# -Manual de Teoría-
# JavaServer Faces

# INDICE

# 1. JSF

JavaServer Faces (JSF) es un framework basado en el patrón MVC (Modelo Vista Controlador) para aplicaciones Java basadas en web que simplifica el desarrollo de interfaces de usuario en aplicaciones Java EE.

JSF pretende normalizar y estandarizar el desarrollo de aplicaciones web.

Ofrece una clara separación entre el comportamiento y la presentación, lo que permite a cada miembro del equipo de Desarrollo de una aplicación Web enfocarse en su parte del proceso de desarrollo, y proporciona un sencillo modelo de programación para enlazar todas las piezas.

Lo que nos ofrece JSF es:

- Un modelo de trabajo basado en componentes UI (user interface), definidos por medio de etiquetas y XML.
- Una arquitectura basada en el patrón MVC.
- Asocia (de forma modular) cada componente gráfico con los datos (beans de respaldo).
- Incluye la capa de control, definida de forma declarativa en archivos XML. Lo que implica control de eventos y errores.
- Validación en cliente y en servidor.
- Control de mensajes y roles.
- etc.

# 2. EJEMPLO

Vamos a desarrollar un ejemplo con JSF para ir contando sus componentes y como configurarlos.

El ejemplo consistirá en presentar un formulario al usuario para que introduzca su nombre, al pulsar el submit del formulario los datos se almacenarán en un componente ManagedBeans y se redirigirá hacia una página donde se mostrará un mensaje de bienvenida personalizado.

## FORMULARIO DE ENTRADA

En JSF 2.0. usaremos páginas xhtml en vez de jsp, estas reciben el nombre de Facelets.

A continuación presentamos index.xhtml que será la encargada de mostrar el formulario al usuario.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Facelet Title</title>
    </h:head>
    <h:body>
        <h:form>
            <h:inputText id="usuario" value="#{saludoBean.nombre}" />
            <h:commandButton id="submit" action="respuesta" value="Enviar" />
        </h:form>
    </h:body>
</html>
```

**Gráfico 1. index.xhtml**

Como podemos comprobar se trata de un documento xml.

Hemos incluido la librería de etiquetas html propia de JSF, gracias a ella podemos crear el formulario con componentes propios.

El value de la caja de texto incluye un lenguaje de expresiones propio de JSF con el podremos especificar que el dato que inserte el usuario se almacenará en el bean saludoBean, concretamente en su propiedad nombre.

En el botón, mediante el atributo action especificamos que redirigiremos al usuario a la página respuesta.xhtml.

A continuación vemos SaludoBean. Esta clase es un ManagedBean o también conocidos como beans de respaldo, su misión es almacenar los datos y los métodos que se manejarán desde la vista.

Como podemos comprobar se trata de un simple JavaBean  anotado mediante la anotación ManagedBean.

La anotación RequestScoped indica que la instancia de este bean solo tiene ámbito de aplicación, por lo cual los datos almacenados una vez se envíe la respuesta al usuario desaparecerán.

```
@ManagedBean
@RequestScoped
public class SaludoBean {

    private String nombre;

    /** Creates a new instance of SaludoBean */
    public SaludoBean() {
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

**Gráfico 2. SaludoBean**

En la pagina respuesta.xhtml se muestra un mensaje de bienvenida.

Gracias al lenguaje de expresiones podemos recuperar el valor de la propiedad nombre del bean saludoBean.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Facelet Title</title>
    </h:head>
    <h:body>
        Saludos #{saludoBean.nombre}!! desde mi primera aplicacion JSF 2
    </h:body>
</html>
```

**Gráfico 3. respuesta.xhtml**

Todo el código de este ejemplo lo encontrareis en **1. JSF Saludo.rar**

# 3. MANAGED BEANS

Son componentes donde encapsular la lógica de negocio de la aplicación. En el paradigma MVC actúan como Modelo.

Los recursos declarados en un Managed Bean pueden ser:

- Propiedades
- Propiedades manejadas
- Métodos de acción
- Métodos de navegación
- Métodos de validación

```
@ManagedBean
@RequestScoped
public class ProductoBean {

    private int id;
    private String nombre;
    private double precio;

