Curso de Java Server Faces 2 con Hibernate 3

Manual del alumno

<JSF 2>

<Hibernate 3>

Ing. Cedric Simon – Tel: 2268 0974 – Cel: 8888 2387 – Email: cedric@solucionjava.com – Web: www.solucionjava.com

SolucionJava.com

Docente: Cédric Simon

Versión 1.1

Índice

1 Introducción al curso

1.1 Objetivo de este curso

En este curso vamos a aprender el lenguaje JSF que nos permitirá crear páginas web dinámicas.

1.2 Manual del alumno

Este manual del alumno es una ayuda para el alumno, para tenga un recuerdo del curso. Este manual contiene un resumen de las materias que se van a estudiar durante el curso, pero el alumno debería de tomar notas personales para completas este manual.

1.3 Ejercicios prácticos

Para captar mejor la teoría, se harán muchos ejercicios con los alumnos, para probar la teoría y verificar la integración de la materia.

También, el alumno podrá copiar sus códigos en un disquete al fin del curso para llevarse, con fin de seguir la práctica en su hogar.

1.4 Requisitos para atender a este curso

Una buen conocimiento de los lenguajes Java, JSP, HTML, y Javascript es requerida para seguir este curso. La creación y el manejo de objetos Java así como el JSP están considerada cómo asimilado antes de empezar este curso.

Si el alumno tiene dificultades en un u otro capitulo, el debe sentirse libre de pedir explicaciones adicionales al profesor.

Pero si aparece que el alumno no posee los requisitos mínimos para este curso, por respeto a los otros alumnos que ya poseen esta materia, el alumno podría ser traslado para otro curso en el futuro, cuando el cumplirá con los requisitos.

1.5 Soporte después del curso

Si tienes preguntas sobre la materia del curso en tus ejercicios prácticos, puedes escribir tus preguntas a cedric@solucionjava.com.

Para informaciones sobre otros cursos, visita el sitio web www.solucionjava.com.

2 Introducción a Java Server Faces

2.1 ¿Que es JSF?

JavaServer Faces (JSF) es un tecnología y framework para aplicaciones Java basadas en web que simplifica el desarrollo de interfaces de usuario en aplicaciones Java EE.

JSF usa JavaServer Pages (JSP) como la tecnología que permite hacer el despliegue de las páginas, pero también se puede acomodar a otras tecnologías como XUL.

JSF incluye:

- Un conjunto de APIs para representar componentes de una interfaz de usuario y administrar su estado, manejar eventos, validar entrada, definir un esquema de navegación de las páginas y dar soporte para internacionalización y accesibilidad.
- Un conjunto por defecto de componentes para la interfaz de usuario.
- Dos bibliotecas de etiquetas personalizadas para JavaServer Pages que permiten expresar una interfaz JavaServer Faces dentro de una página JSP.
- Un modelo de eventos en el lado del servidor.
- Administración de estados.
- Beans administrados.

La especificación de JSF fue desarrollada por la Java Community Process

Versiones de JSF:

- JSF 1.0 (11-03-2004) - lanzamiento inicial de las especificaciones de JSF.
- JSF 1.1 (27-05-2004) - lanzamiento que solucionaba errores. Sin cambios en las especificaciones ni en el renderkit de HTML.
- JSF 1.2 (11-05-2006) - lanzamiento con mejoras y corrección de errores.
- JSF 2.0 (12-08-2009) - último lanzamiento.

Las principales implementaciones de JSF son:

- JSF Reference Implementation de Sun Microsystems.
- MyFaces proyecto de Apache Software Foundation.
- Rich Faces, de Jboss. Trae componentes adicionales para crear aplicaciones más “ricas”
- ICEfaces Contiene diversos componentes para interfaces de usuarios más enriquecidas, tales como editores de texto enriquecidos, reproductores de multimedia, entre otros.
- jQuery4jsf Contiene diversos componentes sobre la base de uno de los más populares framework javascript jQuery.

2.2 Servidor y herramientas utilizados

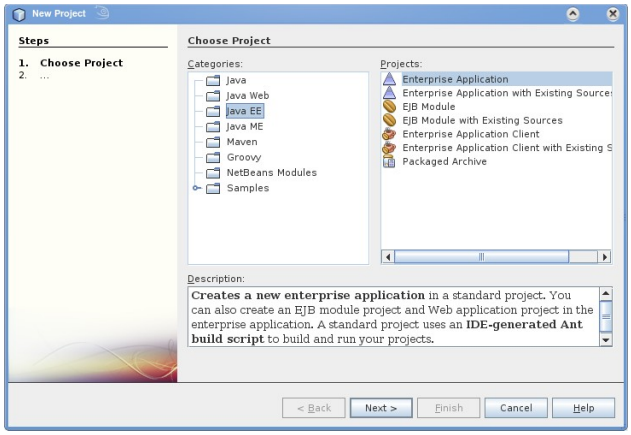
Java Server Faces 2.0 es una tecnología nueva, y necesita las últimas versiones de las herramientas de desarrollo y servidores web para poder usarla.

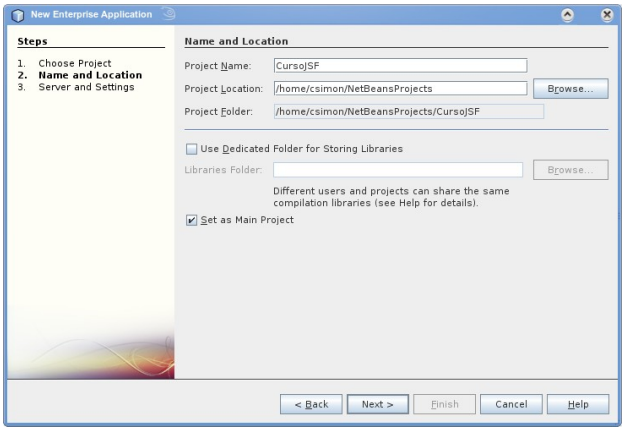
A nivel de herramientas, las más utilizadas son Eclipse y NetBeans. En el curso usaremos NetBeans versión 6.8 o arriba.

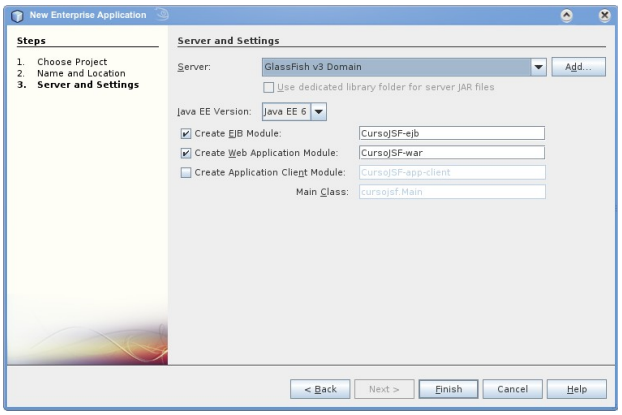
A nivel de servidores web, servidores como Apache Tomcat 6+, Jboss 5+, o GlassFish 3 soportan JSF 2.0.

2.3 Crear una aplicación en NetBeans

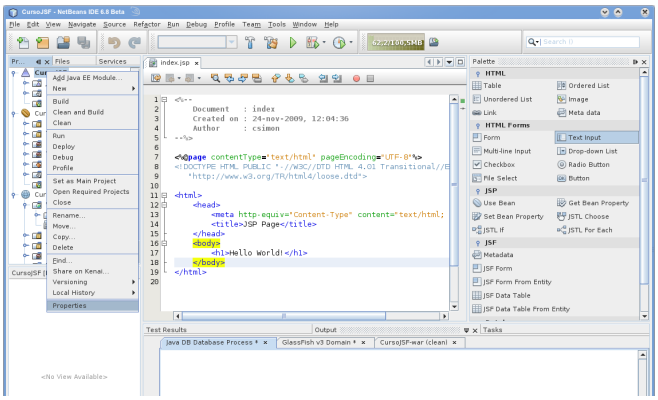
Se requiere NetBeans 6.8 o arriba. Escoge de crear un nuevo proyecto.

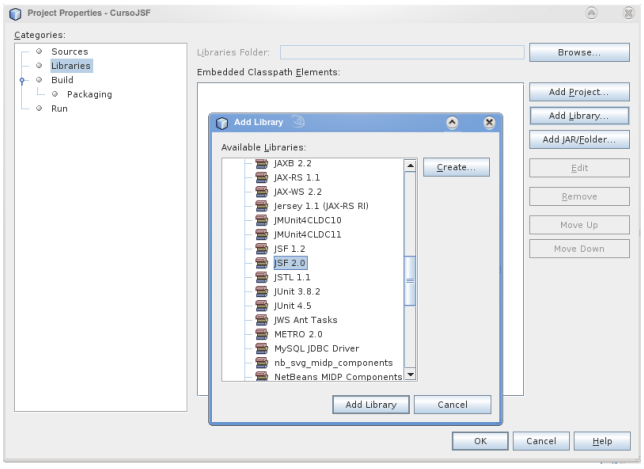




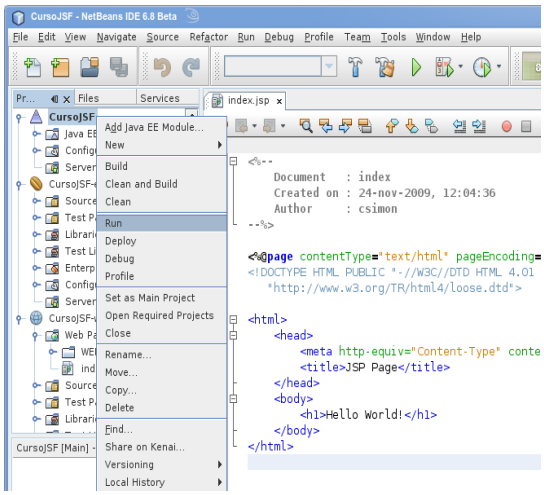


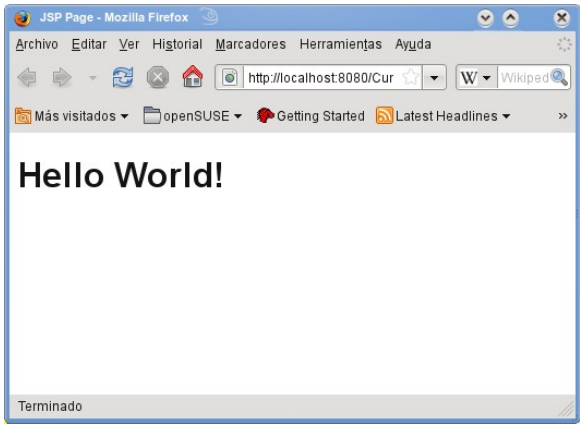
Agregamos la librería JSF 2.0 al servidor.





Iniciamos el servidor web y desplegamos la aplicación.
Miramos el resultado en http://localhost:8080/CursoJSF-war



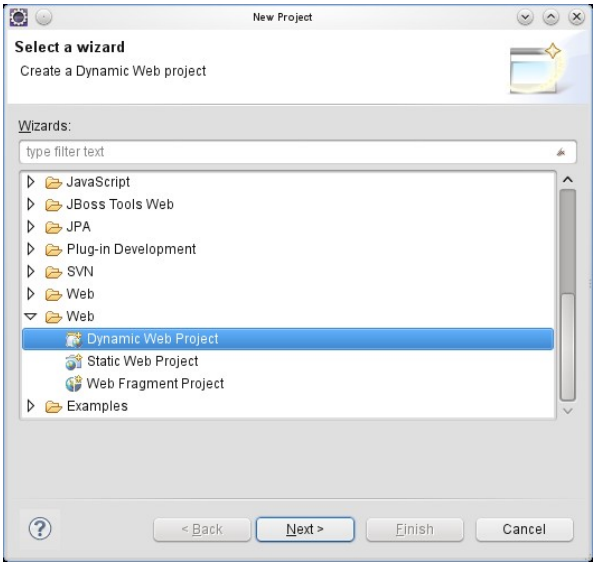


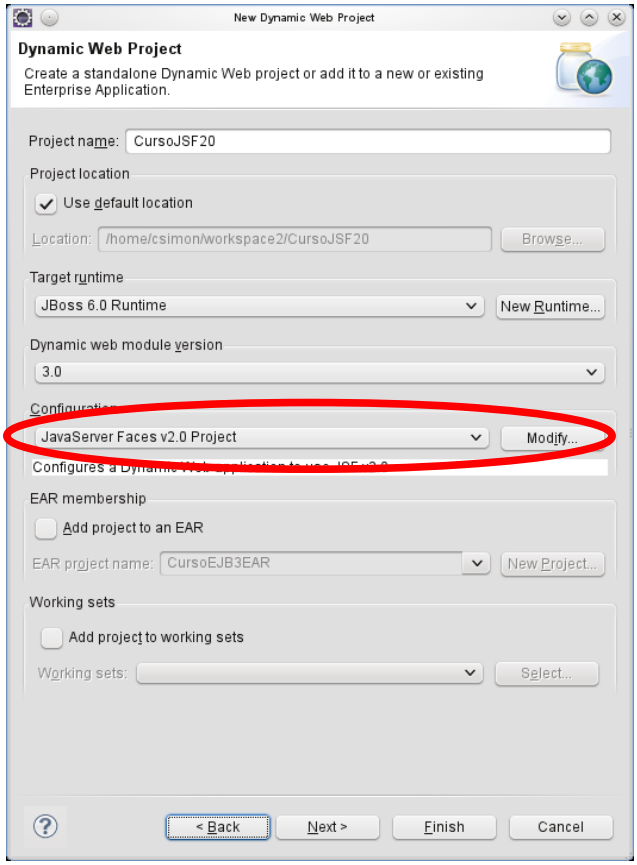
Por el momento solo estamos mirando a una página JSP, nada de JSF.

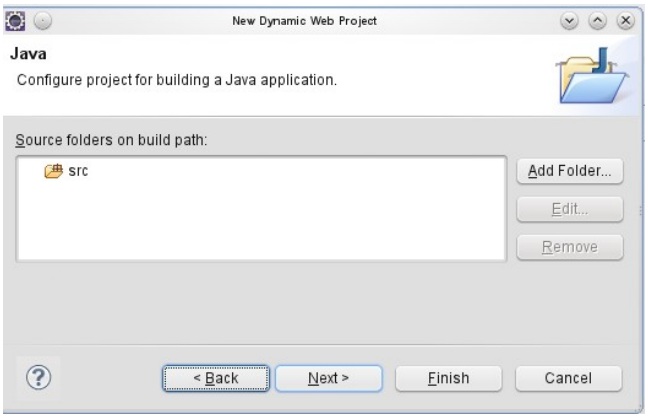
2.4 Crear una aplicación en Eclipse

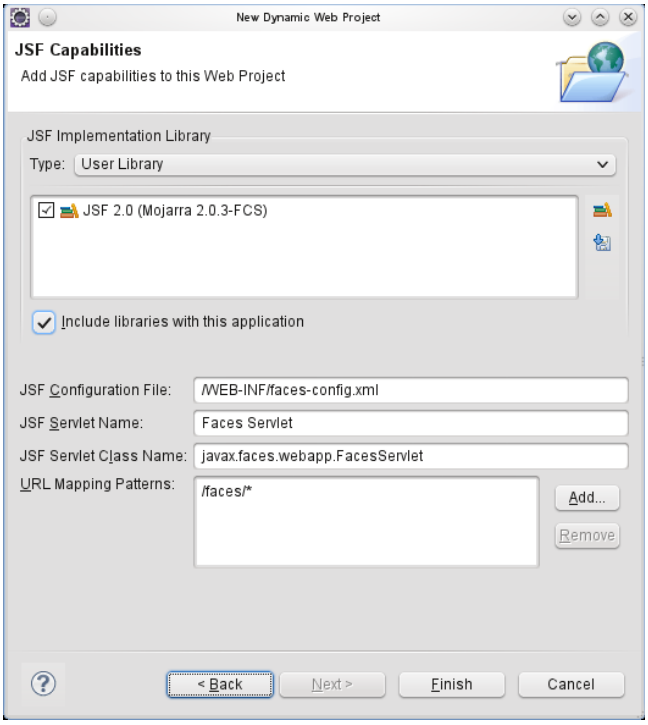
Se requiere Eclipse 3.6SR1 o arriba. Para utilizar Jboss 6, se requiere en plugin de Jboss Tools.

