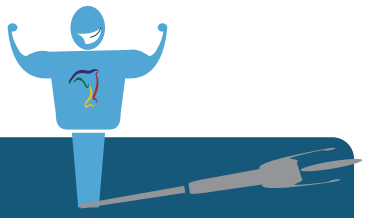
autentia

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...

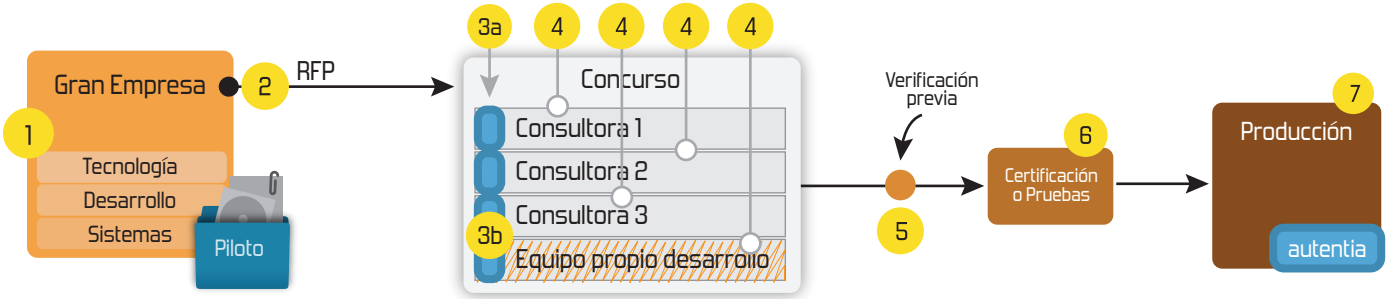
1. Desarrollo de componentes y proyectos a medida



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

Fecha de creación del tutorial: 2007-07-05

Como hacer un componente de JSF

Creación: 27-06-2007

Índice de contenidos

1. Introducción

Ya hemos visto algunas cosas sobre JSF (http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=jsf). Básicamente podríamos decir que se trata de un estándar que ya forma parte de la especificación de Java EE 5, donde el desarrollo se hace en base a componentes.

Lo ideal es usar los componentes estándar o usar librerías de componentes ya construidas, como Tomahawk (http://myfaces.apache.org/tomahawk/) o ICEfaces (http://www.icesoft.com/products/icefaces.html). Pero en algunas ocasiones no encontraremos ningún componente que cubra nuestras necesidades.

Para estas ocasiones siempre nos queda la opción de construirnos nuestros propios componentes. Esto es lo que vamos a ver en este

tutorial.

En concreto vamos a construir un componente que, indicándole un POJO, nos pinte un formulario con una etiqueta y un campo de entrada para cada una de las propiedades de tipo String de ese POJO.

¡¡¡ Atención, el componente que vamos a construir sólo pretende ilustrar la construcción de un componente, pero todavía le quedará mucho para poder ser usando en producción !!!

Y ya, entrando en harina, podemos adelantar que para la construcción de un componente vamos a desarrollar tres piezas:

- el componente: esta clase es realmente el componente, y es donde estará implementada la lógica asociada.
- el render: es la clase que se encarga de pintar el componente y de recuperar la información introducida por el usuario para "inyectarla" en el componente. El render está asociado a un medio concreto (web, wap, ...), mientras el componente es único. Es decir, podemos tener varios render para un mismo componente, o incluso reescribir los render si queremos cambiar el "como se pinta".
- el tag: lo normal (aunque no obligatorio) es que nuestro componente lo queramos usar cómodamente en páginas JSP. El tag será la clase que implemente eso, una etiquetad de JSP.

En los siguientes apartados se ira viendo como construimos cada una de estas piezas.