    @ManagedProperty(value="#{miDao}")
    private ProductosDAO dao;

    public String altaProducto(){
        Producto p = new Producto(id, nombre, precio);

        if (dao.altaProducto(p))
            return "confirmada";
        else
            return "alta";
    }
```

**Gráfico 4. Managed Bean**

Como podemos comprobar en la figura anterior, la clase ProductoBean está anotado como @ManagedBean.

Además incluimos la anotación @RequestScope para indicar que todos los datos almacenados tienen ámbito de petición.

La propiedad dao se considera una propiedad manejada esto indica que el valor de esta propiedad será inyectado utilizando el lenguaje de expresiones de JSF.

Vemos en la imagen siguiente como se declara el Managed Bean del dao para poder inyectarlo en la propiedad manejada.

```
@ManagedBean(name="miDao",eager=true)
@ApplicationScoped
public class ProductosDAO implements Serializable{
```

**Gráfico 5. ProductosDAO**

El método altaProducto() será un método de acción donde se ejecutará el proceso para dar de alta un producto. Devuelve un String que será el nombre de la página que se mostrará al usuario.

En la página alta.xhtml se genera un formulario donde cada dato queda almacenado en una propiedad del Managed Bean.

Al pulsar el botón se invoca al método de acción altaProducto.

```
<h:form>
    <h:outputLabel for="codigo" value="ID:" />
    <h:inputText id="codigo"
                 value="#{productoBean.id}" /> <br/>

    <h:outputLabel for="nombre" value="Nombre:" />
    <h:inputText id="nombre"
                 value="#{productoBean.nombre}" /> <br/>

    <h:outputLabel for="precio" value="Precio:" />
    <h:inputText id="precio"
                 value="#{productoBean.precio}" /> <br/>

    <h:commandButton value="Dar de alta"
                     action="#{productoBean.altaProducto}"/>
</h:form>
```

**Gráfico 6. alta.xhtml**

Vemos un fragmento de la página confirmada.xhtml