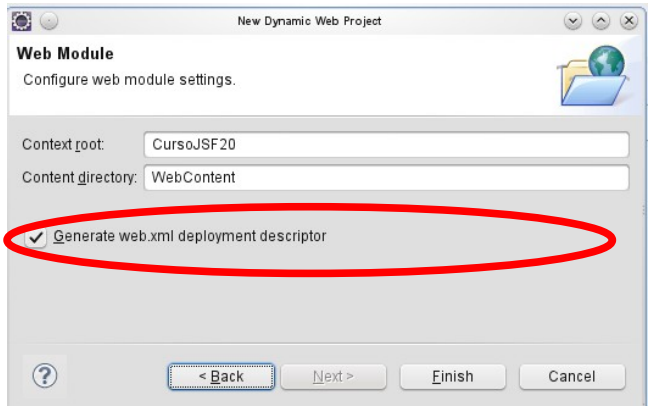
Escogemos de crear un nuevo proyecto Web Dinámico.











Creamos una página index.jsp debajo de WebContent, e iniciamos el servidor web desplegando la aplicación.

Miramos el resultado en http://localhost:8080/CursoJSF20

2.5 ***Primera página JSF***

Las páginas JSF son páginas JSP con unas librerías Taglib adicionales.

Ademas, las páginas JSP no se llaman directamente, pero se llaman a través del motor JSF. Para llamar a la página JSF, basta con agregar /faces/ antes el nombre de la página.

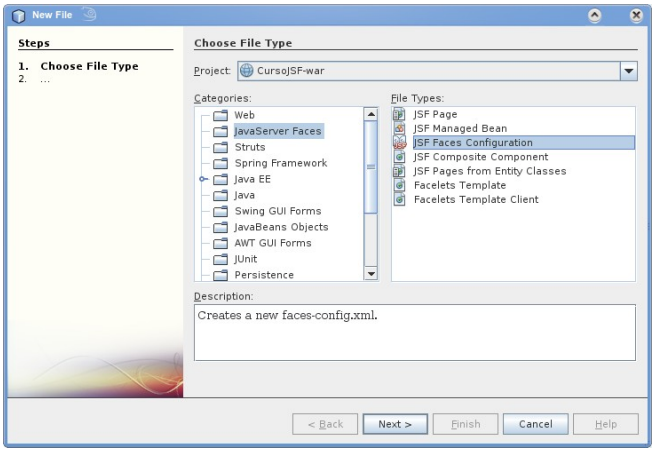
http://localhost:8080/CursoJSF-war/faces/index.jsp

Como lo vemos, una página JSP (sin código JSF) puede ser llamada sin problema.

La configuración del reenvío se puede personalizar (por carpeta y/o por extensión) en web.xml:

```
<servlet>
     <servlet-name>Faces Servlet</servlet-name>
     <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
     <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
     <servlet-name>Faces Servlet</servlet-name>
     <url-pattern>/faces/*</url-pattern>
 </servlet-mapping>
```


Cuando creamos una nueva página JSF, podemos escoger entre dos tipos de sintaxis: Facelets o JSP. Facelets usa un formato XML (XHTML), y JSP usa... el formato JSP.



El JSF usa páginas JSP o XHTML, un archivo de configuración XML (faces-config.xml), y Java POJO's.

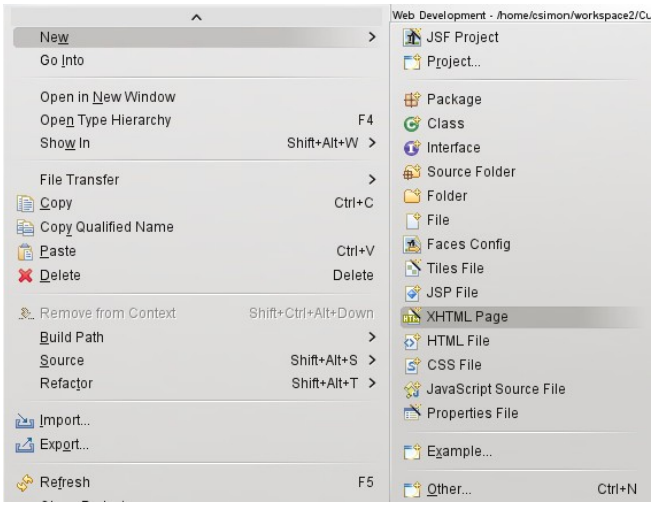
A partir de JSF 2.0, el formato XML es el estándar.

Agregamos el archivo de configuración.

Ahora que tenemos todo listo, creamos nuestra primera página JSF, usando la opción JSP:

```
<%@page contentType="text/html" pageEncoding="UTF-
8"%>

<%@taglib prefix="f"
uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h"
uri="http://java.sun.com/jsf/html"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
<f:view>
    <html>
        <head>
            <meta http-equiv="Content-Type"
content="text/html; charset=UTF-8"/>
            <title>Mi primera pagina JSP</title>
        </head>
        <body>
            <h1><h:outputText value="Hola
alumno!"/></h1>
        </body>
    </html>
</f:view>
```



El código JSF en la página JSP se debe encontrar entre los tags <f:view> y </f:view>.

Como podemos ver, se incluyen las dos librerías Taglib, que se usan luego, con las etiquetas <f: > y <h: >.

Si usamos el formato XHTML, todo el contenido se considera JSF.

```
<?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:body>
            <h1>Hello alumno !!!</h1>
    </h:body>
</html>
```

La biblioteca <h: > contiene equivalentes del HTML, ya sea formularios, botones, cajas de texto, imágenes, etc...

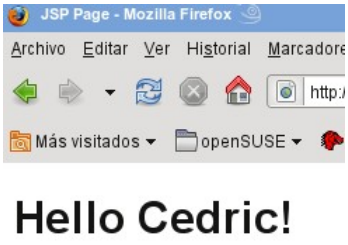
En el futuro usaremos los Facelets y no mas el JSP.

2.6 Recuperando informaciones del código Java

Vamos a crear una clase Java (JSF Backed Bean) Hello y llamarla desde nuestra página JSF.

```
package curso;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;

@ManagedBean(name="Hello")
@RequestScoped
public class Hello {
    public Hello() {    }
    public String getNombre(){
        return "Cedric";
    }
}
```



Y modificamos la página index.xhtml:

```
<h1>Hello #{Hello.getNombre()} !!!</h1>
```

En el futuro usaremos el formato XHTML, y no incluiré las etiquetas <xml>, <DOCTYPE> ni <html>, que serán los siguientes:

```
<?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:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html"
         xmlns:ui="http://java.sun.com/jsf/facelets"
      >
```


3 Aplicaciones JSF

3.1 ¿Qué es una aplicación JavaServer Faces?

En su mayor parte, una aplicación JavaServer Faces es como cualquier otra aplicación Java Web. Una aplicación típica de JavaServer Faces incluye las siguientes piezas:

- Un conjunto de páginas web, en la que los componentes de interfaz de usuario son establecidos.
- Un conjunto de bibliotecas de etiquetas proporcionar etiquetas para añadir componentes de interfaz de usuario a la página web.
- Un conjunto de Backed Bean, que son JavaBeans componentes que definen las propiedades y funciones de los componentes de la interfaz de usuario en una página.
- Opcionalmente, uno o más ficheros de configuración de la aplicación de recursos (como faces-config.xmlfile), que definen las normas de navegación de la página y configura los beans y otros objetos personalizados, tales como componentes personalizados.
- Un descriptor de despliegue (un archivo web.xml).
- Es posible que un conjunto de objetos personalizados creados por el desarrollador de aplicaciones. Estos objetos pueden incluir componentes personalizados, validadores, convertidores, o los oyentes.
- Un conjunto de etiquetas personalizadas para representar los objetos personalizados en la página.

3.2 Modelo de componentes de interfaz de usuario

JavaServer Faces componentes de interfaz de usuario (UI) son los bloques de construcción de vista de JavaServer Faces.

JavaServer Faces componentes de interfaz de usuario son elementos configurables, reutilizables que componen la interfaz de usuario de aplicaciones JavaServer Faces. Un componente puede ser simple, como un botón, o pueden ser compuestos, tales como una tabla, compuesta de múltiples componentes.

La tecnología JavaServer Faces proporciona una rica arquitectura de componentes flexibles que incluye lo siguiente:

- Un conjunto de clases UIComponent para especificar el estado y el comportamiento de los componentes de la interfaz de usuario
- Un modelo de representación que define el modo de hacer los componentes de varias maneras
- Un evento de escucha y el modelo que define cómo manejar los eventos de los componentes
- Un modelo de conversión que define cómo registrar los convertidores de datos en un componente
- Un modelo de validación que define cómo registrar validadores en un componente

Esta sección describe brevemente cada una de estas piezas de la arquitectura de componentes.

3.3 Las clases de componentes de interfaz de usuario

La tecnología JavaServer Faces proporciona un conjunto de clases de componentes de interfaz de usuario y de comportamiento asociados a las interfaces que especifican todas las funcionalidad de los componentes de interfaz de usuario, como componente de participación del estado, mantener una referencia a los objetos, y un evento de conducción y manejo de la prestación de un conjunto de componentes estándar.

Las clases de componentes son totalmente extensible, permitiendo a los escritores de componentes para crear sus propios componentes personalizados. Creación de componentes personalizados es un tema avanzado que veremos más tarde.

La clase base abstracta para todos los componentes de interfaz de usuario es javax.faces.component.UIComponent.

Las clases de componentes de interfaz de usuario amplian la clase UIComponentBase, (una subclase de la clase UIComponent), que define el estado por defecto y el comportamiento de un componente de interfaz de usuario. El siguiente conjunto de clases de componentes de interfaz de usuario se incluye con la tecnología JavaServer Faces:

- UIColumn: Representa una sola columna de datos en un componente UIData.
- UICommand: Representa un control que inicia acciones cuando se activa.
- UIData: Representa un enlace de datos a una colección de datos representados por una instancia dataModel.
- UIForm: Encapsula un grupo de controles que enviar datos a la aplicación. Este componente es análogo a la etiqueta de formulario en HTML.
- UIGraphic: Muestra una imagen.
- UIInput: Toma de entrada de datos de un usuario. Esta clase es una subclase de UIOutput.
- UIMessage: Muestra un mensaje de error traducidos.
- UIMessages: Muestra un conjunto de mensajes de error traducidos.
- UIOutcomeTarget: Muestra un hipervínculo en la forma de un vínculo o un botón.
- UIOutput: Muestra la salida de datos en una página.
- UIPanel: administra el diseño de sus componentes niño.
- UIParameter: Representa los parámetros de sustitución.
- UISelectBoolean: Permite al usuario establecer un valor booleano en un control de selección y anulación de ella. Esta clase es una subclase de la clase UIInput.
- UISelectItem: Representa un solo elemento en un conjunto de elementos.
- UISelectItems: Representa todo un conjunto de elementos.
- UISelectMany: Permite al usuario seleccionar varios elementos de un grupo de elementos. Esta clase es una subclase de la clase UIInput.
- UISelectOne: Permite al usuario seleccionar un elemento de un grupo de elementos. Esta clase es una subclase de la clase UIInput.
- UIViewParameter: Representa los parámetros de consulta en una solicitud. Esta clase es una subclase de la clase UIInput.
- UIViewRoot: Representa la raíz del árbol de componentes.

Además de ampliar UIComponentBase, las clases de componente también aplicar una o más interfaces de comportamiento, cada uno de los cuales define cierto comportamiento de un conjunto de componentes cuyas clases implementan la interfaz.

Estas interfaces de comportamiento son las siguientes:

- ActionSource: Indica que el componente puede desencadenar un evento de acción. Esta interfaz está diseñado para utilizarse con componentes basados en la tecnología JavaServer Faces 1.1_01 y versiones anteriores.
- ActionSource2: Extiende ActionSource, y por lo tanto proporciona la misma funcionalidad. Sin embargo, se permite a los componentes a utilizar el EL unificado cuando se hace referencia a los métodos para controlar los eventos de acción.
- EditableValueHolder: Extiende ValueHolder y especifica las características adicionales para los componentes modificable, como la validación y emitir los eventos de cambio de valor.
- NamingContainer: los mandatos que cada componente de raíz en este componente tiene una identificación única.
- StateHolder: Indica que un componente ha estado que deben guardar entre las solicitudes.

- ValueHolder: Indica que el componente mantiene un valor local, así como la posibilidad de acceder a los datos en el nivel de modelo.
- SystemEventListenerHolder: Mantiene una lista de casos SystemEventListener para cada tipo de SystemEvent definido por esa clase.
- ClientBehaviorHolder: añade la capacidad de unir los casos ClientBehavior como una secuencia de comandos reutilizables.

Cada etiqueta personalizada definida en el estándar HTML hacer kit se compone de la funcionalidad de los componentes (definidos en la clase UIComponent) y la prestación de atributos (definidos por la clase Renderer).

Lista de etiquetas UI Component

Tag	**Funciones**	**Rendered As**	**Apariencia**
column	Representa una columna de datos en un componente UIData	A column of data in an HTML table	Una columna de una tabla
commandButton	Envía un formulario para la solicitud	An HTML <input type=type> element, where the type value can be submit, reset, or image	Un botón
commandLink	Enlaces a otra página o ubicación en una página	An HTML <a href> element	Un hipervínculo
dataTable	Representa un contenedor de datos	An HTML <table> element	Una tabla que se pueden actualizar de forma dinámica
form	Representa una forma de entrada (etiquetas internas de la forma recibir los datos que se presentará con el formulario)	An HTML <form> element	No aparece
graphicImage	Muestra una imagen	An HTML <img> element	Una imagen
inputHidden	Permite a un autor de la página incluir una variable oculta en una página	An HTML <input type=hidden> element	No aparece
inputSecret	Permite al usuario introducir una cadena sin la cadena aparece en claro en el campo	An HTML <input type=password> element	Un campo de texto, que muestra una fila de estrellas en lugar de la cadena real que haya entrado
inputText	Permite al usuario introducir una cadena	An HTML <input type=text> element	Un campo de texto
inputTextarea	Permite a un usuario introducir una cadena de varias líneas	An HTML <textarea> element	Un campo de texto de varias lineas
message	Muestra un mensaje localizado	An HTML tag if styles are used	Una cadena de texto
messages	Muestra los mensajes localizados	A set of HTML tags if styles are used	Una cadena de texto
outputFormat	Muestra un mensaje localizado	Plain text	Texto sin formato
outputLabel	Muestra un componente anidado como una etiqueta para un campo de entrada especificado	An HTML <label> element	Texto sin formato
outputLink	Enlaces a otra página o ubicación en una página sin	An HTML <a> element	Un hipervínculo

	que se genere un evento de acción		
outputText	Muestra una línea de texto	Plain text	Texto sin formato
panelGrid	Muestra una tabla	An HTML <table> element with <tr> and <td> elements	Una tabla
panelGroup	Grupos de un conjunto de componentes en uno de los padres	A HTML or element	Una fila de una tabla de
selectBooleanCheckbox	Permite a un usuario cambiar el valor de una opción boleana	An HTML <input type=checkbox> element.	Una casilla de verificación
selectItem	Representa un elemento en una lista de elementos en un componente UISelectOne	An HTML <option> element	No aparece
selectItems	Representa una lista de elementos en un componente UISelectOne	A list of HTML <option> elements	No aparece
selectManyCheckbox	Muestra un conjunto de casillas de verificación de que el usuario puede seleccionar varios valores	A set of HTML <input> elements of type checkbox	Un conjunto de casillas de verificación
selectManyListbox	Permite al usuario seleccionar varios elementos de un conjunto de elementos, todos muestran a la vez	An HTML <select> element	Un cuadro de lista
selectManyMenu	Permite al usuario seleccionar varios elementos de un conjunto de elementos	An HTML <select> element	Un cuadro combinado desplazable
selectOneListbox	Permite a un usuario para seleccionar un elemento de un conjunto de elementos, todos muestran a la vez	An HTML <select> element	Un cuadro de lista
selectOneMenu	Permite a un usuario para seleccionar un elemento de un conjunto de elementos	An HTML <select> element	Un cuadro combinado desplazable
selectOneRadio	Permite a un usuario para seleccionar un elemento de un conjunto de elementos	An HTML <input type=radio> element	Un conjunto de botones de radio

3.4 Conversión de Modelo

Una aplicación JavaServer Faces, opcionalmente, se puede asociar con un componente de servidor de datos de objetos secundarios. Este objeto es un componente JavaBeans, llamado BackedBean. Una aplicación obtiene y establece el objeto de datos para un componente llamando a las propiedades de objeto apropiado para ese componente.