Vamos a aprovechar y vamos a recordar el ciclo de vida de JSF. Para el desarrollo en JSF (bien sea de aplicaciones o de componentes) es fundamental comprender y dominar las 6 fases del ciclo de vida de JSF:

1. **Restore view**: Se reconstruye el árbol de componentes.
2. **Apply request values**: Se leen los valores de la request y se aplican sobre los componentes. En este momento es cuando se llama a los converters. Si hay algún error en la conversión se irá directamente a la fase "render response". Si un componente tiene "immediate=true" su validación (y el procesamiento de los eventos provocados por esa validación) se hará en esta fase .
3. **Process validations**: se validan todos los componentes (todos los que tienen "immediate=false", ya que los que lo tienen a true ya se validaron en la fase anterior). Si falla alguna de las validaciones se irá directamente a la fase "render response".
4. **Update model values**: Los valores de los back beans de JSF se actualizan con los valores que hasta ahora estaban guardados sólo en los componentes.
5. **Invoke application**: Se invoca a los métodos de los back beans.
6. **Render response**: se pinta la respuesta al usuario. Se pinta el árbol de componentes.

Podéis encontrar más información en http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSFIntro10.html.

Aquí podéis encontrar el código completo de las tres clases necesarias para construir el componente.

2. Entorno

El tutorial está escrito usando el siguiente entorno:

- Hardware: Portátil Asus G1 (Core 2 Duo a 2.1 GHz, 2048 MB RAM, 120 GB HD).
- Sistema Operativo: GNU / Linux, Debian (unstable), Kernel 2.6.21, KDE 3.5
- Máquina Virtual Java: JDK 1.6.0-b105 de Sun Microsystems
- Eclipse 3.2.2
- MyFaces 1.1.5
- Maven 2.0.7

3. El componente

A la hora de construir un componente debemos fijarnos si ya hay alguno que hace algo parecido, ya sea dentro del estándar, o de alguna librería que hayamos adquirido. Si ya tenemos un componente que hace algo parecido la mejor opción suele ser que nuestra clase extienda de ese componente para aprovechar las funcionalidades del padre. Si no encontramos ningún componente que podamos reutilizar tendremos que extender de la clase `javax.faces.component.UIComponentBase`.

En nuestro ejemplo vamos usar el segundo camino, extenderemos la clase `UIComponentBase`. Vamos a ir viendo paso a paso esta clase (recordar que en al final de la introducción tenéis acceso un tar.gz con el código completo):

```
public static final String COMPONENT_TYPE = FormBeanComponent.class.getName();
public static final String DEFAULT_RENDERER_TYPE = HtmlFormBeanRenderer.class.getName();
```

Estas dos constantes son cadena arbitrarias, aunque hemos decidido usar los nombres de las clases, se podría haber puesto cualquier otra cosa. Luego las utilizaremos para determinar el render que se tiene que usar para pintar el componente.

```
private ValueBinding beanBinding = null;
```

En este atributo guardaremos el bean a partir del cual vamos a pintar el formulario, y donde se guardaran los valores introducidos por el usuario (en la fase "update model values"). Es de tipo ValueBinding para poder usar lenguaje de expresiones de JSF para darle valor en la JSP.

```
private Map<String, String> localValues = new HashMap<String, String>();
```

Según hemos visto en el ciclo de vida, los valores de la request se almacenan en el componente hasta alcanzar la fase "update model values". Este atributo lo usaremos para guardar esos valores recuperados de la request. Ya hemos dicho que nuestro componente sólo trabajará con las propiedades de tipo String, así que en este mapa guardaremos como clave el nombre de la propiedad y como valor, el valor introducido por el usuario.

```
public FormBeanComponent() {
    setRendererType(DEFAULT_RENDERER_TYPE);
  }
```

Constructor de la clase. Es importante llamar a `setRenderType()`, este es un método de nuestro padre que permite definir cual será el identificador del render a usar. Si no hacemos esto luego JSF no será capaz de elegir el render apropiado para pintar el componente.

```
@Override
public String getFamily() {
    return COMPONENT_TYPE;
  }
```

Este es el único método que estamos obligados a implementar por extender la clase UIComponentBase. Este método devuelve el identificador de la familia a la que pertenece este componente. JSF, en función de este valor y del tipo de render (lo hemos definido en el constructor), elegirá el render apropiado para pintar este componente.

```
public Object getBean() {
    final Object bean = beanBinding.getValue(FacesContext.getCurrentInstance());
    return bean;
  }
```