```
<h:body>
    La operacion se ha realizado con exito
</h:body>
```

**Gráfico 7. confirmada.xhtml**

## AMBITOS DE UN MANAGED BEAN

El ámbito de un Managed Bean determina su periodo de vida y si puede ser referenciado por otro bean en otro ámbito.

- @NoneScoped; No especifica ningún ámbito. Su vida es manejada por los Managed Bean que lo referencian. Un bean en este ámbito solo puede referenciar a otro bean en el mismo ámbito.
- @RequestScoped; Es el ámbito por defecto. Crea una nueva instancia por cada HTTP request. Un bean en este ámbito puede referenciar otros beans se los siguientes ámbitos: none, request, view, session, application.
- @ViewScoped; Tiene ámbito de paágina. En el momento que el usuario abandone la página el bean desaparecerá. Debe ser serializable. Un bean en este ámbito puede referenciar otros beans se los siguientes ámbitos: none, view, session, application.
- @SessionScoped; Es el ámbito de sesión. El bean ha de ser Serializable. Un bean en este ámbito puede referenciar otros beans se los siguientes ámbitos: none, session, application.
- @ApplicationScoped; Es el ámbito de aplicación. Un bean en este ámbito puede referenciar otros beans se los siguientes ámbitos: none, application.
- @CustomScoped; Es un ámbito personalizado. El bean es almacenado en un mapa y el programador puede controlar su ciclo de vida.

Todo el código de este ejemplo lo encontrareis en 2. JSF TiendaProductos.rar

# 4. FACELETS

Los facelets son archivos xhtml para representar las vistas de la aplicación.

**COMPONENTES**

Todos las etiquetas utilizadas se denominan componentes. Cada componente representa una clase que hereda de javax.faces.component.UIComponent.

A continuación vemos la jerarquía de componentes:

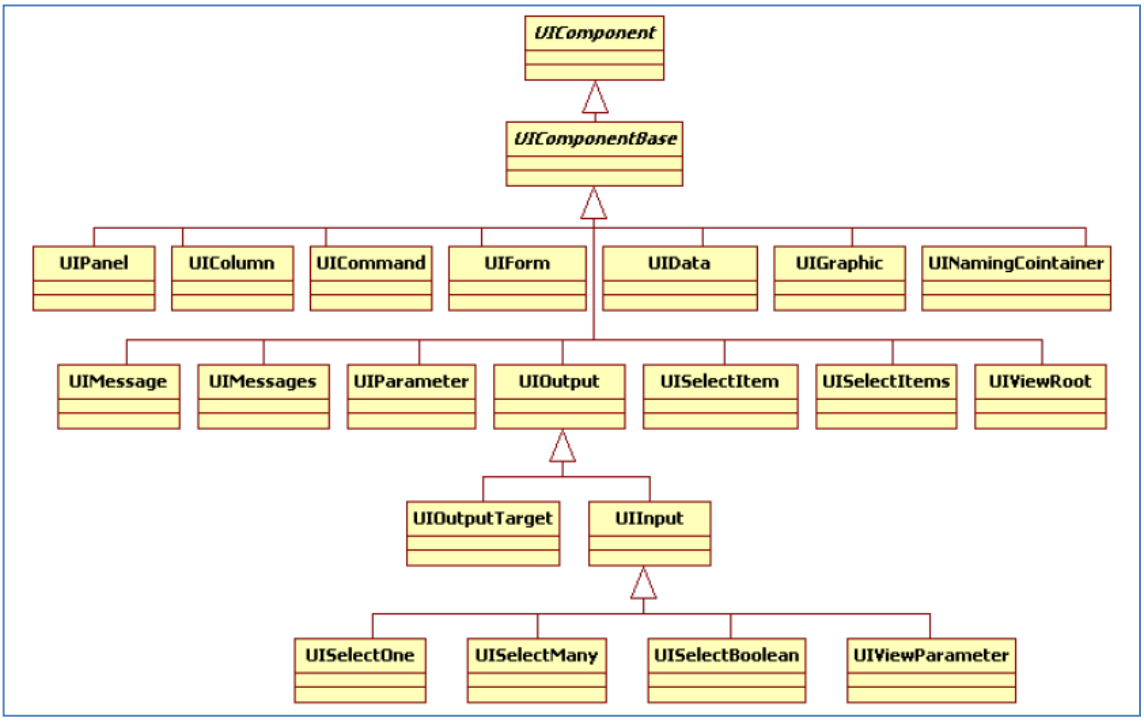


**Gráfico 8. Jerarquía de componentes**

Los componentes se agrupan en tres tipos:

- UIOutput; solo para mostrar datos. El componente más común es la etiqueta.
- UIInput; para introducir datos. El más común es el campo de texto.
- UICommand; son componentes que generan una acción. Los más comunes son los botones y los links

**RENDER KIT**

A través del Render Kit cada componente declarado en el facelet se renderizará en una etiqueta HTML que será mostrada en el navegador del cliente.

## LIBRERIAS DE ETIQUETAS

| Tag Library | URI | Prefix | Contains |
|---|---|---|---|
| Core tags | http://java.sun.com/jsf/core | f | This tag library contains tags for JSF custom actions that are independent of any particular Render Kit. |
| HTML tags | http://java.sun.com/jsf/html | h | JSF UIComponent and HTML RenderKit Renderer combinations defined in the JSF 2.0 Specification. |
| Facelets tags | http://java.sun.com/jsf/facelets | ui | Facelets templating tags |
| JSTL core tags | http://java.sun.com/jsp/jstl/core | fn | JSTL core tag library |
| JSTL functions tags | http://java.sun.com/jsp/jstl/functions | c | JSTL functions tag library |

**Gráfico 9. Librerías de etiquetas en JSF**

| Tag Categories | Tags | Functions |
|---|---|---|
| Event-handling tags | actionListener<br>phaseListener<br>setPropertyActionListener<br>valueChangeListener | Registers a event listener |
| Attribute configuration tag | attribute | Adds configurable attributes |
| Data conversion tags | converter<br>convertDateTime<br>convertNumber | Registers a data converter |
| Facet tag | facet | Register a named facet in the parent component |
| Localization tag | loadBundle | Specifies a ResourceBundle |
| Parameter substitution tag | param | Substitutes parameters into a MessageFormat instance and adds query string name-value pairs to a URL |
| Tags for representing items in a list | selectItem<br>selectItems | Represent one or a set of items in a list of a selection UI component |
| Container tag | subview | |

**Gráfico 10. Etiquetas de la librería Core**

| Tag Categories | Tags | Functions |
|---|---|---|
| Validator tags | validator<br>validateDoubleRange<br>validateLength<br>validateLongRange<br>validateRegEx<br>validateRequired | Registers a data validator |
| Output tag | verbatim | Generates a UIOutput component that gets its content from the body of this tag |
| Container tags | view<br>subview | Encloses all JSF tags on the page, or on a page included in another page for subvew |
| Facelets specific core tags | ajax | Associates ajax action to a single or group of components based on placement |
| | event | Installs ComponentSystemEventListener |
| | metadata | Registers a facet |
| | validateBean | Registers a BeanValidator instance |

**Gráfico 11. Etiquetas de la librería Core (Continuación)**

| Tag | Functions | Rendered As |
|---|---|---|
| inputTextarea | Allows a user to enter a multiline string. | An HTML <textarea> element |