Cuando un componente está asociado a un objeto, la aplicación tiene dos puntos de vista de los datos del componente:

- El punto de vista del modelo, en el que se representan los datos como tipos de datos, tales como int o long.
- La vista de presentación, en el que se representan los datos de una manera que puede ser leído o modificado por el usuario. Por ejemplo, un java.util.Date puede ser representada como una cadena de texto en el formato mm / dd / aa o como un conjunto de tres cadenas de texto.

La implementación de JavaServer Faces convierte automáticamente datos de los componentes entre estos dos puntos de vista cuando la propiedad de bean asociados con el componente de uno de los tipos soportados por los datos del componente.

Por ejemplo, si un componente de UISelectBoolean se asocia con una propiedad de bean de java.lang.Boolean tipo, la implementación JavaServer Faces se convertirá automáticamente los datos del componente de la cadena en Boolean. Además, algunos datos de los componentes deben estar vinculadas a las propiedades de un tipo particular. Por ejemplo, un componente UISelectBoolean debe estar enlazado a una propiedad de tipo boolean o java.lang.Boolean.

A veces puede que desee convertir los datos de un componente a un tipo distinto de un tipo estándar, o puede que desee convertir el formato de los datos. Para facilitar esto, la tecnología JavaServer Faces le permite registrar la aplicación Converter en componentes UIOutput y componentes cuyas clases UIOutput subclase. Si se registra la aplicación Converter en un componente, la aplicación Converter convierte los datos del componente entre los dos puntos de vista.

Puede utilizar los convertidores estándar suministrados con la implementación JavaServer Faces o crear su propio convertidor personalizado. La creación de convertidor personalizado es un tema avanzado que se verá más adelante.

3.5 Evento y el oyente de modelo

JavaServer Faces 2.0 define tres tipos de eventos: los eventos de aplicación, los eventos del sistema y los datos de eventos de modelo.

Eventos de aplicación están vinculados a una aplicación en particular y son generados por un UIComponent. Ellos representan el estándar de eventos disponibles en versiones anteriores de la tecnología JavaServer Faces.

Un objeto de evento identifica el componente que generó el evento y almacena información sobre el evento. Para ser notificado de un evento, una aplicación debe proporcionar una implementación de la clase de escucha y debe registrarlo en el componente que genera el evento. Cuando el usuario activa un componente, por ejemplo, hacer clic en un botón, se dispara un evento. Esto hace que la implementación JavaServer Faces para invocar el método de escucha que procesa el evento.

JavaServer Faces soporta dos tipos de eventos de aplicación: eventos de acción y eventos de cambio de valor.

Un evento de acción (ActionEvent clase) se produce cuando el usuario activa un componente que implemente ActionSource. Estos componentes incluyen botones y los hipervínculos.

Un evento de cambio de valor (ValueChangeEvent clase) se produce cuando el usuario cambia el valor de un componente representada por UIInput o uno de sus subclases. Un ejemplo es la selección de una casilla de verificación, una acción que resulta en el valor del componente está cambiando a true. Los tipos de componentes que puede generar este tipo de eventos son los UIInput, UISelectOne, UISelectMany, y los componentes de UISelectBoolean. Valor eventos de cambio son despedidos sólo si no se detectaron errores de validación.

Los sucesos del sistema son generados por un objeto en lugar de un UIComponent. Que se generan durante la ejecución de una aplicación en tiempos predefinidos. Son aplicables a toda la aplicación en lugar de a un componente específico.

Un modelo de datos de eventos se produce cuando se selecciona una nueva fila de un componente de UIData.

Los sucesos del sistema y los datos de eventos de modelo son temas avanzados que se mirarán más tarde.

1.1 Validación

La tecnología JavaServer Faces soporta un mecanismo para la validación de los datos locales de los componentes modificables (como los campos de texto). Esta validación se produce antes de que el modelo de datos correspondiente se actualiza para que coincida con el valor local.

Al igual que el modelo de conversión, el modelo de validación define un conjunto de clases estándar para la realización de comprobaciones de validación de datos comunes. La etiqueta de JavaServer Faces núcleo biblioteca también define un conjunto de etiquetas que corresponden a las implementaciones estándar de Validator.

La mayoría de las etiquetas tienen un conjunto de atributos para configurar las propiedades del validador, tales como los valores máximo y mínimo permitidos para los datos del componente. Los registros de autor es el validador en un componente por la etiqueta de anidación del validador dentro de la etiqueta del componente.

3.6 Modelo de navegación

El modelo de navegación JavaServer Faces hace que sea fácil de manejar cualquier procesamiento adicional que se necesita para elegir la secuencia en la que se cargan las páginas.

En la tecnología JavaServer Faces, la navegación es un conjunto de reglas para la elección de la página siguiente o la vista que se mostrará después de una acción de aplicación, como cuando un botón o hipervínculo se hace clic.

Estas normas se declaran en cero o más recursos de configuración de la aplicación, tales como <faces-config.xml>, utilizando un conjunto de elementos XML. La estructura por defecto de una regla de navegación es el siguiente:

```
<navigation-rule>
   <description>
   </description>
   <from-view-id></from-view-id>
   <navigation-case>
      <from-action></from-action>
      <from-outcome></from-outcome>
      <if></if>
      <to-view-id></to-view-id>
   </navigation-case>
</navigation-rule>
```

En JavaServer Faces 2.0, la navegación puede ser implícito o definidos por el usuario. Las reglas de navegación implícitas entran en juego cuando las normas de navegación no están disponibles en un archivo de configuración de la aplicación de recursos.

3.6.1 Configuración de reglas de navegación

Como se explica en el modelo de navegación, la navegación es un conjunto de reglas para la elección de la siguiente página que se muestra después de un botón o un componente de hipervínculo se hace clic.

Las reglas de navegación se definen en el expediente de solicitud de recursos de configuración.

Cada regla de navegación especifica cómo navegar de una página a un conjunto de otras páginas. La implementación de JavaServer Faces elige la regla de navegación adecuado de acuerdo a la página que se muestra actualmente.

Después de la regla de navegación adecuado es seleccionado, la elección de que para acceder a la página siguiente de la página actual depende de dos factores:

- El método de acción que se invoca cuando el componente se ha hecho clic
- El resultado lógico que se hace referencia por el componente en la etiqueta , o fue devuelto por el método de acción

El resultado puede ser cualquier cosa que el desarrollador decide, pero la tabla aquí van algunos resultados de uso común en aplicaciones web.

Final	Lo que significa comúnmente
success	Todo ha funcionado. Ir a la página siguiente.
failure	Algo está mal. Ir a una página de error.
logon	El usuario debe iniciar sesión primero. Ir a la página de inicio de sesión.
no results	La búsqueda no encontraron nada. Ir a la página de búsqueda de nuevo.

Normalmente, el método de acción realiza un procesamiento de los datos del formulario de la página actual.

Por ejemplo, el método podría comprobar si el nombre de usuario y la contraseña introducida en el formulario de coincidir con el nombre de usuario y contraseña en el archivo. Si coinciden, el método devuelve el éxito de los resultados. De lo contrario, devuelve la falta de resultados.

Como demuestra este ejemplo, tanto el método utilizado para procesar la acción y los resultados devueltos son necesarias para determinar la página propia de acceso.

He aquí una regla de navegación que podrían ser utilizados con el ejemplo que acabamos de describir:

```
<navigation-rule>
    <from-view-id>/logon.jsp</from-view-id>
    <navigation-case>
        <from-action>#{LogonForm.logon}</from-action>
        <from-outcome>success</from-outcome>
        <to-view-id>/storefront.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
        <from-action>#{LogonForm.logon}</from-action>
        <from-outcome>failure</from-outcome>
        <to-view-id>/logon.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
```

Esta regla de navegación define las posibles maneras de navegar de logon.jsp. Cada elemento de navegación caso, define una ruta de navegación posible de logon.jsp. La navegación primer caso se dice que si LogonForm.logon devuelve un resultado de éxito, entonces storefront.jsp se tendrá acceso. La navegación segundo caso se dice que logon.jsp se vuelven a representar, si vuelve LogonForm.logon fracaso.

La configuración de una aplicación, el flujo de la página consta de un conjunto de reglas de navegación.

Cada regla se define por el elemento de regla de navegación en el archivo faces-config.xml.

Cada elemento de regla de navegación corresponde a un identificador del componente arbóreo definido por el facultativo de vista de elemento de identificación. Esto significa que cada Estado define todas las posibles maneras de navegar de una página especial en la aplicación.

Si no hay ningún elemento de la vista-de-id, las reglas de navegación se define en el elemento de regla de navegación se aplican a todas las páginas de la aplicación. La concordancia de patrones-view-ID de elemento también permite comodín. Por ejemplo, este de-vista-ID de elemento dice que la regla de navegación se aplica a todas las páginas en el directorio de libros:

```
<from-view-id>/libros/*</ from-view-id>
```

Como se indica en la regla de navegación ejemplo, un elemento de regla de navegación puede contener cero o más elementos de navegación caso. El elemento de navegación caso, define un conjunto de criterios de coincidencia. Cuando se cumplen estos criterios, la aplicación se vaya a la página definida por el a-ver-ID de elemento contenido en la navegación del mismo elemento de caso.

Los criterios son definidos por la navegación opcional de resultados-y de elementos de acción. El resultado de elemento define un resultado lógico, como el éxito. El elemento de la acción método utiliza una expresión para referirse a un método de acción que devuelve una cadena, que es el resultado lógico. El método realiza alguna lógica para determinar el resultado y devuelve el resultado.

Los elementos de navegación caso de que se cotejarán con los resultados y el método de expresión en este orden:

- Los casos que especifica la vez un resultado de valor y de valor de la acción. Ambos elementos pueden ser utilizados si el método de acción devuelve resultados diferentes dependiendo del resultado de la transformación que realiza.
- Los casos que especifica sólo un valor de resultado. El elemento de resultado debe coincidir con el resultado ya sea definido por el atributo de acción del componente de UICommand o el resultado devuelto por el método mencionado por el componente de UICommand.
- Especificando los casos sólo una de valor de la acción. Este valor debe coincidir con la expresión acción especificada por la etiqueta del componente.

Cuando se compara cualquiera de estos casos, el árbol de componentes definidos por el a-ver-elemento de identificación serán seleccionados para la representación.

3.6.2 Reglas de navegación implícitas

A partir de JavaServer Faces 2.0, las reglas de navegación implícita están disponibles para aplicaciones de Facelets. Las reglas de navegación implícita entrar en juego si no hay reglas de navegación se configuran en los archivos de configuración de recursos de aplicación.

Cuando se agrega un componente de interfaz de usuario como un comando y asigna una página como el valor de su propiedad la acción, el controlador de navegación por defecto tratan de combinar una página adecuada dentro de la aplicación.

```
<h:commandButton value="Enviar" action="response">
```

En el ejemplo anterior, el controlador de navegación por defecto tratar de localizar la página response.xhtml y navegar hacia él.

3.7 Backed Beans

Una aplicación típica de JavaServer Faces incluye uno o más beans de apoyo, cada uno de ellos es un JavaServer Faces gestionados de bean que está asociado con los componentes de la interfaz de usuario utilizados en una determinada página.

Los beans Gestionado son JavaBeans componentes que se pueden configurar mediante la instalación de bean gestionado, que se describe en Configuración de Beans. Esta sección presenta los conceptos básicos sobre la creación, configuración y uso de semillas de apoyo en una aplicación.

3.7.1 Crear una clase de backed bean

Además de definir un constructor sin argumento, como todos los componentes JavaBeans debe hacer, una clase de backed bean también define un conjunto de propiedades de los componentes de interfaz de usuario y, posiblemente, un conjunto de métodos que realizan funciones de un componente.

Cada una de las propiedades de los componentes se pueden enlazar a una de las siguientes:

- El valor de un componente
- Una instancia de componente
- Un ejemplo del convertidor
- Un ejemplo de escucha
- Un ejemplo de validador

Las funciones más comunes que los métodos de backed bean realizar son las siguientes:

- Validar los datos de un componente
- Manejo de un evento disparado por un componente de
- Realización de tratamiento para determinar la siguiente página para que la solicitud debe navegar

Como con todos los componentes JavaBeans, una propiedad consta de un campo de datos privados y un conjunto de métodos de acceso, como lo muestra este código:

```
UserNumber Integer = null;
...
setUserNumber public void (Integer user_number) (
    = userNumber user_number;
 )
getUserNumber public Integer () (
    userNumber retorno;
)
public String getResponse () (
    ...
)
```

Cuando una propiedad de bean está ligada al valor de un componente, puede ser cualquiera de los tipos básicos primitivos y numérico o cualquier tipo de objeto de Java para que la aplicación tenga acceso a un convertidor apropiado. Por ejemplo, una propiedad puede ser de tipo fecha, si la aplicación tiene acceso a un convertidor que puede convertir el tipo de fecha en una cadena y viceversa.

Cuando una propiedad se une a una instancia del componente, el tipo de la propiedad debe ser el mismo que el objeto de componentes. Por ejemplo, si un UISelectBoolean está ligado a la propiedad, la propiedad debe aceptar y devolver un objeto UISelectBoolean.

Del mismo modo, si la propiedad está enlazado a un convertidor, validador, o el oyente ejemplo, la propiedad debe ser del convertidor caso, validador, o el oyente tipo.

3.7.2 Configuración de un Bean

La tecnología JavaServer Faces soporta un sofisticado establecimiento administrado por la creación de bean, que permite a los arquitectos de aplicaciones para hacer lo siguiente:

- Configurar beans simple y árboles más complejo de beans
- Inicializar el bean con los valores
- Poner los beans en un ámbito particular (ámbitos disponibles: request, view, session, application)
- Expone los beans a la EL unificada para que los autores de páginas se puede acceder a ellos

El siguiente ejemplo muestra un ejemplo de archivo faces-config.xml:

```
<managed-bean>
    <managed-bean-name> UserNumberBean </ managed-bean-name>
    <managed-bean-class>
        guessNumber.UserNumberBean
    </ managed-bean-class>
    <managed-bean-scope> sesión </ managed-bean-scope>
    <managed-property>
        <property-name> mínimos </ propiedad name>
        <property-class> largo </ propiedad de clase>
        <valor> 0 </ value>
    </ managed-property>
    <managed-property>
        <property-name> <máximo / property-name>
        <property-class> largo </ propiedad de clase>
        <valor> 10 </ value>
    </ managed-property>
</ managed-bean>
```

La implementación de JavaServer Faces procesa la <managed-bean-scope> elemento en el momento de inicio de la aplicación. Cuando un bean es llamado la primera vez de la página, se crea una instancia.

