa dicho método, no es necesario que el bean “siguienteBean” este en sesión, sino que JSF lo creara, si es necesario, al llamar a Util.getValueBinding() (al estilo de como lo hacen los tags JSF).

Por otro lado, si los beans están en sesión, puede ser deseable, al llamar al setter, pasar un clon del objeto seleccionado, en lugar del objeto seleccionado en si. De esta forma evitamos que cambios en la página de detalle se propaguen al listado.

13.6.Atributo actionListener y valueChangeListener

Estos atributos permiten, en los componentes en los que son aplicables, definir oyentes sin necesidad de implementar la interfaz ActionListener o ValueChangeListener entera.

Los atributos deben apuntar a un método del bean con las siguientes signaturas:

- `actionListener: void método( ActionEvent evento );`

- `valueChangeListener: void método( ValueChangeEvent evento );`

Generalmente se puede programar JSF sin usar oyentes, simplemente haciendo uso del atributo action. Sin embargo, es buena práctica emplearlos para ejecutar código que no tiene que ver con el método apuntado por action.

Por ejemplo, si en el action de un botón se llama al método "guardar()" y esto provoca como efecto directo que se salve un registro en la base de datos y como efecto colateral que se oculte el botón y se muestre una pantalla de confirmación, es conveniente poner el código para ocultar el botón en un oyente y dejar en el método "guardar()" sólo el código que guarda el objeto de negocio. Así separamos el código de manejo de la interfaz del código que hace cosas reales (llamar a la aplicación para que guarde el registro). No obstante, esto es un poco en plan purista, y yo seguiría durmiendo tranquilo si ocultase el botón en el mismo método guardar().

En general, sólo hay que usar Listeners cuando son estrictamente necesarios o cuando queremos separar código (como en el ejemplo anterior del botón y el método guardar()). Si se puede hacer algo sin necesidad de listeners, ¿para que complicarse la vida?.

14.Implementación de componentes JSF

Un componente consta de tres partes: una clase derivada de UIComponent u otro control ya existente, un productor, y un tag JSP de usuario. No siempre es necesario escribir las tres piezas.

Un componente debe cumplir los siguientes requisitos (algunos de los cuales no nos deben preocupar puesto que vienen implementados a través de la clase UIComponentBase, de la que podemos heredar):

- Implementar javax.faces.component.StateHolder para guardar el estado del componente entre petición y petición
- Implementar el método "decode()" para que el componente obtenga los valores enviados cuando no tenga productor asociado. No obstante es mejor proveer un productor (Renderer) por defecto en la misma librería que el componente y hacer que el componente delegue en el cuando no haya un productor explicito.
- Implementar el método validate() si fuese necesaria alguna validación especial. No obstante, es conveniente delegar todas las validaciones susceptibles de reutilización a clases de tipo Validator.
- Implementar "updateModel()" si la implementación por defecto en UIComponentBase no nos sirve (por ejemplo, porque estemos implementado un componente que agrega varios y se pueda enlazar a más de una propiedad del bean).
- Implementar encodeBegin(), encodeChildren() y encodeEnd() para pintar el componente. Al igual que en decode(), es conveniente delegar en un productor si no se especifica ninguno (en lugar de proveer el código de producción directamente en la clase del componente).

Dada la complejidad de este punto, es recomendable consultar el código fuente de la implementación de JSF para ver que hacen los controles estándar en cada método: es la única forma de aprender en este tipo de frameworks (esta es mi opinión personal, pero fijaos en que estoy cargadísimo de razón ;-)).

15.Arquetipos Maven para MyFaces

Para los que uséis Maven, el proyecto MyFaces ha definido dos arquetipos que permiten crear el esqueleto de una aplicación web basada en MyFaces y el esqueleto de una librería de componentes de usuario. Es altamente recomendable utilizarlos para evitar tener que empezar los proyectos de cero.

En el caso de la librería de componentes, el arquetipo crea incluso un control con su tag, su productor y su componente, y con varios de sus métodos ya implementados.

Si decidís usarlos, visitad la web de MyFaces (ver capitulo final “Enlaces interesantes”) porque el plugin aun no esta liberado y tenéis que bajaros el código con Subversion, compilarlo e instalarlo (con “mvn install”, por supuesto) antes de usarlo.

16.Enlaces interesantes

- Consola JSF: http://www.jamesholmes.com/JavaServerFaces/console/
- Arquetipos Maven para MyFaces: http://wiki.apache.org/myfaces/MyFaces_Archetypes_for_Maven

Puedes opinar sobre este tutorial aquí

Recuerda

que el personal de Autentia te regala la mayoría del conocimiento aquí compartido (Ver todos los tutoriales)

¿Nos vas a tener en cuenta cuando necesites consultoría o formación en tu empresa?

¿Vas a ser tan generoso con nosotros como lo tratamos de ser con vosotros?

info@autentia.com

Somos pocos, somos buenos, estamos motivados y nos gusta lo que hacemos .....
Autentia = Soporte a Desarrollo & Formación

Autentia S.L. Somos expertos en:
J2EE, Struts, JSF, C++, OOP, UML, UP, Patrones de diseño ..
y muchas otras cosas

Nuevo servicio de notificaciones

Si deseas que te enviemos un correo electrónico cuando introduzcamos nuevos tutoriales, inserta tu dirección de correo en el siguiente formulario.

Subscribirse a Novedades	
e-mail	
	Enviar

Otros Tutoriales Recomendados (También ver todos)

Nombre Corto	Descripción
JSF en Java Studio Creator 2	En este tutorial os mostramos como realizar una aplicación JSF utilizando la herramienta Java Studio Creator en su segunda versión
Validar en JSF con Commons Validator	En este nuevo tutorial sobre el framework JSF os mostramos como utilizar y extender la validación del Commons Validator
Probando entornos para JSF	En este tutorial os mostramos con ejemplos como utliizar dos conocidos entornos de desarrollo para JSF: Exadel Studio y Sun Studio Creator
Conversión y validación en JSF	En este nuevo tutorial sobre JSF os mostramos como utilizar y extender los mecanismos básicos de conversión y validación
JSF y comparativa con Struts	Os mostramos los pasos necesarios para empezar a utilizar JSF (Java Server Faces) y su comparación / relación con Struts
Utilizando JSTL en JSF	Os mostramos como utilizar la librería estandar de etiquetas en JSF, implementando una sencilla aplicación web
JSF y NetBeans 5.5	Os mostramos como dar vuestros primeros pasos utilizando Java Server Faces (JSF) con ayuda del conocido entorno de desarrollo NetBeans
Manejar tablas de datos con JSF	En este tutorial os mostramos un ejemplo de utilización de la extension del componente DataTable, realizada por la implementación Tomahawk de MyFaces
Pruebas unitarias Web para aplicaciones JSF	En este tutorial se puede encontrar una introducción y un análisis de los diferentes frameworks disponibles para realizar pruebas unitarias web de aplicaciones JSF
Upload de ficheros en JSF	Os mostramos de una forma sencilla y guiada como crear una utilidad de upload de ficheros utilizando JSF