| message | Displays a localized message. | An HTML  tag if styles are used |
| messages | Displays localized messages. | A set of HTML  tags if styles are used |
| outputFormat | Displays a localized message. | Plain text |
| outputLabel | Displays a nested component as a label for a specified input field. | An HTML <label> element |
| outputLink | Links to another page or location on a page without generating an action event. | An HTML <a> element |
| outputText | Displays a line of text. | Plain text |
| panelGrid | Displays a table. | An HTML <table> element with <tr> and <td> elements |
| panelGroup | Groups a set of components under one parent. | |
| selectBooleanCheckbox | Allows a user to change the value of a Boolean choice. | An HTML <input type=checkbox> element. |
| selectManyCheckbox | Displays a set of check boxes from which the user can select multiple values. | A set of HTML <input> elements of type checkbox |
| selectManyListbox | Allows a user to select multiple items from a set of items, all displayed at once. | An HTML <select> element |
| selectManyMenu | Allows a user to select multiple items from a set of items. | An HTML <select> element |
| selectOneListbox | Allows a user to select one item from a set of items, all displayed at once. | An HTML <select> element |
| selectOneMenu | Allows a user to select one item from a set of items. | An HTML <select> element |
| selectOneRadio | Allows a user to select one item from a set of items. | An HTML <input type=radio> element |

**Gráfico 12. Etiquetas de la librería HTML**

| Tag | Functions | Rendered As |
|---|---|---|
| column | Represents a column of data in a UIData component. | A column of data in an HTML table |
| commandButton | Submits a form to the application. | an `input` element, with the type being `submit`, `reset`, or `image` |
| commandLink | Links to another page or location on a page. | <a href> |
| dataTable | Represents a data wrapper. | An HTML <table> element |
| form | Represents an input form. The inner tags of the form receive the data that will be submitted with the form. | An HTML <form> element |
| graphicImage | Displays an image. | An HTML <img> element |
| inputHidden | Allows a page author to include a hidden variable in a page. | An HTML <input type=hidden> element |
| inputSecret | Allows a user to input a string without the actual string appearing in the field. | An HTML <input type=password> element |
| inputText | Allows a user to input a string. | An HTML <input type=text> element |

**Gráfico 13. Etiquetas de la librería HTML (Continuación)**

| Tag | Functions | Rendered As |
|---|---|---|
| body | Render the markup for a body element | an HTML <body> element |
| head | Render the markup for a head element | An HTML <head> element |
| button | Submit a form using a GET request | an `input` element, with the type being `submit`, `reset`, or `image` |
| link | Link to another page or location on the page | an anchor element: <a:href> |
| outputScript | Render the markup for a script elemtnf | an HTML <script> element |
| outputStylesheet | Render the markup for a link element | an HTML <link> element |

**Gráfico 14. Etiquetas de la librería HTML (Continuación)**

## LENGUAJE DE EXPRESIONES

Podemos utilizar el lenguaje de expresiones para referenciar la propiedad de un bean:

- #{usuario.nombre}
- #{cliente.direccion.ciudad}

También podemos invocar a un método del bean:

- #{usuario.validate}
- #{productoBean.altaProducto}

Unir el valor de un componente a la propiedad del bean:

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

Unir el propio componente con una propiedad del bean:

- <h:inputText binding="#{usuario.nombreCampo}" />

Esto sirve para poder manejar el componente dentro del managed bean.

Acceder a un elemento de un array

- #{array[0]}

Acceder a un atributo:

- #{requestScope["pet"]}
- #{requestScope.pet}

# 5. INTERNACIONALIZACION

Como sabemos, la internacionalización o también conocida como I18N es la utilidad que me permite mostrar todos los mensajes de mi aplicación en diferentes idiomas.

Para ello tan solo basta con extraer los mensajes estáticos de nuestros facelets y crearlos dentro de un archivo de propiedades.

Posteriormente duplicaremos este archivo en varios idiomas añadiendo la extensión de cada idioma.

Cuando un usuario emita una petición desde su navegador, en el request viaja la configuración de su navegador, incluido el idioma predeterminado.

En ese momento se buscará el archivo de mensajes de ese idioma para mostrar los textos.

Si ese idioma no estuviese disponible entonces se mostrará el idioma por defecto.

A continuación utilizamos el archivo faces-config.xml para declarar los archivos de propiedades, así como sus idiomas.

Este es el archivo de configuración estándar de JSF.