Este método devuelve el POJO que hemos asociado al componente (el POJO que usaremos para pintar el formulario y posteriormente para guardar los valores). Nótese como se extrae el valor del atributo `beanBinding` (dijimos que este atributo va a hacer referencia al POJO mediante lenguaje de expresiones de JSF). También hay "setter" correspondiente para fijar el valor del POJO (se puede ver en el código completo).

```
@Override
public void restoreState(FacesContext context, Object state) {
    final Object values[] = (Object[])state;
    super.restoreState(context, values[0]);
    beanBinding = (ValueBinding)restoreAttachedState(FacesContext.getCurrentInstance(), values[1]);
  }

@Override
public Object saveState(FacesContext context) {
    List<Object> state = new ArrayList<Object>();
    state.add( super.saveState(context) );
    state.add( saveAttachedState( FacesContext.getCurrentInstance(), beanBinding ) );
    return state.toArray();
  }
```

Estos dos métodos vienen de la interfaz `javax.faces.component.StateHolder`. La implementación de esta interfaz permite a un componente mantener valores entre diferentes request. Esto es necesario porque en cada request se reconstruye todo el árbol de componentes (recordar la fase "restore view"), es decir se crean nuevas instancias de cada componente. Para nuestro componente es importante guardar el POJO al que se está haciendo referencia. Como estamos sobreescribiendo los métodos de nuestro padre, es muy importante que no se nos olvide llamar a `super`.

```
@Override
public void processUpdates(FacesContext facesContext) {
    final Object bean = getBean();
    final Class<?> clazz = bean.getClass();
    final Class[] parameterTypes = { String.class };

    for (Map.Entry newValue : localValues.entrySet()) {
        final String key = newValue.getKey().toString();
        final String setterName = "set" + key.substring(0, 1).toUpperCase() + key.substring(1);
                Method setterMethod;
        try {
            final Object[] setterArguments = { newValue.getValue() };
                        setterMethod = clazz.getMethod(setterName, parameterTypes );
                        setterMethod.invoke(bean, setterArguments);

        } catch (Exception e) {
                        e.printStackTrace();
                }
      }
  }
```

Este método se va a invocar en la fase de "update model values", y es el que se tiene que encargar de actualizar el back bean (en nuestro caso el POJO que tenemos referenciado con el atributo `beanBinding`) con los valores que están guardados localmente en el componente (en nuestro caso los valores que tenemos en el atributo `localVlaues`). Igual que esté método, tenemos métodos similares por si tenemos que hacer cosas en el resto de las fases (`processRestoreState`, `processDecodes`, `processValidators`, ...).

El código lo que hace es recorrer el mapa con los valores locales que ha recogido el componente de la request y buscar esas propiedades en el POJO por introspección. En cualquier caso no es más que un ejemplo y no importa tanto su implementación como que quede clara la

responsabilidad del método, y que este se invoca en la fase "update model values".

Con esto hemos terminado con el componente (se han omitido algunos getter y setter que podéis ver en el código completo). Cabría destacar especialmente los métodos para guardar el estado del componente entre request (`restoreState`, `saveState`) y el método que actualiza los valores de los back beans (`processUpdates`).

4. El render

Esta clase será la que se encargue de recuperar los parámetros de la request http y pasárselos al componente, y de consultar el componente para "pintar" el HTML. Esto lo podría haber hecho el propio componente pero no es recomendable. Lo bueno de los componentes de JSF es que podemos reutilizarlos para diferentes dispositivos, y esto lo conseguiremos teniendo diferentes renders para un sólo componente.