Un autor de la página puede acceder a las propiedades de bean de las etiquetas de componentes en la página utilizando el EL unificada, como se muestra aquí:

```
<h:outputText value="#{UserNumberBean.minimum}"/>
```

También es posible aprovechar la función de anotaciones para el bean logrado evitar la configuración de bean gestionado en archivo de configuración de recursos de aplicación.

Ejemplo:

```
@ManagedBean(name = "Login")
@RequestScoped
public class Login {...}
```

3.8 Usando el EL unificada para referenciar Backed beans

Para obligar a los valores de la interfaz de usuario de los componentes y propiedades de los objetos al apoyo de bean o de respaldo para hacer referencia a los métodos de bean de etiquetas de componentes de interfaz de usuario, los autores de páginas utilizar el lenguaje de expresión unificada (EL) sintaxis definida por JSP 2.1.

Las siguientes son algunas de las características que ofrece este lenguaje:

- Evaluación diferida de las expresiones
- La capacidad de usar una expresión de valor tanto para leer y escribir datos
- Expresión de métodos

Estas características son especialmente importantes para apoyar el modelo de interfaz de usuario sofisticado componente que ofrece la tecnología JavaServer Faces.

Evaluación diferida de las expresiones es importante porque el ciclo de vida de JavaServer Faces está dividido en fases distintas, de manera que el manejo de eventos de componentes, la conversión de datos y validación, y la propagación de datos a los objetos externos, se realizan en forma ordenada. La aplicación debe ser capaz de retrasar la evaluación de las expresiones, hasta la fase apropiada del ciclo de vida se ha alcanzado. Por lo tanto, su etiqueta de atributos siempre utiliza la sintaxis de la evaluación diferida, que se distingue por el delimitador # {}.

Con el fin de almacenar datos en los objetos externos, casi todos los atributos de etiquetas JavaServer Faces utilizar expresiones de valor valor-, que son expresiones que permiten tanto obtener y establecer datos sobre los objetos externos.

Por último, algunos atributos de etiqueta de componente de aceptar expresiones método que los métodos de referencia para controlar los eventos de componentes, o validar o convertir los datos de los componentes.

Para ilustrar una etiqueta de JavaServer Faces usando el EL unificado, supongamos que una etiqueta de una solicitud de referencia a un método para realizar la validación de entrada de usuario:

```
<h:inputText id="userNo"
     value="#{UserNumberBean.userNumber}"
     validator="#{UserNumberBean.validate}" />
```

Esta etiqueta se une el valor del componente userNo para el bean de la propiedad UserNumberBean.userNumber utilizando una expresión value. Se utiliza un método de expresión para hacer referencia al método de UserNumberBean.validate, que realiza la validación de valor local del componente. El valor local es lo que el usuario entra en el campo correspondiente a esta etiqueta. Este método se invoca cuando se evalúa la expresión, que es durante la fase de validación del proceso del ciclo de vida.

Casi todos los atributos de etiquetas JavaServer Faces aceptar expresiones de valor. Además de las propiedades referencias, expresiones de valor puede también listas de referencias, mapas, paneles, objetos implícita, y los paquetes de recursos.

Otro uso de las expresiones de valor vinculante es una instancia del componente a una propiedad de backed bean. Un autor de la página hace referencia a la propiedad del atributo :

```
<inputText binding="#{UserNumberBean.userNoComponent}" />
```

Esas etiquetas de componentes que las expresiones método de uso son las etiquetas y las etiquetas de componentes UIInput componente UICommand.

3.8.1 El Ciclo de Vida de una página JavaServer Faces

El ciclo de vida de una página JavaServer Faces es algo similar a la de una página JSP: El cliente realiza una solicitud HTTP de la página y el servidor responde con la página traducida a HTML. Sin embargo, el ciclo de vida de JavaServer Faces difiere del ciclo de vida de JSP en que se divide en varias fases para apoyar el modelo de interfaz de usuario sofisticado componente. Este modelo requiere que los elementos de ser convertidos y validados, eventos de los componentes se manipulan, y los datos de los componentes se propaga a las judías en una manera ordenada.

Una página de JavaServer Faces es también diferente de una página JSP en la que es representado por un árbol de componentes de interfaz de usuario, denominado punto de vista. Durante el ciclo de vida, la implementación JavaServer Faces debe construir el punto de vista al considerar el estado guardado de una presentación anterior de la página. Cuando el cliente envía una página, la implementación JavaServer Faces realiza varias tareas, tales como la validación de la entrada de datos de los

componentes de la vista y la conversión de los datos de entrada a los tipos especificados en el lado del servidor.

La implementación de JavaServer Faces realiza todas estas tareas como una serie de pasos en la solicitud de JavaServer Faces ciclo de vida de respuesta.

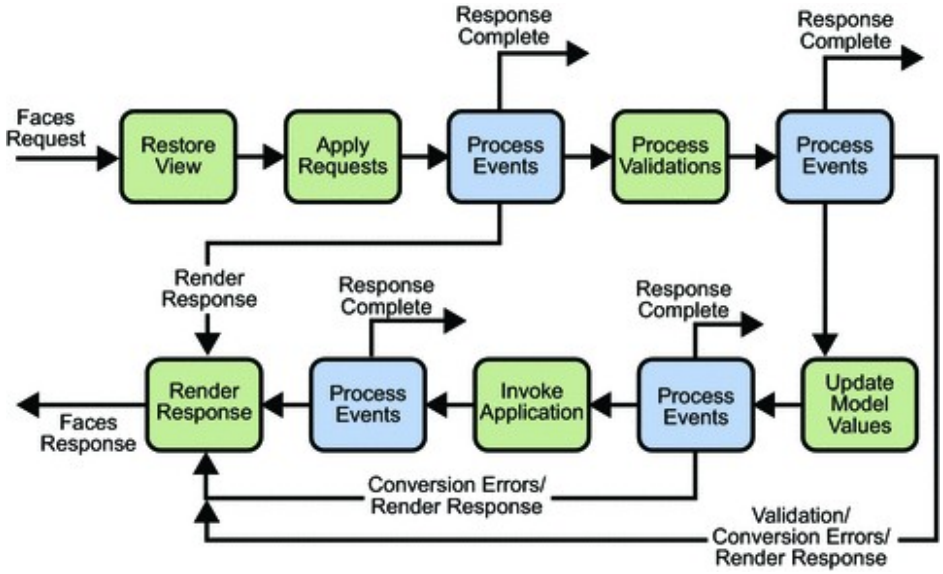


Diagrama de flujo de la solicitud de Faces y respuesta de Faces, incluyendo el evento de validación y el procesamiento, manejo de errores, el modelo de actualización, la invocación de la aplicación.

El ciclo de vida maneja ambos tipos de solicitudes: Las solicitudes iniciales y devoluciones de datos.

Cuando un usuario hace una solicitud inicial de una página, él o ella está solicitando la página por primera vez. Cuando un usuario ejecuta una devolución de datos, se le envía el formulario que figura en una página que fue previamente cargado en el navegador como resultado de la ejecución de una solicitud inicial.

Cuando el ciclo de vida se encarga de la petición inicial, sólo se ejecuta la restauración de vista y hacer que las fases de respuesta, porque no hay ninguna entrada del usuario o acciones para el proceso. Por el contrario, cuando el ciclo de vida maneja una devolución de datos, ejecuta todas las fases.

Normalmente, la primera solicitud para una página JavaServer Faces trata de un cliente, como resultado de hacer clic en un hipervínculo en una página HTML que enlaza con la página de JavaServer Faces.

Para hacer una respuesta que es otra página JavaServer Faces, la aplicación crea una nueva visión y la almacena en la instancia FacesContext, que representa a toda la información contextual asociada con el procesamiento de una solicitud entrante y crear una respuesta. La aplicación se adquiere referencias a objetos que necesita la opinión y solicita FacesContext.renderResponse, que obliga a la prestación inmediata de la opinión de pasar por la respuesta a la fase de procesamiento del ciclo de vida, como indican las flechas etiquetados Render de respuesta en el diagrama.

A veces, una aplicación podría necesidad de reorientar los recursos a diferentes aplicaciones web, como un servicio web, o generar una respuesta que no contiene componentes JavaServer Faces. En estas situaciones, el desarrollador debe saltarse la fase de renderizado (Render Fase de respuesta) llamando

FacesContext.responseComplete. Esta situación también se muestra en el diagrama, esta vez con las flechas de la etiqueta respuesta completa.

La propiedad de la currentPhaseID FacesContext, que representa la fase que se encuentra, debe ser actualizada tan pronto como sea posible por la aplicación.

La situación más común es que un componente JavaServer Faces presenta una solicitud de la página de otro JavaServer Faces. En este caso, la implementación JavaServer Faces maneja la solicitud y automáticamente pasa a través de las fases del ciclo de vida para realizar cualquier conversión necesaria, validaciones y actualización de los modelos, y para generar la respuesta.

Los detalles del ciclo de vida se explica en esta sección están destinados principalmente para desarrolladores que necesitan conocer dicha información como cuando validaciones, conversiones, y los eventos son generalmente tramitadas y lo que pueden hacer para cambiar cómo y cuándo se les da. Los autores de páginas no tienen por qué conocer los detalles del ciclo de vida.

3.8.2 Fase de restauración de vista

Cuando una solicitud de una página JavaServer Faces se hace, como cuando un enlace o un botón se presiona, la implementación JavaServer Faces comienza la fase de restauración de vista.

Durante esta fase, la implementación JavaServer Faces construye el punto de vista de la página, controladores de eventos y los validadores de los componentes en la vista, y guarda la vista en el ejemplo FacesContext, que contiene toda la información necesaria para procesar una solicitud única. Todas las etiquetas de componentes de la aplicación, los controladores de eventos, convertidores y validadores de tener acceso a la instancia de FacesContext.

Si la solicitud de la página es una solicitud inicial, la implementación JavaServer Faces crea una visión de vacío en esta etapa y los avances del ciclo de vida para hacer la fase de respuesta, durante el cual se llena con los componentes de referencia de las etiquetas en la página .

Si la solicitud de la página es una devolución de datos, una vista correspondiente a esta página ya existe. Durante esta fase, la implementación JavaServer Faces restaura la vista mediante el uso de la información de estado guardada en el cliente o el servidor.

3.8.3 Fase de aplicación de valores

Después de que el árbol de componentes que se restablezca, cada componente en el árbol de los extractos de su nuevo valor de los parámetros de la petición mediante el uso de su decodificar (processDecodes ()) método. El valor se almacena localmente en el componente. Si la conversión del valor de falla, un mensaje de error que está asociado con el componente se genera y en la cola en FacesContext. Este mensaje se mostrará durante la fase de dar respuesta, junto con los errores de validación resultante de la fase de validación del proceso.

Si los métodos descifrar o detectores de eventos llamado renderResponse FacesContext en la instancia actual, la implementación JavaServer Faces salta a la fase de dar respuesta.

Si los acontecimientos se han cola durante esta fase, las emisiones de aplicación JavaServer Faces los acontecimientos a los oyentes interesados.

Si algunos componentes de la página tienen sus atributos de inmediato establece en true, entonces la validación, la conversión, y los eventos relacionados con estos componentes serán tratados durante esta fase.

En este punto, si la solicitud tiene que reorientar los recursos a diferentes aplicaciones web o de generar una respuesta que no contiene componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

Al final de esta fase, los componentes se fijan a sus nuevos valores, y los mensajes y eventos se han puesto en cola.

Si la petición actual se identifica como una petición parcial, el contexto parcial se recupera de las caras de contexto y el método de transformación parcial es aplicado.

3.8.4 Fase de validaciones de proceso

Durante esta fase, la implementación JavaServer Faces procesa todos los validadores registradas en los componentes en el árbol, mediante su validación ((processValidators)) método. Se examina el componente de atributos que especifican las normas para la validación y compara estas normas para el valor local almacenado para el componente.

Si el valor local no es válido, la implementación JavaServer Faces añade un mensaje de error a la instancia FacesContext, y el ciclo de vida avanza directamente a la fase de dar respuesta para que la página se representa con el mensaje de error. Si hubo errores de conversión de la solicitud de aplicar los valores de fase, los mensajes de estos errores también se muestran.

Si alguna validar métodos o detectores de eventos llamado renderResponse en el FacesContext actual, la implementación JavaServer Faces salta a la fase de dar respuesta.

En este punto, si la solicitud tiene que reorientar los recursos a diferentes aplicaciones web o de generar una respuesta que no contiene componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

Si los acontecimientos se han cola durante esta fase, la implementación JavaServer Faces emisiones a oyentes interesados.

Si la petición actual se identifica como una petición parcial, el contexto parcial se recupera de las caras de contexto y el método de transformación parcial es aplicado. Procesamiento parcial está cubierto en Java EE 6 Tutorial, Volumen II: Temas avanzados.

3.8.5 Fase de actualización de valores de modelo

Después de la implementación JavaServer Faces determina que los datos son válidos, se puede recorrer el árbol de componentes y establecer el servidor correspondiente de las propiedades del objeto de lado a los valores locales de los componentes. La implementación de JavaServer Faces actualizar sólo las propiedades judías apuntado por un componente de valor de atributo de entrada. Si los datos locales no pueden ser convertidos a los tipos especificados por las propiedades judías, el ciclo de vida avanza directamente a la fase de dar respuesta para que la página es re-emitida con errores mostrados. Esto es similar a lo que sucede con los errores de validación.

Si los métodos updateModels o cualquier oyentes llamado renderResponse FacesContext en la instancia actual, la implementación JavaServer Faces salta a la fase de dar respuesta.

En este punto, si la solicitud tiene que reorientar los recursos a diferentes aplicaciones web o de generar una respuesta que no contiene componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

Si los acontecimientos se han cola durante esta fase, la implementación JavaServer Faces emisiones a oyentes interesados.

Si la petición actual se identifica como una petición parcial, el contexto parcial se recupera de las caras de contexto y el método de transformación parcial es aplicado.

3.8.6 Fase de invocación de la aplicación

Durante esta fase, la implementación JavaServer Faces maneja cualquier solicitud de eventos de nivel, tales como la presentación de un formulario o un vínculo a otra página.

En este punto, si la solicitud tiene que reorientar los recursos a diferentes aplicaciones web o de generar una respuesta que no contiene componentes JavaServer Faces, puede llamar a FacesContext.responseComplete.

Si se procesa el punto de vista fue reconstruido a partir de información de estado de una solicitud anterior y si un componente ha disparado un evento, estos eventos se transmiten a los oyentes interesados.

Por último, las transferencias de JavaServer Faces aplicación de control para hacer que la fase de respuesta.

3.8.7 Fase de creación de la respuesta

Durante esta fase, JavaServer Faces construye el punto de vista y delega la autoridad para el renderizado de las páginas. Por ejemplo, para el contenedor de JSP si la aplicación es la utilización de páginas JSP.

Si se trata de una solicitud inicial, los componentes que están representados en la página se agrega al árbol de componentes. Si esto no es una solicitud inicial, los componentes se añaden ya al árbol para que no se necesita añadir más.

Si la solicitud es una devolución de datos y los errores fueron encontrados durante la fase de solicitud de aplicar los valores, las validaciones de proceso de fase, o fase de actualización de los valores del modelo, la página original se representa en esta fase. Si las páginas contienen mensajes o los mensajes de las etiquetas, los mensajes de error en la cola se muestran en la página.