```
<faces-config version="2.0"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">

    <application>
        <resource-bundle>
            <base-name>mensajes</base-name>
            <var>msg</var>
        </resource-bundle>
        <locale-config>
            <default-locale>es</default-locale>
            <supported-locale>en</supported-locale>
            <supported-locale>de</supported-locale>
        </locale-config>
    </application>

</faces-config>
```

**Gráfico 15. Declaración de archivos de mensajes en faces-config.xml**

Los archivos de mensajes se deben colocar en el classpath de la aplicación, concretamente en la carpeta classes.

Por lo cual los ubicamos en el paquete por defecto tal como muestra la imagen siguiente.

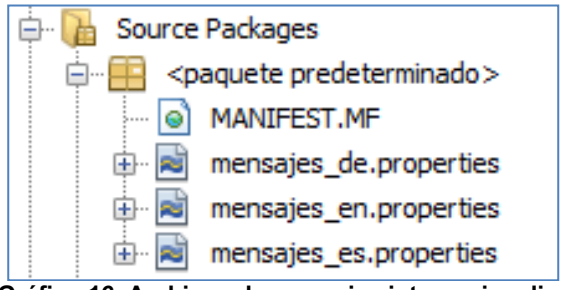


**Gráfico 16. Archivos de mensajes internacionalizados**

Para mostrar el mensaje mostramos un fragmento del facelets index.xhtml

```
<h:form>
    <h:commandLink action="alta">
        #{msg.menu_alta}
    </h:commandLink>
</h:form>
```

**Gráfico 17. Mensajes internacionalizados**

La siguiente imagen muestra la declaración de los mensajes en el archivo mensajes_es.properties.

```
alta_id=ID:
alta_nombre=Nombre:
alta_precio=Precio:

menu_alta=Dar de alta un producto
boton_alta=Dar de alta
confimacion=La operacion se ha realizado con exito
```

**Gráfico 18. Archivo de mensajes**

Todo el código de este ejemplo lo encontraréis en **3. JSF TiendaProductos Internacionalización.rar**

# 6. NAVEGACION ENTRE PAGINAS

La navegación implica un conjunto de reglas que define el flujo de páginas de la aplicación.

El flujo se determina en base a los siguientes datos:

- La pagina actual en la que nos encontramos
- El resultado de la acción de un usuario
- La salida (outcome) de un método de acción.

Una vez que la regla de navegación se ha evaluado la aplicación puede:

- Navegar a otra página
- Quedarse en la misma página.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
                      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Facelet Title</title>
    </h:head>
    <h:body>
        <h:form>
            <h:inputText id="usuario" value="#{saludoBean.nombre}" />
            <h:commandButton id="submit" action="respuesta" value="Enviar" />
        </h:form>
    </h:body>
</html>
```

**Gráfico 19. Navegación implícita**

En la imagen anterior vemos nuestro primer ejemplo con JSF donde utilizamos una navegación implícita, esto es, sin declarar reglas de navegación.