Para implementar un render extenderemos de la clase `javax.faces.render.Renderer`.

```
@Override
public void decode(FacesContext facesContext, UIComponent uiComponent) {
   final FormBeanComponent formBeanComponent = (FormBeanComponent)uiComponent;

   // Se va a usar el POJO asociado al componente para buscar las propiedades,
   // pero los valores recogidos de la request se almacenaran en la variable local.
   final Object componentValue = formBeanComponent.getBean();
   final Map<String, String> localValues = formBeanComponent.getLocalValues();
       localValues.clear();

   final Class<?> clazz = componentValue.getClass();
   final Method[] methods = clazz.getMethods();
   for (Method method : methods) {
      final String methodName = method.getName();
      final String propertyName = methodName.substring(3);

      if (method.getParameterTypes().length == 0
            && method.getReturnType() == String.class
            && methodName.startsWith("get")) {
         // Sabemos que es un geter, ahora hay que comprobar si existe el setter equivalente
         final Class[] parameterTypes = { method.getReturnType() };
         final Method setterMethod;
         try {
            setterMethod = clazz.getMethod("set" + propertyName, parameterTypes);
         } catch (Exception e) {
            continue; // no existe setter equivalente, así que seguimos buscando propiedades.
             }
         if (setterMethod.getReturnType() != Void.TYPE) {
            continue; // no es realmente un setter ya que un setter no debería tener tipo de retorno.
             }

         // Se busca en la request el nombre de la propiedad
         final String requestParameterName
                        = propertyName.substring(0,1).toLowerCase() + propertyName.substring(1);
         final String requestParameter
               = (String)JsfUtils.getRequestParametersMap().get(requestParameterName);

            localValues.put(requestParameterName, requestParameter);
       }
    }
 }
```

Este método `decode` se encarga de extraer los parámetros de la request de http, y guardarlos en la variable local del componente. Igual que antes, no es tan importante la implementación del algoritmo, como que quede claro que este método es el que se llama en la fase de "apply request values" para guardar los parámetros de la request en el componente.

```
@Override
public void encodeEnd(FacesContext facesContext, UIComponent uiComponent) throws IOException {
   final ResponseWriter out = facesContext.getResponseWriter();
   final FormBeanComponent formBeanComponent = (FormBeanComponent)uiComponent;
   final Object bean = formBeanComponent.getBean();
   final Class<?> clazz = bean.getClass();
   final Object[] getterArguments = {};

   out.startElement("table", null);
   for (Method method : clazz.getMethods()) {
      final String methodName = method.getName();
      final String propertyName = methodName.substring(3);

      if (method.getParameterTypes().length == 0
            && method.getReturnType() == String.class
            && methodName.startsWith("get")) {
         // Sabemos que es un geter, ahora hay que comprobar si existe el setter equivalente
            Class[] parameterTypes = { method.getReturnType() };
         final Method setterMethod;
         try {
            setterMethod = clazz.getMethod("set" + propertyName, parameterTypes);
         } catch (Exception e) {
            continue; // no existe setter equivalente, así que seguimos buscando propiedades.
             }
         final Class setterReturnType = setterMethod.getReturnType();
```

```
            if (setterReturnType != Void.TYPE) {
                continue; // no es realmente un setter ya que un setter no debería tener tipo de retorno.
                }

            // Se vuelca en la salida el campo de entrada para la propiedad
            final String outParameterName
                                = propertyName.substring(0,1).toLowerCase() + propertyName.substring(1);
            out.startElement("tr", null);
            out.startElement("td", null);
            out.startElement("label", null);
            out.writeAttribute("for", outParameterName, null);
            out.writeText(outParameterName, null);
            out.endElement("label");
            out.endElement("td");
            out.startElement("td", null);
            out.startElement("input", null);
            out.writeAttribute("type", "text", null);
            out.writeAttribute("name", outParameterName, null);
            try {
                final String value = (String)method.invoke(bean, getterArguments);
                    out.writeAttribute("value", value, null);
            } catch (Exception e) {
                throw new IOException(e);
                }
            out.endElement("input");
            out.endElement("td");
            out.endElement("tr");
              }
          }
    out.endElement("table");
      }
```

Para "pintar" el componente tenemos los métodos `encodeBegin`, `encodeChildren` y `encodeEnd`. Estos tres métodos permiten pintar la etiqueta inicial, pintar el cuerpo o etiquetas anidadas, y pintar el final de la etiqueta. En los casos en los que no es necesario hacer esta separación (como en nuestro caso), se mete todo el código en el método `encodeEnd`. Nuevamente recordar que no es más que un ejemplo.