Después de que el contenido de la vista se representa, el estado de la respuesta se guarda para que las solicitudes posteriores se puede acceder a él. El estado guardado se está a disposición de la fase de restauración de vista.

4 Formularios

4.1 Formulario sencillo

Para poner en practica lo anterior visto, vamos a usar un formulario de entrada al sitio.

Lo vamos a construir poco a poco para llegar al código siguiente:
login.xhtml:

```
 <h:head>
        <title>Entrada al sitio</title>
        <link rel="stylesheet" href="curso.css" />
        <ui:include  src="WEB-INF/include/head.xhtml"/>
    </h:head>
    <h:body>
        <ui:include  src="WEB-INF/include/menu.xhtml"/>
       <h1><h:outputText value="Hello World!"/></h1>
            <h:graphicImage alt="GIF" url="/images/Duke_Blocks.gif" width="100px" />
            <h:form>
                <h:panelGrid columns="3">
                    <h:outputText value="Usuario" />
                    <h:inputText id="username" value="#{Login.username}" required="true"
validatorMessage="Minimo 2 caracteres!"
                                 requiredMessage="Value is required!"  >
                    <f:validateLength minimum="2" />
                    </h:inputText>
                    <h:message for="username" styleClass="error"/>
                    <h:outputText value="Clave" />
                    <h:inputText value="#{Login.password}" />
                    <h:commandButton value="Entrar" action="#{Login.submit}"
                                     actionListener="#{Login.onLogin}" />
                </h:panelGrid>
            </h:form>
    </h:body>
```

curso.css:

```
root {
    display: block;
}
.logo{
    font-size: 20px;
    color: teal;
}
```

Login.java:

```
package curso;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
@ManagedBean(name = "Login")
@RequestScoped
public class Login {
    private String username;
    private String password;
    public void onLogin(ActionEvent e) {
        if (!username.equalsIgnoreCase("dukes") && !username.equalsIgnoreCase("admin")) {
            throw new AbortProcessingException("Usuario invalido");
        }
        // password=null;
    }
    public String submit() {
        String outcome = "none";
        if (username.equalsIgnoreCase("dukes")) {
            outcome = "success";
        } else {
            outcome = "admin";

        }
```



```
        System.out.println(outcome);
        return outcome;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public String getUsername() {
        return username;
    }
}
```

faces-config.xml:

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

<navigation-rule>
    <from-view-id>/login.xhtml</from-view-id>
    <navigation-case>
        <from-outcome>admin</from-outcome>
        <to-view-id>/admin.xhtml </to-view-id>
   </navigation-case>
    <navigation-case>
        <from-outcome>success</from-outcome>
        <to-view-id>/index2.xhtml </to-view-id>
   </navigation-case>
</navigation-rule>
</faces-config>
```

Hay que crear las páginas index2. `xhtml` y admin. `xhtml` tambien.

4.2 Formulario mas complejo

Vamos a crear un formulario para agregar un usuario, con diferentes tipos de campos en el formulario.

user_add.xhtml

```
    <h:head>
            <title>Agregar un usuario</title>
        <link rel="stylesheet" href="curso.css" />
    </h:head>
    <h:body>
       <h1>Agregar un usuario</h1>
            <h:form>
                <h:panelGrid columns="3"  >
                    <h:outputLabel value="Nombre:"/>
                    <h:inputText id="nombre" value="#{Usuario.nombre}" size="20" maxlength="30"
required="true" validatorMessage="Minimo 2 caracteres!"
                                 requiredMessage="El nombre no puede estar vacio"  >
                        <f:validateLength minimum="2" />
                    </h:inputText>
                    <h:message for="nombre" styleClass="error"/>
```

```
                    <h:outputLabel value="Apellido:"/>
                    <h:inputText id="apellido" value="#{Usuario.apellido}" size="20" maxlength="30"
required="true" validatorMessage="Minimo 2 caracteres!"
                                requiredMessage="El apellido no puede estar vacio"  >
                        <f:validateLength minimum="2" /></h:inputText>
                    <h:message for="apellido" styleClass="error"/>
                    <h:panelGroup style="text-align:right;"><h:outputLabel
value="Domicilio:"/></h:panelGroup>
                    <h:inputTextarea id="add1" value="#{Usuario.add1}" cols="50" rows="3"
></h:inputTextarea>
                    <f:verbatim />
                    <h:outputLabel value="Sexo:"/>
                    <h:selectOneListbox title="Selecciona el sexo del usuario"
value="#{Usuario.sexo}" size="1">
                        <f:selectItem itemValue="M" itemLabel="Masculino"/>
                        <f:selectItem itemValue="F" itemLabel="Feminino"/>
                    </h:selectOneListbox>
                    <f:verbatim />
                    <h:outputLabel value="Cuantos herman@s"/>
                    <h:inputText id="hermanos" value="#{Usuario.hermanos}" size="1" maxlength="2"
required="true"  validatorMessage="Valor aceptados: de 0 a 25."
                                requiredMessage="Este campo no puede estar vacio"
                                converterMessage="Se espera un numero entero aqui." >
                        <f:validateLongRange minimum="0" maximum="25" />
                    </h:inputText>
                    <h:message for="hermanos" styleClass="error"/>
                    <h:outputLabel value="Salario"/>
                    <h:inputText id="salario" value="#{Usuario.salario}" size="10" maxlength="12"
required="true"  validatorMessage="Valor aceptados: de 0 a 250000."
                                requiredMessage="Este campo no puede estar vacio"
                                converterMessage="Formato incorrecto." >
                        <f:convertNumber maxFractionDigits="2" maxIntegerDigits="6" type="currency"
currencySymbol="C$" locale="es" pattern=""/>
                        <f:validateDoubleRange minimum="0" maximum="2500" />
                    </h:inputText>
                    <h:message for="salario" styleClass="error"/>
                    <h:panelGroup>
                        <h:commandButton value="Agregar el usuario" action="#{Usuario.onSubmit}"
></h:commandButton>
                    </h:panelGroup>
                </h:panelGrid>
            </h:form>
    </h:body>
```


5 HtmlDataTable

Un DataTable muestra una tabla HTML <table>.

Las columnas se especifican con los componentes secundarios UIColumn.

El HtmlDataTable tiene varios parametros:

- ID: Se utiliza para identificar el componente de la tabla. Este debe ser único en el componente más cercano de los padres.
- Value: Representa el valor del componente. Representa el valor sobre el que la iteración se debe hacer. Puede ser una matriz o cualquier objeto iterador.
- Var: Este es el nombre de la variable creada por la tabla de datos que representa el elemento actual en el valor. Este atributo permite exponer los datos en las filas de la tabla.
- Bgcolor: Este atributo se utiliza para establecer el color de fondo de la tabla.
- Border: Se puede configurar el ancho del borde de la mesa alrededor de la mesa.
- Cellpadding: Esto establece el espacio entre el contenido y la frontera de la célula.
- Cellspacing: Se especifica la cantidad de espacio para dejar entre las células.
- First: Se utiliza para especificar el número de fila de la primera fila de la que presentamos es que se inicie en adelante. Supongamos, esta propiedad se establece en 3, que muestra se iniciará desde la tercera fila de los datos subyacentes.
- Rows: Este atributo especifica el número de filas a mostrar. Viendo este se pondrá en marcha desde el índice especificado en la "primera" atributo. Si se establece este atributo a cero, entonces todas las filas que se mostrarán.
- Width: Esto se utiliza para establecer el ancho de toda la tabla. Su valor se especifica en%. Supongamos que se establece que el 50%, entonces esta tabla se muestra en el espacio del 50% de la anchura de la pantalla.
- Dir: Este atributo indica la dirección del texto que se mostrará en la celda. Se necesita "ltr" (de izquierda a derecha) y "RTL" (de derecha a izquierda) valores. Si no se especifica este atributo a continuación, el contenido se mostrará en el centro.
- Frame: Este atributo que specifyes lados de la estructura en torno a esta mesa será visible. Este atributo puede tomar algunos valores que se muestran a continuación:
    1. none Ninguna de las partes, Valor predeterminado
    2. top lado sólo por encima de
    3. below de lado el fondo sólo
    4. hsides partes superior e inferior sólo
    5. vsides lados derecho e izquierdo sólo
    6. LHS lado izquierdo sólo
    7. RHS lado derecho sólo
    8. box Todas las partes cuadro de cuatro
    9. border Todas las partes frontera cuatro

- Rules: Este atributo se utiliza para dibujar líneas entre las células. Puede tomar algunos valores que se indican a continuación:
    1. none: No hay reglas ninguno, el valor por defecto
    2. group: Entre los grupos de los grupos de filas
    3. rows: Entre las filas de las filas sólo
    4. cols: Entre columnas sólo
    5. all: Entre todas las todas las filas y columnas
- Summary: Puede especificar resumen del propósito de la tabla.
- Rendered: Se necesita value.This boolean Indica si este componente debe ser prestado. Su valor predeterminado es "true". Si se establece en false entonces se impide la prestación de este componente a la página.

- CaptionClass: lista separada por espacios de la clase o clases CSS que se aplicará a cualquier título generado para esta tabla.
- CaptionStyle: Especifica estilo CSS o estilos que deben aplicarse cuando se representa este epígrafe.
- ColumnClasses: lista separada por comas de las clases CSS que se aplicará a las columnas de este cuadro.
- FooterClass: Este atributo tiene lista separada por espacios de estilo CSS de la clase o clases que se aplicará a aheaderter generado para esta tabla.
- HeaderClass: Este atributo tiene lista separada por espacios de la clase de estilo CSS o clases que se aplicarse a cualquier encabezado generado para esta tabla.
- RowClasses: Es una lista de clases CSS aplicados a las filas de las clases table.These deben estar separadas por comas. Si queremos aplicar la clase de CSS para filas individuales, entonces podemos especificar lista separada por espacios de clases CSS. Clases de estilo se aplican a las filas en el mismo orden que se definidos. Si tenemos dos clases CSS continuación de primera categoría se aplica a la primera fila y la segunda se aplica a la segunda. Entonces, de nuevo en la tercera fila, el CSS se aplica primero y así sucesivamente. Este proceso continúa hasta la última fila de la tabla.
- Lang: Establece el idioma base de los atributos de un elemento, es decir, el texto y el lenguaje utilizado en el marcado generado para este componente.
- StyleClass: Se establece el nombre de las clases de classor CSS que se aplica en el momento de la prestación del elemento.
- Title: El atributo del título se utiliza para establecer el texto para mostrar información sobre herramientas para el component.Tooltip prestados describe un elemento cuando se representa para el cliente.
- Binding: Es un valor de la expresión de unión que es utilizado para conectar componentes a una propiedad en un backed bean.
- OnClick: Se establece el código JavaScript a ejecutar cuando se hace clic en los botones del ratón sobre este elemento.
- Ondblclick: Se establece el código JavaScript que se ejecuta cuando el botón del puntero es doble clic sobre este elemento.
- Onkeydown: Se establece el código JavaScript que se ejecuta cuando se presiona una tecla hacia abajo sobre este elemento.
- Onkeypress: Se establece el código JavaScript que se ejecuta cuando se pulsa una tecla y publicado sobre este elemento.
- Onkeyup: Se establece el código JavaScript que se ejecuta cuando se suelta una tecla sobre este elemento.
- Onmousedown: Se establece el código JavaScript que se ejecuta cuando el botón es pulsado el puntero sobre este elemento.
- Onmousemove: Se establece el código JavaScript a ejecutar cuando un botones del ratón se mueve dentro de este elemento.
- Onmouseout: Se establece el código JavaScript a ejecutar cuando un botones del ratón se aleja de este elemento.
- Onmouseover: Se establece el código JavaScript a ejecutar cuando un botones del ratón se mueve sobre este elemento.
- Onmouseup: Se establece el código JavaScript a ejecutar cuando un botones del ratón se suelta sobre este elemento.

Ejemplo:

user_details.xhtml

```
<h:head>
            <title>Detalles del usuario</title>
        <link rel="stylesheet" href="curso.css" />
    </h:head>
    <h:body>
```



```
            <h1>Detalles del usuario</h1>
            <h:dataTable value="#{Usuario}" var="user" border="1"
                         cellspacing="2">
                <h:column>
                    <f:facet name="header">
                        <h:outputText value="First Name"/>
                    </f:facet>
                    <h:outputText value="#{user.nombre}"/>
                </h:column>
                <h:column>
                    <f:facet name="header">
                        <h:outputText value="Last Name"/>
                    </f:facet>
                    <h:outputText value="#{user.apellido}"/>
                </h:column>
                <h:column>
                    <f:facet name="header">
                        <h:outputText value="Balance"/>
                    </f:facet>
                    <h:outputText value="#{user.salario}">
                        <f:convertNumber type="currency"/>
                    </h:outputText>
                </h:column>
            </h:dataTable>
    </h:body>

Usuario.java:
package curso.beans;
import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;
import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;

@ManagedBean(name="Usuario")
@RequestScoped
public class Usuario {
private String nombre;
private String apellido;
private String sexo;
private Date fecnac=new Date();
private String add1;
private String pasatiempo;
private Byte hermanos;
private Double salario=1550.25564;
private Integer testInt=12900;
    public Usuario() {
    }

    public Usuario(String nombre, String apellido, String sexo, Date fecnac, String add1, String
pasatiempo, Byte hermanos, Double salario) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.sexo = sexo;
        this.fecnac = fecnac;
        this.add1 = add1;
        this.pasatiempo = pasatiempo;
        this.hermanos = hermanos;
        this.salario = salario;
    }
    public String onSubmit(){
        return "success";
    }
    public void validateCCExpiry(FacesContext cntx, UIComponent cmp, Object val) {
        System.out.println("Validando la entrada "+val);
        String messS = null;
        String[] fields = ((String) val).split("/", 3);
        if (fields.length != 2) {
            messS = "Se espera MM/AA!";
        } else {
            int month = 0;
            int year = 0;
            try {
```

```
                month = Integer.parseInt(fields[0]);
                year = Integer.parseInt(fields[1]);
            } catch (NumberFormatException e) {
            }
            if (month <= 0 || month > 12) {
                messS = "Mes " + fields[0] + " invalido!";
            } else if (year < 0 || year > 99) {
                messS = "A&ntild:o " + fields[1] + " invalido!";
            } else {
                GregorianCalendar cal = new GregorianCalendar();
                int thisMonth = cal.get(Calendar.MONTH) + 1;
                int thisYear = cal.get(Calendar.YEAR) - 2000;
                if (year < thisYear || year == thisYear && month < thisMonth) {
                    messS = "Tarjeta vencida!";
                }
            }
        }
        if (messS != null) {
            FacesMessage mess = new FacesMessage(messS);
            cntx.addMessage(cmp.getClientId(cntx), mess);
        }
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getApellido() {
        return apellido;
    }
    public void setApellido(String apellido) {
        this.apellido = apellido;
    }
    public String getSexo() {
        return sexo;
    }
    public void setSexo(String sexo) {
        this.sexo = sexo;
    }
    public Date getFecnac() {
        return fecnac;
    }
    public void setFecnac(Date fecnac) {
        this.fecnac = fecnac;
    }
    public String getAdd1() {
        return add1;
    }
    public void setAdd1(String add1) {
        this.add1 = add1;
    }
    public String getPasatiempo() {
        return pasatiempo;
    }
    public void setPasatiempo(String pasatiempo) {
        this.pasatiempo = pasatiempo;
    }
    public void setSalario(Double salario) {
        this.salario = salario;
    }
    public void setHermanos(Byte hermanos) {
        this.hermanos = hermanos;
    }
    public Double getSalario() {
        return salario;
    }
    public Byte getHermanos() {
        return hermanos;
    }
    public void setTestInt(Integer testInt) {
        this.testInt = testInt;
    }
    public Integer getTestInt() {
        return testInt;
    }
```