Cuando el usuario pulsa el submit, automáticamente se navega a la página respuesta.xhtml.

## DECLARAR REGLAS DE NAVEGACIÓN

En el faces-config.xml declaramos la regla de navegación para ir desde la página index.xhtml hasta la pagina alta.xhtml

```
<navigation-rule>
    <from-view-id>/index.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>add</from-outcome>
        <to-view-id>/alta.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

**Gráfico 20. Regla de navegación**

Cuando el usuario pulsa obre el link, este emite un action con valor "add" el cual se comprobará como caso de navegación <navigation-case> en la regla de navegación para la página index.xhtml.

En la imagen anterior vemos que existe un outcome con este valor con lo cual navegaremos hasta la página alta.xhtml.

```
<h:commandLink action="add">
    #{msg.menu_alta}
</h:commandLink>
```

**Gráfico 21. Invocar regla de navegación**

También podemos declarar una regla de navegación común a todas las páginas de la aplicación.

```
<navigation-rule>
    <from-view-id>/*</from-view-id>
    <navigation-case>
        <from-outcome>home</from-outcome>
        <to-view-id>/index.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

**Gráfico 22. Regla de validación común a todas las páginas**

Establecemos una regla de navegación para la página alta.xhtml. En función del valor devuelto (outcome) por el método de acción altaProducto() declarado en el bean productoBean, se navegará hacia la página de confirmada.xhtml o de nuevo a alta.xhtml.

```
<navigation-rule>
    <from-view-id>/alta.xhtml</from-view-id>
    <navigation-case>
        <from-action>#{productoBean.altaProducto}</from-action>
        <from-outcome>ok</from-outcome>
        <to-view-id>/confirmada.xhtml</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-action>#{productoBean.altaProducto}</from-action>
        <from-outcome>kao</from-outcome>
        <to-view-id>/alta.xhtml</to-view-id>
    </navigation-case>
</navigation-rule>
```

**Gráfico 23. Regla de validación para alta.xhtml**

Todo el código de este ejemplo lo encontraréis en **4.JSF TiendaProductos Navegación.rar**

# 7. CONVERSIONES

Utilizamos las conversiones para convertir los parámetros de la aplicación de un tipo String al tipo de dato adecuado.

Podemos efectuar dos tipos de conversiones:

- Implícitas; son cuando utilizamos una propiedad del Managed Bean de otro tipo diferente a String.

```
<h:inputText value="#{user.age}"/>
```

**Gráfico 24. Conversión implícita**

- Explicitas; para ello necesitamos un objeto Converter.

```
<h:outputText value="#{user.birthDay}">
  <f:converter converterId="javax.faces.DateTime"/>
</h:outputText>
```

**Gráfico 25. Conversión explicita**

En la siguiente tabla podemos ver los converters que nos proporciona JSF. Todos ellos forman parte de la librería Core.

| Converter Class | Converter ID |
|---|---|
| BigDecimalConverter | javax.faces.BigDecimal |
| BigIntegerConverter | javax.faces.BigInteger |
| BooleanConverter | javax.faces.Boolean |
| ByteConverter | javax.faces.Byte |
| CharacterConverter | javax.faces.Character |
| DateTimeConverter | javax.faces.DateTime |
| DoubleConverter | javax.faces.Double |
| EnumConverter | javax.faces.Enum |
| FloatConverter | javax.faces.Float |
| IntegerConverter | javax.faces.Integer |
| LongConverter | javax.faces.Long |
| NumberConverter | javax.faces.Number |
| ShortConverter | javax.faces.Short |

**Gráfico 26. Converters estándar**

También podemos crear nuestros propios converters. Basta con crear una clase que implemente la interface Converter y anotarla como @FacesConverter.

```
@FacesConverter(value="zipCodeConverter")
public class ZipCodeConverter implements Converter {
    ...
}
```

**Gráfico 27. Converter personalizado**

La clase debe implementar dos métodos:

- Método getAsObject(); se invocará para convertir el dato de String al tipo adecuado
- Método getAsString(); se utilizará para presentar el objeto en modo texto.

```
1   public Object getAsObject(FacesContext context, UIComponent
component,
2   String newValue) throws ConverterException {
3     String convertedValue = null;
4     if ( newValue == null ) return newValue;
5     convertedValue = newValue.trim();
6     if ( ((convertedValue.indexOf("-")) != -1) ||
7       ((convertedValue.indexOf(" ")) != -1)) {
8       char[] input = convertedValue.toCharArray();
9       StringBuffer buffer = new StringBuffer(50);
10      for ( int i = 0; i < input.length; ++i ) {
11        if ( input[i] == '-' || input[i] == ' '  ) continue;
12        else buffer.append(input[i]);
13      }
14      convertedValue = buffer.toString();
15    }
16    return convertedValue;
17  }
```

**Gráfico 28. Método getAsObject()**

```
1   public String getAsString(FacesContext context, UIComponent
component,
2   Object value) throws ConverterException {
3     String inputVal = null;
4     if ( value == null ) return null;
5     try {
6       inputVal = (String)value;
7     } catch (ClassCastException ce) {
8       throw new ConverterException(Util.getExceptionMessage(
9         Util.CONVERSION_ERROR_MESSAGE_ID));
10    }
11    char[] input = inputVal.toCharArray();
12    StringBuffer buffer = new StringBuffer(50);
13    for ( int i = 0; i < input.length; ++i ) {
14      if ( (i % 4) == 0 && i != 0)
15        if (input[i] != ' ' || input[i] != '-') buffer.append(" ");
16      buffer.append(input[i]);
17    }
18    return buffer.toString();
19  }
```

**Gráfico 29. Método getAsString()**

# 8. VALIDACIONES

Al igual que ocurre con los converters, la librería Core nos ofrece varios componentes para poder efectuar validaciones.

También aprenderemos a crear nuestros propios validadores.

En la tabla siguiente podemos ver los validadores estándar proporcionados por JSF.

| Validator Class | Tag | Function |
|---|---|---|
| DoubleRangeValidator | validateDoubleRange | Checks whether the local value of a component is within a certain range. The value must be floating-point or convertible to floating-point. |
| LengthValidator | validateLength | Checks whether the length of a component's local value is within a certain range. The value must be a java.lang.String. |
| LongRangeValidator | validateLongRange | Checks whether the local value of a component is within a certain range. The value must be any numeric type or String that can be converted to a long. |
| RegexValidator | validateRegEx | Checks whether the local value of a component is a match against a regular expression from java.util.regex package. |
| BeanValidator | validateBean | Register a bean validator |
| RequiredValidator | validateRequired | Ensures the local value is not empty on a EditableValueHolder component |

**Gráfico 30. Validadores estándar**

En la imagen siguiente vemos como utilizar estos validadores para comprobar los datos introducidos por el usuario en el formulario alta.xhtml.

```
<h:form>
    <h:outputFormat  value="#{msg.alta_id}" />
    <h:inputText id="codigo"
                 value="#{productoBean.id}" >
        <f:validateLongRange minimum="1" maximum="100" />
    </h:inputText>
    <h:message for="codigo" /><br/>

    <h:outputFormat value="#{msg.alta_nombre}" />
    <h:inputText id="nombre"
                 value="#{productoBean.nombre}" >
        <f:validateLength minimum="1" maximum="20" />
    </h:inputText>
    <h:message for="nombre" /><br/>

    <h:outputFormat value="#{msg.alta_precio}" />
    <h:inputText id="precio"
                 value="#{productoBean.precio}" >
        <f:validateDoubleRange minimum="3" maximum="300" />
    </h:inputText>
    <h:message for="precio" /><br/>

    <h:commandButton value="#{msg.boton_alta}"
                     action="#{productoBean.altaProducto}"/>
</h:form>
```

**Gráfico 31. Validación de formulario**

En el faces-config.xml declaramos el archivo que usaremos para los errores de validación.

```
<message-bundle>errores</message-bundle>
```