5. El tag

Como último paso vamos a crear un tag de JSP para poder usar cómodamente nuestro componente en una página JSP. Para ello vamos a extender de la clase `javax.faces.webapp.UIComponentTag`. Veamos como queda nuestra clase:

```
private String bean = null;
```

El tag admitirá un parámetro donde se recogerá el lenguaje de expresiones de JSF que indica el POJO que se usará en el componente. En el atributo `bean` será donde guardemos la cadena con este lenguaje de expresiones.

```
@Override
public String getComponentType() {
      return FormBeanComponent.COMPONENT_TYPE;
    }

@Override
public String getRendererType() {
      return FormBeanComponent.DEFAULT_RENDERER_TYPE;
    }
```

Estos dos métodos estamos obligados a sobreescribirlos por extender de `UIComponent`. `getComponentType()` sirve para indicar la familia del componente, y `getRenderType()` el tipo de render que sabe pintarlo. Estos dos métodos determinarán el render que hay que usar para pintar el componente cuando se procese el tag. Se están usando las constantes que habíamos definido en la clase del componente.

```
@Override
protected void setProperties(UIComponent component) {
   super.setProperties(component);

   final Application application = FacesContext.getCurrentInstance().getApplication();
   final FormBeanComponent formBeanComponent = (FormBeanComponent)component;

   if (bean != null) {
      // Con isValueReference comprobamos si el valor del atributo puesto
      // en la JSP se trata de EL (Expresion Languaje #{ } )
      if (isValueReference(bean)) {
         formBeanComponent.setBeanBinding(application.createValueBinding(bean));
          }
       }
    }
```

Sobreescribimos este método para procesar los parámetros indicados en el tag en la JSP. Se comprueba que el atributo `bean` tenga valor, y que este realmente esté apuntando a un back bean. No se nos debe olvidar llamar a `super`. Nótese como el parámetro `component` es nuestro componente ya creado por JSF.

```
@Override
public void release() {
      super.release();
      bean = null;
```

```
}
```

Los tags se reutilizan, así que es muy conveniente sobreescribir el método `release()` para no encontrarnos con sorpresas.

Toda librería de tags necesita un descriptor (*.tld) para que el servidor sepa manejarlo. Para nuestro tag tendremos el siguiente descriptor:

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE taglib
      PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
      "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">

<taglib>
      <tlib-version>1.0</tlib-version>
      <jsp-version>1.2</jsp-version>
      <short-name>LibrerÃa de etiquetas de validaciÃ³n para JSF</short-name>
      <uri>app-jsf-validator-taglibrary_1.0</uri>
      <display-name>app-jsf-validator</display-name>
      <description>Etiquetas para JSF</description>

      <tag>
            <name>beanFormComponent</name>
            <tag-class>com.app.web.jsf.component.FormBeanTag</tag-class>
            <body-content>JSP</body-content>
            <description>
                        Etiqueta para usar el componente personalizado en la una JSP
            </description>
            <attribute>
                  <name>bean</name>
                  <required>true</required>
                  <rtexprvalue>false</rtexprvalue>
                  <type>String</type>
            </attribute>
      </tag>
</taglib>
```

Básicamente estamos indicando el nombre del tag "`beanFormComponent`", la clase que lo implementa, y que tiene un atributo "`bean`" de tipo String que es obligatorio.

6. Ejemplo de uso

Ya tenemos todas las piezas para usar nuestro componente a medida. Una JSP de ejemplo podría ser la siguiente:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://myfaces.apache.org/tomahawk" prefix="t" %>
<%@ taglib uri="app-jsf-tags" prefix="a" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<f:view>
<f:loadBundle basename="MessageResources" var="msg"/>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>

<body>

<h:form id="form">
   <a:beanFormComponent bean="#{contact}" />

   <h:commandButton id="button" action="#{contactCtrl.saveContact}" value="#{msg['btn.save']}" />
</h:form>

</body>

</f:view>

</html>
```