```
}

demo_table.xhtml
<h:head>
        <title>Lista de usuarios</title>
        <link rel="stylesheet" href="curso.css" />
    </h:head>
    <h:body>
        <h1>Lista de usuarios</h1>
        <h:dataTable value="#{DemoTable.data}" var="user" border="1" cellspacing="2"
                     columnClasses="col1,col2,col1,col2" rowClasses="row1,row2"
                     headerClass="header">
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Nombre"/>
                </f:facet>
                <h:outputText value="#{user.nombre}"/>
            </h:column>

            <h:column>
                <f:facet name="header">
                    <h:outputText value="Apellido"/>
                </f:facet>
                <h:outputText value="#{user.apellido}"/>
            </h:column>

            <h:column>
                <f:facet name="header">
                    <h:outputText value="Sexo"/>
                </f:facet>
                <h:outputText value="#{user.sexo}"/>
            </h:column>

            <h:column>
                <f:facet name="header">
                    <h:outputText value="Salario"/>
                </f:facet>
                <h:outputText value="#{user.salario}">
                    <f:convertNumber type="currency"/>
                </h:outputText>
            </h:column>
        </h:dataTable>
    </h:body>

DemoTable.java
package curso.beans;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import javax.faces.bean.ApplicationScoped;
import javax.faces.bean.ManagedBean;

@ManagedBean(name="DemoTable")
@ApplicationScoped
public class DemoTable {
        List<Usuario> data = new ArrayList();
    public DemoTable() {
        populateData();
    }
    public List getData(){
        return data;
    }
    public void setData(Usuario a){
        data.add(a);
    }
    public void populateData(){
        data.add(new Usuario("Cedric","Simon","M",new Date(),"Managua","demo",(byte)2,2500.80));
        data.add(new Usuario("Juan","Palmado","M",new Date(),"Managua","demo",(byte)2,25.08));
        data.add(new Usuario("Melanie","Castro","F",new Date(),"Managua","demo",(byte)2,1205.05));
        data.add(new Usuario("Melanie2","Castro","F",new Date(),"Managua","demo",(byte)2,1205.05));
        data.add(new Usuario("Melanie3","Castro","F",new Date(),"Managua","demo",(byte)2,1205.05));
        data.add(new Usuario("Melanie4","Castro","F",new Date(),"Managua","demo",(byte)2,1205.05));
    }
}
```

6 Mensaje de error personalizados

6.1 Lista de mensaje personalizados

En faces-context.xml:

```
<application>
    <resource-bundle>
        <base-name>curso.ApplicationMessages</base-name>
        <var>ErrMsg</var>
    </resource-bundle>
</application>
```

En src/curso/ crear archivo ApplicationMessages.properties:

```
test=Hola Mundo
```

Uso: `#{ErrMsg.test}` → Hola Mundo

6.2 Página de error personalizada:

error.xhtml :

```
    <h:head>
        <title>Pagina de error</title>
    </h:head>
    <h:body>
        <h2>Un error fata ocurio!</h2>


        <a href="#{request.getContextPath()}/faces/login.xhtml">Favor prueba de nuevo.</a>

        <p><h:link outcome="success" value="Favor prueba de nuevo."></h:link></p>

        <p><h:link outcome="index" value="Favor prueba de nuevo."></h:link></p>
    </h:body>
```

Configurar en web.xml:

```
    <error-page>
        <location>/error.xhtml</location>
    </error-page>


    <error-page>
            <error-code>404</error-code>
            <location>/error.xhtml</location>
    </error-page>
```


7 Convertidores

7.1 ¿Qué es un convertidor?

Un convertidor se utiliza para dar formato "agradable" a un objeto, texto que se mostrará.

Por ejemplo, si desea para mostrar una fecha en JSP se puede utilizar un convertidor de formato de la fecha a un formato mas común para el usuario, como "10/03/2005".

Pero hay otra forma de usar un convertidor. Si los utiliza en combinación con un control de entrada, la entrada del usuario debe estar en el formato especificado por el convertidor. Si el formato de la entrada no coincide en el formato, puede lanzar una excepción en el convertidor que se muestra al usuario. El objeto asociado no se actualiza hasta que se corrige el error.

7.2 Uso de los convertidores

El convertidore puede ser fijado dentro de JSP, o usted puede registrar el convertidor mediante programación.

Usted puede registrar un convertidor con JSP en una de las tres maneras:

Especifique el identificador de convertidor con la propiedad “converter” dentro de la etiqueta.

```
<h:outputText value="#{myBean.date}" converter="myConverter">
```

Juntar <f:converter> de dentro de la etiqueta del componente.

```
<h:outputText value="#{myBean.date}">
<f:converter converterId="myConverter"/>
</ h: outputText>
```

Juntar la etiqueta personalizada del convertidor dentro de una etiqueta del componente.

```
<h:outputText value="#{myBean.date}">
<laliluna:myConverter />
</ h: outputText>
```

Los siguientes JSF Etiquetas apoya convertidores.

```
<h:outputText>
<h:outputFormat>
<h:outputLink>
<h:outputLabel>
<h:inputText>
<h:inputTextarea>
<h:inputHidden>
<h:inputSecret>
<h:selectBooleanCheckbox>
<h:selectManyListbox>
<h:selectMaynyMenu>
<h:selectOneRadio>
<h:selectOneListbox>
<h:selectOneMenu>
```

7.3 Convertidores incluidos

Si no se especifica un convertidor, JSF escoger uno para usted. El marco ha convertidores estándar para todos los tipos básicos: Long, byte, integer, short, Character, Double, Float, BigDecimal, BigInteger y Boolean .

Por ejemplo, si el componente se asocia con una propiedad de tipo boolean, JSF elija el convertidor de Boolean. Los tipos primitivos se convierten automáticamente a sus homólogos de objeto.

Además tiene convertidores predefinidos para fechas u números.

7.3.1 Convertidor DateTime

Para todos los tipos básicos de Java JSF usará automáticamente los convertidores. Pero si desea dar formato a una fecha objeto de JSF proporciona una etiqueta convertidor <f:convertDateTime>.

Esta etiqueta debe ser anidadas dentro de un tag componente que apoya los convertidores.

```
<h:outputText value="#{myBean.date}">
<f:convertDateTime type="date" dateStyle="medium"/>
</h:outputText>
```

El convertidor DateTime admite atributos, como el tipo o datestyle, para configurar el convertidor. El lista a continuación muestra los atributos que puede utilizar con el convertidor de DateTime.

Nombre del atributo	Descripción
datestyle	Especifica el estilo de formato para la porción de la fecha de la cadena. Las opciones válidas son de short, medium (por defecto), long y full. Sólo es válido si se establece el tipo de atributo.
timeStyle	Especifica el estilo de formato para la porción de tiempo de la cadena. Las opciones válidas son de short, medium (por defecto), long y full. Sólo es válido si se establece el tipo de atributo.
timezone	Especifica la zona horaria para la fecha. Si no se establece, hora del meridiano de Greenwich (GMT) se utilizará.
locale	El idioma local a utilizar para la visualización de esta fecha. Reemplaza la localización actual
pattern	El modelo de formato de fecha utilizado para convertir esta fecha. Utilice este o el tipo de propiedad.
type	Especifica si se debe mostrar la fecha (date), hora (time) o ambas (both).

7.3.2 Convertidor de Número

Este segundo convertidor se puede personalizar mediante el uso de atributos adicionales, es el convertidor de Número.
El es útil para mostrar números en los formatos de base que trabaja para la configuración regional del usuario.

```
<h:outputText value="#{myBean.date}">
<f:convertNumber type="number" maxIntegerDigits="3"/>
</ h: outputText>
```


La lista a continuación muestra el atributo que se puede utilizar con el convertidor de Número. Estos atributos permiten controlar de manera precisa cómo se muestra un número.

Nombre del atributo	Descripción
CurrencyCode	Especifica un período de tres dígitos del código de moneda internacional cuando el atributo tipo es la moneda. Utilice este o CurrencySymbol.
CurrencySymbol	Especifica un símbolo específico, como "$", que se utiliza cuando el tipo de atributo es moneda. Utilice este o CurrencyCode.
groupingUsed	True si un símbolo de agrupación, como "," o "" debe ser utilizado. El valor predeterminado es true.
integerOnly	Verdadero si sólo la parte entera del valor de entrada debe ser procesado (todos los decimales será ignorado). El valor predeterminado es falso.
locale	El local que se utilizará para la visualización de este número. Reemplaza el usuario localización actual
minFractionDigits	Una cantidad mínima de decimales que se vea.
maxFractionDigits	Máxima número de decimales que se vea.
minIntegerDigits	Una cantidad mínima de dígitos enteros para mostrar.
maxIntegerDigits	Máxima número de dígitos enteros para mostrar.
pattern	El modelo de formato decimal para convertir este número. Utilice este o tipo de atributo.
tipo	El tipo de número, por el número (number, por defecto), la moneda (currency), o por ciento (percent). Usar este o el patrón de este atributo.

Ejemplo:

```
converters.xhtml:
    <h:head>
        <title>Convertidores JSF</title>
    </h:head>
    <h:body>
        <h4>Convertidores</h4>

        <p> Mostrar sólo la fecha y la datestyle es <i> short </i> </p>
        <h:outputText value="#{Usuario.fecnac}">
            <f:convertDateTime type="date" dateStyle="short" />
        </h:outputText>

        <p> Mostrar sólo el tiempo y el timeStyle es <i> full </i> </p>
        <h:outputText value="#{Usuario.fecnac}">
            <f:convertDateTime type="time" timeStyle="full" />
        </h:outputText>

        <p> Pantalla de fecha y hora, la datestyle es <i> full </i> y la configuración regional es
<i> ru </i> </p>
        <h:outputText value="#{Usuario.fecnac}">
            <f:convertDateTime type="both" dateStyle="full" locale="ru" />
        </h:outputText>

        <p> Mostrar tanto, la fecha y la hora y el datestyle es <i> short </i> </p>
```

```
        <h:outputText value="#{Usuario.fecnac}">
            <f:convertDateTime type="both" dateStyle="short" />
        </h:outputText>

        <p> Muestra una fecha con el patrón de <i> dd.mm.aaaa HH: mm </i> </p>
        <h:outputText value="#{Usuario.fecnac}">
            HH:mm" pattern="dd.MM.yyyy <f:convertDateTime />
        </h:outputText>

        <h:form id="datetime1">
            <p> entrada de una fecha y el datestyle es <i> short </i> </p>
            <h:inputText value="#{Usuario.fecnac}">
                <f:convertDateTime type="date" dateStyle="short" />
            </h:inputText>
            <h:commandButton value="Send" />
        </h:form>
        <h:form id="datetime2">
            <p> de entrada de una fecha que coincide con este patrón <i> dd.mm.aaaa </i> </p>
            <h:inputText value="#{Usuario.fecnac}">
                <f:convertDateTime pattern="dd.MM.yyyy" />
            </h:inputText>
            <h:commandButton value="Send" />
        </h:form>

        <h4> convertidor Número </h4>

        <p> máxima de pantalla <i> 3 dígitos entero </i> </p>
        <h:outputText value="#{Usuario.testInt}">
            <f:convertNumber maxIntegerDigits="3" />
        </h:outputText>

        <p> Tipo de pantalla es <i> moneda </i> y la CurrencySymbol es <i> $ </i> </p>
        <h:outputText value="#{Usuario.testInt}">
            <f:convertNumber type="currency" currencySymbol="$"/>
        </h:outputText>

        <p> Tipo de pantalla es <i> por ciento </i> </p>
        <h:outputText value="#{Usuario.testInt}">
            <f:convertNumber type="percent"/>
        </h:outputText>

        Mostrar <p> máximo de 4 dígitos de fracción </p>
        <h:outputText value="#{Usuario.salario}">
            <f:convertNumber maxFractionDigits="4"/>
        </h:outputText>

        <p> Mostrar el número de patrones <i> # # # 0,00% </i> </p>
        <h:outputText value="#{Usuario.salario}">
            <f:convertNumber pattern="###0.00%"/>
        </h:outputText>

        <h:form id="number1">
            <p> entrada de un número, pero sólo los dígitos entero será procesado </p>
            <h:inputText value="#{Usuario.testInt}">
                <f:convertNumber integerOnly="true"/>
            </h:inputText>
            <h:commandButton value="Send" />
        </h:form>

        <h:form id="number2">
            <p> entrada de un número coincida con el patrón <i> ##0,00 </i> </p>
            <h:inputText value="#{Usuario.testInt}">
                <f:convertNumber pattern="##0,00"/>
            </h:inputText>
            <h:commandButton value="Send" />
        </h:form>
    </h:body>
```

7.4 Convertidores personalizados

A veces, los convertidores estándar no son suficientes.

Por ejemplo, usted puede necesitar guardar en una base de datos los números de tarjeta de crédito sin guiones o espacios. Para usar convertidor personalizado, usted necesita para crear una implementación de la interfaz que javax.faces.Converter, y sobreescibir sus métodos getAsObject y getAsString.

Debe implementar las dos direcciones del convertidor. Durante al Apply Request Values (fase 2), el servlet JSF utiliza el getAsObject método para convertir la cadena de entrada al modelo de objetos de datos. Durante Render Response (fase 9), el servlet JSF utiliza el método getAsString para hacer la conversión en la dirección opuesta, de manera que una cadena puede ser incluido en la respuesta HTML.

Una vez finalizado el conversor, tiene que registrarse con la aplicación. Se puede registrar en faces-config.xml o usando la notación `@FacesConverter`.

```
<converter>
<converter-id>CCNumberConverter</converter-id>
<converter-class>curso.converters.CCNumberConverter</converter-class>
</converter>
```

Para invocar el convertidor, usted necesita juntar como una propiedad de f: convertidor o asignar a la propiedad Converter del componente de entrada.

Ejemplo:
Vamos a crear un convertidor que va a limpiar un número de tarjeta de crédito de cualquier carácter no numérico.