**Gráfico 32. Entrada del archivo de errores**

```
javax.faces.component.UIInput.REQUIRED = Error de validacion: Dato requerido

javax.faces.validator.DoubleRangeValidator.NOT_IN_RANGE = Error de validacion: El valor debe estar comprendido entre {0} y {1}

javax.faces.validator.LongRangeValidator.NOT_IN_RANGE = Error de validacion: El valor debe estar comprendido entre {0} y {1}

javax.faces.validator.LengthValidator.MAXIMUN = Error de validacion: Debe contener como maximo {0} caracteres

javax.faces.validator.LengthValidator.MINIMUN = Error de validacion: Debe contener como minimo {0} caracteres
```

**Gráfico 33. Archivo de mensajes de error**

Todo el código de este ejemplo lo encontraréis en **5. JSF TiendaProductos Validación.rar**

Para crear nuestro validador personalizado debemos crear una clase que implemente la interface Validator con su método validate() y anotarla como @FacesValidator

```
@FacesValidator(value="FormalValidator")
public class FormalValidator implements Validator {
    ...
}
```

**Gráfico 34. Validador personalizado**

```
public void validate(FacesContext context,
UIComponent component, Object toValidate) {
  boolean valid = false;
  String value = null;
  if((context == null) || (component == null)) {
    throw new NullPointerException();
  }
  if(!(component instanceof UIOutput)) {
    return;
  }
  // Continued...
}
```

**Gráfico 35. Método validate()**

A continuación vemos como podemos utilizar el validador creado.

```
<h:inputText id="zip" value="#{CustomerBean.zip}" size="10" ... >
  <f:validator validatorId="FormatValidator" />
  ...
</h:inputText>
```

**Gráfico 36. Utilización del validador personalizado**

# 9. INTEGRACION CON AJAX

JSF 2.0. proporciona una fácil integración con AJAX a través de la etiqueta <f:ajax> perteneciente a la librería Core.

Esta etiqueta presenta cuatro atributos:

- render; Son los elementos que se refrescarán en la página.
- execute; El elemento que se envía al servidor para su proceso.
- event; Evento que provoca la petición AJAX
- onevent; Función JavaScript que se ejecutará cuando el evento sea disparado.

```
<h:commandButton … action="…">
    <f:ajax render="id1 id2" execute="id3 id4"
            event="blah" onevent="javaScriptHandler"/>
</h:commandButton>
```

**Gráfico 37. Ejemplo AJAX**

Esta etiqueta nos permite utilizar las siguientes anotaciones:

- @this. Hace referencia al elemento donde se incluye la etiqueta f:ajax. Es el valor por defecto.
- @form. El formulario donde se incluye la etiqueta f:ajax
- @none. No se envía ningún dato al servidor.
- @all. Hace referencia a todos los componentes de la página.
-

```
<h:form />
    …
    <f:ajax render="elementId" execute="@form" />
    …
</h:form>
```

**Gráfico 38. Ejemplo de @form**

# 10. INTEGRACION CON SPRING

JSF 2.0. también puede ser integrado con Spring 3.0.

A continuación vemos los pasos a seguir sobre nuestro ejemplo.

Lo primero que debemos hacer es declarar en el web.xml el listener de contexto y el parámetro inicial para indicar la ubicación del spring.xml.

```
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring.xml</param-value>
</context-param>
```

**Gráfico 39. Declaración del listener de contexto de Spring**

En el faces-config.xml únicamente declaramos un elemento de tipo Resolver con el fin de poder resolver las peticiones conjuntamente con JSF y Spring.

```
<application>
    <el-resolver>
        org.springframework.web.jsf.el.SpringBeanFacesELResolver
    </el-resolver>
</application>
```

**Gráfico 40. faces-config.xml**

En el spring.xml declararemos como beans los Managed Beans o clases que sean necesarios.

```
<bean id="dao" class="app.persistencia.ProductoDAO">
    <property name="sf" ref="sessionFactory" />
</bean>

<bean id="servicioProductos" class="app.negocio.ServicioProductos">
    <property name="dao" ref="dao" />
</bean>
```

**Gráfico 41. spring.xml**

Todo el código de este ejemplo lo encontrareis en **Spring JSF Hibernate Productos.rar**

# INDICE DE GRÁFICOS