Arriba se incluye el taglib `uri="app-jsf-tags"` (para ver el significado de esta uri ver el web.xml del tar.gz con el código completo), y le damos el prefijo "`a`". Ahora para usar la etiqueta sólo tenemos que hacer:

```
   <a:beanFormComponent bean="#{contact}" />
```

Donde "`#{contact}`" es el lenguaje de expresiones de JSF que indica el POJO que vamos a vincular al componente. Ojo "`contact`" deberá existir en algún ámbito (request, session, aplicación), o estar definido el faces-config.xml.

Y hablando del faces-config.xml, no se nos puede olvidar dar de alta en este fichero tanto el componente como el render que lo va a pintar:

```
<!-- components -->
<component>
      <component-type>com.app.web.jsf.component.FormBeanComponent</component-type>
      <component-class>com.app.web.jsf.component.FormBeanComponent</component-class>
</component>

<!-- renderkit -->
<render-kit>
      <renderer>
            <component-family>com.app.web.jsf.component.FormBeanComponent</component-family>
            <renderer-type>com.app.web.jsf.component.HtmlFormBeanRenderer</renderer-type>
            <renderer-class>com.app.web.jsf.component.HtmlFormBeanRenderer</renderer-class>
      </renderer>
</render-kit>
```

Primero hemos dado de alta el componente hemos definido la clase que lo implementa y el tipo del componente (corresponde con la constante `COMPONENT_TYPE` que habíamos definido en la clase `FormBeanComponent`).

Luego damos de alta el render, indicamos su clase, el tipo de render (corresponde con la constante `DEFAULT_RENDER_TYPE` que habíamos definido en la clase `FormBeanComponent`), y la familia de componentes que sabe pintar (corresponde con la constante `COMPONENT_TYPE` que habíamos definido en la clase `FormBeanComponent`).

7. Conclusiones

Recordamos que este componente que hemos construido es simplemente un ejemplo, y que hay cosas que no se han tenido en cuenta (como por ejemplo las validaciones, o uso de los converters). Cuando implementéis vuestros componentes tenéis que prestar mucha atención sobre el ciclo de vida completo.

Y ya sabéis, intentar usar algo que ya exista (siempre se más fácil y más barato integrar que desarrollar), construir a partir de algo que ya exista, y si no os queda más remedio hacerlo de cero. Y siempre tendréis una cuarta opción: contactar con nosotros en www.autentia.com para que os echemos una mano ;)

8. Sobre el autor

Alejandro Pérez García, Ingeniero en Informática (especialidad de Ingeniería del Software)

Socio fundador de Autentia (Formación, Consultoría, Desarrollo de sistemas transaccionales)

mailto:alejandropg@autentia.com

Autentia Real Business Solutions S.L. - "Soporte a Desarrollo"

http://www.autentia.com

Puedes opinar sobre este tutorial aquí

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
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
Utilizando JSTL en JSF	Os mostramos como utilizar la librería estandar de etiquetas en JSF, implementando una sencilla aplicación web
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
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
Proyecto con JSF Myfaces, Maven y Eclipse	En este tutorial vamos a aprender a construir una aplicación básica JSF (Java Server Pages) utilizando el Maven 2.0 y las bibliotecas de MyFaces. Lo mejor de todo es que para crear el ejemplo no vamos a programar ni una línea.
Introducción a Ajax4Jsf	En este tutorial se hablará de Ajax4jsf, una librería open source que se integra totalmente en la arquitectura de JSF y extiende la funcionalidad de sus etiquetas dotándolas con tecnología Ajax de forma limpia y sin añadir código Javascript.
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
Upload de ficheros en JSF	Os mostramos de una forma sencilla y guiada como crear una utilidad de upload de ficheros utilizando JSF
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