```
package curso.converters;
import javax.faces.convert.Converter;
import javax.faces.context.FacesContext;
import javax.faces.component.UIComponent;
import javax.faces.convert.ConverterException;
import javax.faces.convert.FacesConverter;

@FacesConverter(value="CCNumberConverter")

public class CCNumberConvertEr implements Converter {
// getAsObject extracts from the input string all numeric characters
    public Object getAsObject(FacesContext ctx, UIComponent cmp,
            String val) {
        String convVal = null;
        if (val != null) {
            char[] chars = val.trim().toCharArray();
            convVal = "";
            for (int k = 0; k < chars.length; k++) {
                if (chars[k] >= '0' && chars[k] <= '9') {
                    convVal += chars[k];
                }
            }

            System.out.println("CCNumberConverter.getAsObject: '"
            + val + "' -> '" + convVal + "'");

        }
        return convVal;
    }
// getAsString inserts into the object string spaces to make it readable
// default: nnnn nnnn nnnn nnnn, Amex: nnnn nnnnnn nnnnn
    public String getAsString(FacesContext ctx, UIComponent cmp, Object val)
            throws ConverterException {
        String convVal = null;
        if (val != null) {
            int[] spaces = {3, 7, 11, 99};
            int[] amex = {3, 9, 99};
            String sVal = null;
            try {
                sVal = (String) val; // The val object should be a String!
            } catch (ClassCastException e) {
                throw new ConverterException("CCNumberConverter: Conversion Error");
            }
            int kSpace = 0;
```

```
            char[] chars = sVal.toCharArray();
            if (chars.length == 15) {
                spaces = amex;
            }
            convVal = "";
            for (int k = 0; k < chars.length; k++) {
                convVal += chars[k];
                if (spaces[kSpace] == k) {
                    convVal += ' ';
                    kSpace++;
                }
            }

            System.out.println("CCNumberConverter.getAsString: '"
            + sVal + "' -> '" + convVal + "'");
        }
        return convVal;
}
```

converters_perso.xhtml:

```
<h:head>
        <title>Convertidor Personalizado</title>
    </h:head>
    <h:body>
        <h4>Convertidor Personalizado </h4>
        <h:form id="number2">
            <p> entrada de un número de tarjeta</p>
            <h:inputText value="#{Usuario.pasatiempo}" converter="CCNumberConverter">
            </h:inputText>
            <h:commandButton value="Send" />
        </h:form>
</h:body>
```


8 Validadores

8.1 Uso y Creación de los validadores

¿Cómo garantizar que el usuario de la aplicación no puede compra un número negativo de libros? En realidad, la aplicación también debe rechazar cualquier intento de compra de cero libros. ¿Y qué hay de comprobar la validez de un número de tarjeta de crédito? Estas son tareas para los validadores.

JSF cuenta con cuatro tipos de mecanismos de validación:
• Integrada en los componentes
• Validación a nivel de aplicación
• Componentes de validación personalizada
• Validación por métodos de Backed Beans

8.2 Validadores incluidos

JSF proporciona los siguientes tres componentes de validación:
• F: validateDoubleRange: Se valida que una entrada numérica está dentro de un rango determinado. Es aplicables a los valores que se pueden convertir a un doble.
• f: validateLength: Se valida que la longitud de la cadena de entrada está dentro de un rango determinado.
• F: validateLongRange: Se valida que una entrada numérica está dentro de un rango determinado. Es aplicables a los valores que se pueden convertir a un long.

Para utilizar estos componentes de validación, simplemente anidar dentro de la h: input que necesita validación. Por ejemplo, para comprobar que sólo cantidades positivas se pueden introducir:

```
<h:inputText id="quantity" value="#{item.quantity}" size="2"
required="true"
requiredMessage="Cuantos? Ninguno?"
converterMessage="Un entero por favor!"
validatorMessage="Minimo uno!">
<f:validateLongRange minimum="1"/>
</h:inputText>
```

8.3 Validación a nivel de aplicación

La validación a nivel de aplicación tiene sentido si es necesario para validar la lógica de aplicación, a diferencia de la validación de la corrección formal de los campos individuales. Por ejemplo, antes de aceptar un pedido, le gustaría comprobar que su banco no tiene en la lista negra el número de tarjeta de crédito.

Si desea hacer alguna solicitud de validación a nivel de aplicación, puede insertar su lógica al principio del método llamada por el formulario y hacer que la actualización de bases de datos y el resultado método depende del resultado de validación. En caso de error de validación, también podría enviar un mensaje para el usuario, como se muestra en las pocas líneas siguientes:

```
FacesContext ctxt = FacesContext.getCurrentInstance();
FacesMessage mess = new FacesMessage();
mess.setSeverity(FacesMessage.SEVERITY_ERROR);
mess.setSummary("Este es el mensaje de error principal");
mess.setDetail("Este es el detalle");
ctxt.addMessage(null, mess);
```

El mensaje creado de esta manera es un mensaje global, no vinculada a ningún componente en particular, y se puede visualizar con el componente JSF siguiente:

```
<h:messages globalOnly="true" styleClass="error"/>
```

8.4 Validadores personalizados

En el capitulo sobre los convertidores, se explico cómo implementar un convertidor personalizado. Para implementar un validador personalizado, es un proceso casi idéntico:

- Crear una aplicación de la interfaz que javax.faces.validator.Validator y reemplaza el método validar.
- Registrar el validador en faces-config.xml o usa la notación Doclet en la clase.
- Dentro de la aplicación de JSF, consulte el atributo validator del componente.

Supongamos que desea asegurarse de que el crédito, fecha de caducidad de tarjeta proporcionada por el usuario durante el pedido está en el formato mm / año y que la tarjeta no ha caducado.

```
package curso.validators;
import javax.faces.validator.Validator;
import javax.faces.context.FacesContext;
import javax.faces.component.UIComponent;
import javax.faces.application.FacesMessage;
import javax.faces.validator.ValidatorException;
import java.util.GregorianCalendar;
import java.util.Calendar;
import javax.faces.validator.FacesValidator;

@FacesValidator("CCExpiryValidator")
public class CCExpiryValidator implements Validator {
    public CCExpiryValidator() {
    }

    public void validate(FacesContext cntx, UIComponent cmp, Object val) {
        String messS = null;
        String[] fields = ((String) val).split("/", 3);
        if (fields.length != 2) {
            messS = "Se espera MM/AA!";
        } else {
            int month = 0;
            int year = 0;
            try {
                month = Integer.parseInt(fields[0]);
                year = Integer.parseInt(fields[1]);
            } catch (NumberFormatException e) {
            }
            if (month <= 0 || month > 12) {
                messS = "Mes " + fields[0] + " invalido!";
            } else if (year < 0 || year > 99) {
                messS = "A&ntild:o " + fields[1] + " invalido!";
            } else {
                GregorianCalendar cal = new GregorianCalendar();
                int thisMonth = cal.get(Calendar.MONTH) + 1;
                int thisYear = cal.get(Calendar.YEAR) - 2000;
                if (year < thisYear || year == thisYear && month < thisMonth) {
                    messS = "Tarjeta vencida!";
                }
            }
        }
        if (messS != null) {
            FacesMessage mess = new FacesMessage(
                    FacesMessage.SEVERITY_ERROR, messS, messS);
            throw new ValidatorException(mess);
        }
    }
}
```

Para registrar el validador, es necesario agregarlo a faces-config.xml o usar la anotación `@FacesValidator`.

```
<validator>
<validator-id>CCExpiryValidator</validator-id>
<validator-class>curso.validators.CCExpiryValidator</validator-class>
</validator>
```

Uso:

```
<h:form>
                <h:inputText id="ccexpiry" value="#{Usuario.apellido}" rendered="true"
                             requiredMessage="No puede estar vacio">
                    <f:validator validatorId="CCExpiryValidator" />
                </h:inputText>
                    <h:message for="ccexpiry" errorClass="error" />
                    <f:verbatim><br/></f:verbatim>
                <h:commandButton value="Probar"></h:commandButton>
</h:form>
```

8.5 Validador en Backed Bean

En lugar de crear una nueva clase como se describe en la sección anterior, puede agregar un método a un backed bean.

En este caso, puedes hacer lo siguiente:

- Copiar el método de validación dentro de la clase
- Copia de las importaciones
- Sustituya la línea que lanza la ValidatorException con :

```
        if (messS != null) {
            FacesMessage mess = new FacesMessage(messS);
            cntx.addMessage(cmp.getClientId(cntx), mess);
        }
```

Ejemplo:

```
<h:inputText id="ccexpiry" value="#{Usuario.apellido}" rendered="true" requiredMessage="No puede
estar vacio"  validator="#{Usuario.validateCCExpiry}">
</h:inputText>
<h:message for="ccexpiry" errorClass="error" />
```

9 Componentes personalizados

9.1 Creación de componentes personalizados para JSF

La funcionalidad de un componente se centra en la conversión de lo sometido por un usuario (es decir, los parámetros HTTP de la petición) a valores de los componentes (a través del método de decodificación durante Apply Request Values) y la conversión de valores de los componentes de vuelta a HTML (a través del método de codificar durante Render Response).

Cuando se diseña un componente JSF, se puede optar por cambiar la codificación y decodificación de una clase separada de procesamiento. La ventaja de este enfoque es que se puede desarrollar más de un procesador para el mismo componente, cada uno con una representación diferente en HTML. Usted a continuación, tienen el mismo comportamiento asociados con diferentes maneras de leer los datos de la solicitud de y la escritura a la respuesta.

En general, teniendo en cuenta que JSF es de código abierto, podría considerar la modificación de un componente existente en lugar de desarrollar un nuevo, o quizás, gracias a la separación de los componentes y los renderer, la modificación de un renderer existentes.

La clase raíz de todos los componentes JSF es la clase abstracta javax.faces.component.UIComponent, y de la clase raíz de todos los renderers es javax.faces.render.Renderer. Para desarrollar un componente, sin embargo, usted probablemente va a preferir ampliar un componente ya existente o, por lo por lo menos, la clase UIComponentBase, que proporciona implementaciones por defecto de todos los abstractos los métodos de UIComponent. De esta forma, usted sólo tiene que desarrollar código para los métodos que se necesidad de reemplazar. Lo mismo ocurre con el procesador.

Para completar el panorama de lo que usted necesita hacer para tener su componente personalizado, es crear una etiqueta personalizada para usarlo con JSP. La clase raíz de todas las clases de etiqueta javax.faces.webapp.UIComponentELTag.

En resumen, para desarrollar un componente personalizado, tiene que ir a través de los pasos siguientes, aunque no necesariamente en este orden:

- Crear una clase de componentes que las subclases UIComponent por la ampliación de una existente componente.
- Registrar el componente en faces-config.xml.
- Crear una clase de procesador que subclasa Renderer y reemplaza los métodos de codificación y descodificación.
- Registrar el renderirer en faces-config.xml.
- Crear una etiqueta personalizada que subclasa UIComponentELTag.
- Crear un TLD para la etiqueta personalizada.

Una última palabra acerca de los componentes y los renderers: a menos que usted realmente piensa que usted va a la reutilizar el mismo componente para diferentes aplicaciones, te harás la vida mucho más fácil si mantienes el renderer dentro del componente.

En primer lugar, vamos a ver lo que se debe hacer cuando están separados, y a continuación veremos cómo mantenerlos juntos.

Le mostraremos cómo desarrollar un componente que combina la funcionalidad de los tres componentes estándar necesarios para aceptar la entrada de un usuario: una etiqueta que explica lo que se espera, el

campo de texto para aceptar la entrada, y un mensaje para informar de errores de entrada. En otras palabras, vamos a muestra cómo reemplazar el código de JSF siguientes:

```
<h:outputText value="Nombre de Contacto"/>
<h:inputText id = "nombre" required = "true"
value = "# {Usuario.nombre}"
requiredMessage = "El valor es necesario!"
/>
<h:message for="name" styleClass="error"/>
```

con este componente personalizado:

```
<curso:inputEntry label="Nombre de Contacto" required = "true"
value = "#{Usuario.nombre}"
errorStyleClass ="error" requiredMessage = "El valor es necesario!"
/>
```

También le mostraremos cómo este nuevo componente inputEntry imprime un asterisco al lado de la etiqueta si es necesario = "true".

9.1.1 El Componente

El componente es en realidad la parte más fácil. Vamos a través de los métodos de uno por uno.

InputEntryComponent es la inicialización de los componentes.

Su única tarea es registrar con el componente de la cadena que identifica el renderer. La única propiedad del componente se define en este archivo es la etiqueta (label). Esto se debe a que esta ampliación UIInput, que se encarga de definir todo lo que tiene que ver con el campo de entrada.

Se utiliza el método getFamily para encontrar todos los renderers asociados con este componente.

Estamos planeando crear un solo procesador, pero sigue siendo apropiada para definir una familia más bien que heredan de la familia de UIInput, porque no se podía utilizar renderizadores UIInput con InputEntryComponent.

El estado del componente consiste en el estado de UIInput más la propiedad de la etiqueta. Por lo tanto, definir su estado como un arreglo de dos objetos. El metodo SaveState forma una matriz y lo devuelve, para que JSF puede salvarlo. El método restoreState recibe el Estado, lo descomprime, y lo almacena localmente. Observe cómo las operaciones que tienen que ver con UIInput son siempre delegadas.

```
package curso.components;

import javax.faces.component.FacesComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;

@FacesComponent("curso.inputEntry")
public class InputEntryComponent extends UIInput {

    private String label;

    public InputEntryComponent() {
        this.setRendererType("curso.inputEntry");
    }

    public String getLabel() {
        return label;
    }

    public void setLabel(String label) {
```

```
        this.label = label;
    }
// Overridden methods
    public String getFamily() {
        return "curso.inputEntry";
    }

    public void restoreState(FacesContext ctxt, Object state) {
        Object val[] = (Object[]) state;
        super.restoreState(ctxt, val[0]);
        label = (String) val[1];
    }

    public Object saveState(FacesContext ctxt) {
        Object val[] = new Object[2];
        val[0] = super.saveState(ctxt);
        val[1] = label;
        return ((Object) val);
    }
}
```

Ahora que tiene el componente, usted tiene que registrarlo. Para ello, mediante el uso de una notación `@FacesComponent` o la inserción de la siguientes líneas en faces-config.xml:

```
<component>
<component-type>curso.inputEntry</component-type>
<component-class>curso.components.InputEntryComponent</component-class>
</component>
```

9.1.2 Renderer

El renderer es un poco más complicado que el componente. Para implementarlo, se define una clase que extiende javax.faces.render.Renderer.

Primero vamos a sobreescribir tres métodos, decode, encodeBegin, y getConvertedValue.

La única propiedad que se agrega a UIInput es la etiqueta, que el usuario puede modificar.

Por lo tanto, sólo es necesario para decodificar el campo de entrada.

El proceso de decodificación ocurre en la fase de Apply Request Values (fase 2).

El proceso de codificación requiere más trabajo que el proceso de decodificación, porque tiene para enviar a la respuesta HTTP los tres componentes que se combinan para formar InputEntryComponent. Esto tiene lugar durante Render Response (fase 9).

Tenemos que crear un método para codificar la etiqueta. Se abre el elemento HTML con la etiqueta el método de startElement, escribe la etiqueta con un simple método de escritura, escribe un asterisco, pero sólo si se requiere el componente, y se cierra el elemento de la etiqueta con el método de endElement.

El resultado es algo así como `<label>Nombre de contacto * </label>`.

Creamos otro método para codificar el campo de entrada. Se abre el elemento de entrada HTML input, añade los atributos con el método writeAttribute, y cierra el elemento. Los tres parámetros de writeAttribute son el nombre y el valor del atributo HTML y el nombre de la propiedad del componente.

El resultado es algo como el siguiente elemento:

```
<input type="text" id="form:nameEntry" name="form:nameEntry" value="" />
```


encodeMessage es un método para codificar el mensaje de error. Se obtiene la lista de todos los mensajes de la cola para el componente, pero sólo muestra el primero. Si usted desea mostrar a todos ellos, sólo tiene que sustituir la palabra clave `if` con un `while`.

Para mostrar el mensaje, el método abre el elemento SPAN HTML, agrega el atributo de clase para mostrar el mensaje con el estilo correcto, muestra el mensaje, y se cierra el elemento. El resultado es algo como lo siguiente elemento: `<span class="error">Un valor es necesario!</span>`

```
package curso.renderers;

import curso.components.InputEntryComponent;
import java.io.IOException;
import java.util.Iterator;
import java.util.Map;
import javax.el.ValueExpression;
import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;
import javax.faces.render.FacesRenderer;
import javax.faces.render.Renderer;
@FacesRenderer(rendererType="curso.inputEntry", componentFamily="curso.inputEntry")
public class InputEntryRenderer extends Renderer {

    public void decode(FacesContext ctxt, UIComponent cmp) {
        InputEntryComponent ieCmp = (InputEntryComponent) cmp;
        Map requestMap = ctxt.getExternalContext().getRequestParameterMap();
        String clientId = cmp.getClientId(ctxt);
        String val = (String) requestMap.get(clientId);
        ((UIInput) ieCmp).setSubmittedValue(val);
    }

    public void encodeBegin(FacesContext ctxt, UIComponent cmp)
            throws IOException {
        InputEntryComponent ieCmp = (InputEntryComponent) cmp;
        ResponseWriter respWr = ctxt.getResponseWriter();
        encodeLabel(respWr, ieCmp);
        encodeInput(respWr, ieCmp);
        encodeMessage(ctxt, respWr, ieCmp);
        respWr.flush();
    }

    public Object getConvertedValue(FacesContext ctxt, UIComponent cmp,
            Object subVal) throws ConverterException {
        Object convVal = null;
        ValueExpression valExpr = cmp.getValueExpression("value");
        if (valExpr != null) {
            Class valType = valExpr.getType(ctxt.getELContext());
            if (valType != null) {
                convVal = subVal;
                if (!valType.equals(Object.class) && !valType.equals(String.class)) {
                    Converter converter = ((UIInput) cmp).getConverter();
                    converter = ctxt.getApplication().createConverter(valType);
                    if (converter != null) {
                        convVal = converter.getAsObject(ctxt, cmp, (String) subVal);
                    }
                }
            }
        }
        return convVal;
    }

    private void encodeLabel(ResponseWriter respWr, InputEntryComponent cmp)
            throws IOException {
        respWr.startElement("label", cmp);
        respWr.write(cmp.getLabel());
        if (cmp.isRequired()) {
            respWr.write("*");
        }
```

```
        respWr.endElement("label");
    }

    private void encodeInput(ResponseWriter respWr, InputEntryComponent cmp)
            throws IOException {
        FacesContext ctxt = FacesContext.getCurrentInstance();
        respWr.startElement("input", cmp);
        respWr.writeAttribute("type", "text", "type");
        respWr.writeAttribute("id", cmp.getClientId(ctxt), "id");
        respWr.writeAttribute("name", cmp.getClientId(ctxt), "name");
        if (cmp.getValue() != null) {
            respWr.writeAttribute("value", cmp.getValue().toString(), "value");
        }
        respWr.endElement("input");
    }

    private void encodeMessage(FacesContext ctxt, ResponseWriter respWr,
            InputEntryComponent cmp) throws IOException {
        Iterator it = ctxt.getMessages(cmp.getClientId(ctxt));
// Notice: an if instead of a while
        if (it.hasNext()) {
            FacesMessage mess = (FacesMessage) it.next();
            if (!cmp.isValid()) {
                String errorStyleClass =
                        (String) cmp.getAttributes().get("errorStyleClass");
                respWr.startElement("span", cmp);
                respWr.writeAttribute("class", errorStyleClass, "class");
                respWr.write(mess.getDetail());
                respWr.endElement("span");
            }
        }
    }
}
```

Para registrar el renderer, usa una anotación `@FacesRenderer` o inserta las líneas siguientes en faces-config.xml:

```
<render-kit>
<renderer>
<component-family>curso.inputEntry</component-family>
<renderer-type>curso.inputEntry</renderer-type>
<renderer-class>curso.renderers.InputEntryRenderer</renderer-class>
</renderer>
</render-kit>
```

9.1.3 Tag

El componente personalizado se hizo, pero para usarlo con JSP es necesario definir la etiqueta personalizada correspondiente. Se crea igual que una acción personalizada pero extendiendo la clase UIComponentELTag en este caso.

Se define una propiedad para cada atributo con el apoyo de la etiqueta, pero no para el atributo id. La razón es que UIComponentELTag ya lo define.

Tenga en cuenta que sólo tienen métodos setter, sin los getters correspondiente. Esto es porque nunca se necesita el métodos get. El método SetProperties copia los valores de atributo de la etiqueta al componente, y el método de liberación limpia lo que ya no es necesario.

```
package curso.tags;

import javax.el.ValueExpression;
import javax.faces.component.UIComponent;
import javax.faces.webapp.UIComponentELTag;

public class InputEntryTag extends UIComponentELTag {
    private ValueExpression errorStyleClass;
    private ValueExpression label;
    private ValueExpression required;
    private ValueExpression requiredMessage;
    private ValueExpression value;
```

```
// Setters
    public void setErrorStyleClass(ValueExpression errorStyleClass) {
        this.errorStyleClass = errorStyleClass;
    }

    public void setLabel(ValueExpression label) {
        this.label = label;
    }

    public void setRequired(ValueExpression required) {
        this.required = required;
    }

    public void setRequiredMessage(ValueExpression requiredMessage) {
        this.requiredMessage = requiredMessage;
    }

    public void setValue(ValueExpression value) {
        this.value = value;
    }
// Overridden methods

    public String getComponentType() {
        return "curso.inputEntry";
    }

    public String getRendererType() {
        return "curso.inputEntry";
    }

    protected void setProperties(UIComponent cmp) {
        super.setProperties(cmp);
        if (errorStyleClass != null) {
            cmp.setValueExpression("errorStyleClass", errorStyleClass);
        }
        if (label != null) {
            cmp.setValueExpression("label", label);
        }
        if (required != null) {
            cmp.setValueExpression("required", required);
        }
        if (requiredMessage != null) {
            cmp.setValueExpression("requiredMessage", requiredMessage);
        }
        if (value != null) {
            cmp.setValueExpression("value", value);
        }
    }

    public void release() {
        super.release();
        errorStyleClass = null;
        label = null;
        requiredMessage = null;
        value = null;
        required = null;
    }
}
```

Antes de poder utilizar la etiqueta personalizada en JSP aún necesidad de crear un TLD a poner en la carpeta WEB-INF/tlds/ .

curso.tld:

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.1" 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-
jsptaglibrary_2_1.xsd">
    <tlib-version>1.0</tlib-version>
    <short-name>curso</short-name>
    <uri>/WEB-INF/tlds/curso</uri>
    <tag>
        <display-name>inputEntry</display-name>
```

```
        <name>inputEntry</name>
        <tag-class>curso.tags.InputEntryTag</tag-class>
        <attribute>
            <name>id</name>
            <required>false</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>
        <attribute>
            <name>value</name>
            <required>false</required>
            <deferred-value><type>java.lang.Object</type></deferred-value>
        </attribute>
        <attribute>
            <name>required</name>
            <required>false</required>
            <deferred-value><type>boolean</type></deferred-value>
        </attribute>
        <attribute>
            <name>label</name>
            <required>false</required>
            <deferred-value><type>java.lang.String</type></deferred-value>
        </attribute>
        <attribute>
            <name>errorStyleClass</name>
            <required>false</required>
            <deferred-value><type>java.lang.String</type></deferred-value>
        </attribute>
        <attribute>
            <name>requiredMessage</name>
            <required>false</required>
            <deferred-value><type>java.lang.String</type></deferred-value>
        </attribute>
    </tag>
</taglib>
```

Con esto, usted estará preparado para utilizar el nuevo h JSF componente de interfaz de usuario: inputEntry. Sólo es necesario agregar la siguiente declaración de espacio de nombres `curso` a la jsp: elemento raíz en el inicio de el documento JSP: `<%@taglib prefix="curso" uri="WEB-INF/tlds/curso.tld"%>`

La desventaja del nuevo componente en comparación con tener de manera independiente los label, input, y message, es que no es posible alinear los campos de entrada vertical.

Ahora que ya está hecho, usted puede preguntar, "¿Por qué curso: inputEntry sólo admiten un atributo, mientras que h:inputText tiene como 40? "Eso es un buen punto. Podríamos haber añadido más atributos, tales como el tamaño del campo de entrada, que h:inputText pasa a HTML, pero nuestro objetivo era ser capaz de reemplazar con un componente único de los tres los campos en que se utilicen. Más atributos de lo estrictamente necesario sólo generaría más código sin añadir nada útil.

9.1.4 Inline Renderer

Es posible incluir la funcionalidad de renderer dentro de la clase de componentes, de modo que el componente se representa (render) por sí mismo. Como hemos mencionado antes, a menos que usted planea utilizar más de un procesador con el mismo componente, es posible que no se va a molestar creando rentderers separados.

Para hacer curso: inputEntry con auto-representación, tiene que hacer lo siguiente:
1. Mueva los métodos de InputEntryRenderer.java a InputEntryComponent.java. Usted necesidad hacer algunos cambios cosméticos que vamos a explicar en un momento. Despues se puede borrar el archivo de renderer.
2. Agregue el método encodeEnd a InputEntryComponent.java.
3. Return null en el método de getRendererType de InputEntryTag.java.
4. Eliminar el registro del procesador de faces-config.xml.

La clase UIInput, que se extiende para crear el componente, hospeda los tres métodos decode, encodeBegin, y getConvertedValue que utilizó en el renderer por separado, pero sin el parámetro UIComponent. Tiene mucho sentido, porque el objeto componente es directamente accesibles con la palabra clave `this`.

Cuando se quita el parámetro de cmp de los tres métodos, también deberá eliminar esta línea de decodificar y encodeBegin, porque se ha convertido en inútil:
```
InputEntryComponent ieCmp = (InputEntryComponent)cmp;
```

Luego, remplaza todos los `cmp` y `ieCmp` con `this`.

Se necesita el método encodeEnd para reemplazar el método en UIComponentBase, que arroja un NullPointerException. De hecho, usted no necesita hacer nada en encodeEnd, solamente tienes que escribir un método de vacío: `public void encodeEnd(FacesContext context) throws IOException { }`

Tenga en cuenta que sólo necesita este método cuando se ejecuta un componente en sí mismo, no cuando se utiliza una clase de procesador separado.

En InputEntryTag.java, el método devuelve getRendererType "curso.inputEntry". Si el método consiste en utilizar sus métodos de representación interna, getRendererType tiene que volver null. Por último, eliminar las líneas del elemento de render kit de faces-config.xml si están presente.

9.2 Componentes personalizados para Facelets

Con los Facelets es mas sencilla la creación de componentes personalizados.

No hay necesidad de crear una clase Java, solo necesitamos archivos XHTML (y XML).

Hay que crear un equivalente al TLD para facelets (.xml), y declararlo en web.xml

Ejemplo de inclusión en web.xml:
```
<context-param>
    <param-name>facelets.LIBRARIES</param-name>
    <param-value>/WEB-INF/facelets/curso.xml</param-value>
</context-param>
```

Ejemplo de descriptor curso.xml:
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE facelet-taglib PUBLIC
"-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
"http://java.sun.com/dtd/facelet-taglib_1_0.dtd">
<facelet-taglib>
    <namespace>http://localhost</namespace>
    <tag>
        <tag-name>pie</tag-name>
        <source>../include/footer.xhtml</source>
    </tag>
</facelet-taglib>
```

Pagina footer.xhtml:
```
<?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"
      xmlns:ui="http://java.sun.com/jsf/facelets">
    <h:body>
        <ui:component>
            <h:panelGroup styleClass="footer">
                <h:outputText value="Docente: Cedric Simon - "/>
```

```
                <h:outputLink
value="mailto:cedric@solucionjava.com">cedric@solucionjava.com</h:outputLink>
            </h:panelGroup>.
        </ui:component>
    </h:body>
</html>
```

La página a incluir es una página JSF XHTML, y el componente que queremos incluir se encuentra entre `<ui:component>` y `</ui:component>`.

Los parametros se recuperan con #{nombreDelParametro}. Se acceptan como parametros tambien objetos JSF o metodos.

Ejemplo mas complejo, equivalente a InputEntry:
input_entry.xhtml:

```
<?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"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:c="http://java.sun.com/jsp/jstl/core">
    <h:body>
        <ui:component>
            <h:outputText value="Nombre de Contacto"/>
            <c:if test="#{required}">*</c:if>:
            <h:inputText id = "nombre" required = "#{required}" value = "#{value}"
                         requiredMessage = "El valor es necesario!" />
            <h:message for="nombre" styleClass="error"/>
        </ui:component>
    </h:body>
</html>
```

custom_component.xhtml:

```
<h:form>
            <curso:inputEntry label="Nombre de Contacto" required = "true"
                              value = "#{Usuario.nombre}"
                              errorStyleClass ="error" requiredMessage = "El valor es necesario!"
                              />
            <f:verbatim><br/></f:verbatim>
            <h:commandButton value="Probar" type="submit"></h:commandButton>
</h:form>
```

box_doble.xhtml:

```
        <h:panelGrid border="1" columns="2">
            <ui:insert name="izquierdo"/>
            <ui:insert name="derecho"/>
        </h:panelGrid>
```

Uso de box_doble:

```
<curso:box_doble>
                    <ui:define name="derecho">Derecho</ui:define>
                    <ui:define name="izquierdo">Izquierdo</ui:define>
</curso:box_doble>
```


10Componentes JSF para Ajax

10.1¿Cómo JSF y Ajax pueden trabajar juntos?

- Utilizar el (Ajax) JavaScript en su página JSF
- Poner su código de Ajax JavaScript en los componentes JSF
- Aprovechar la creciente colección de marcos de trabajo Ajax JSF
- Utilizar el nuevo tag <f:ajax>

10.2Utilizar el (Ajax) JavaScript en su página JSF

El más práctico / forma fácil de inyectar algo de funcionalidad Web 2.0 en su interfaz de usuario JSF,

Puedes agregar tus propios JS, o extender los marcos de trabajo Ajax existentes como:

- Dojo
- JQuery
- Prototype
- Scriptaculous
- OAT
- …

10.2.1 Pros

- Técnicamente sencillo
- Fácil para cosas simples / pequeñas

10.2.2 Contras

- Trabajo con JavaScript más avanzado puede ser un reto para los programadores Java (usted tiene que pasar código (javascript) al lado del cliente).
- Puede ser difícil depurar la aplicación (busque errores tanto en el servidor y el cliente)

10.3Poner su código de Ajax JavaScript en los componentes JSF

Mediante el uso de la tecnología JSF de componentes es posible escribir sus propios componentes que generan el JavaScript necesario en el cliente.

Esto es factible porque JSF puede generar cualquier tipo de texto en el cliente

```
/ / Javascript Render personalizado para llamar a JSSlider.js
writer.Write ( "<script language=\"JavaScript\" src=\"" contextPath + + "/js/JSSlider.js\"> </
script>");
```

10.3.1 Pros

- Arquitectura sólida
- Usuario final / desarrollador de aplicaciones lo puede utilizar fácilmente – sin tratar con JS.

10.3.2 Contras

- Puede ser un reto para construir desde un punto de vista técnico (tienes que ser un experto cliente Ajax, así como un experto en JSF)

- Difíciles de depurar en el desarrollo de componentes

10.4 Aprovechar la creciente colección de marcos de trabajo Ajax JSF

Ahora pueden descargar diversas bibliotecas de componentes JSF
- Instalación fácil y rápida
- Soporte IDE también está disponible para muchos (Eclipse, NetBeans, Exadel, JDeveloper ...)
- Algunos ejemplos: JBoss RichFaces (Ajax4JSF), ICEfaces, ADF Faces (Trinidad) Rich Client

10.4.1 Pros

- Aprendizaje más fácil
- No tienes que lidiar con las complejidades del JavaScript
- Pueden elegir entre una creciente colección de las bibliotecas

10.4.2 Contras

- Si todo funciona, ¡excelente!
- Todavía puede ser difícil de depurar en el cliente (esto es porque el código de cliente se representa desde el cliente, y usted tiene poco o ningún control sobre este)

10.5 Utilizar el nuevo tag <f:ajax>

JSF 2.0 trae una nueva librería para Ajax.

10.5.1 Pros

- Incluida en JSF 2.0
- No tienes que lidiar con las complejidades del JavaScript

10.5.2 Contras

- Existen limitaciones al uso de h: outputText con Ajax
- Tecnología muy nueva (-->menos fuentes para aprendizaje autodidacto, depuración), solo JSF 2.0
- Todavía puede ser difícil de depurar en el cliente (esto es porque el código de cliente se representa desde el cliente, y usted tiene poco o ningún control sobre este)

10.5.3 ¿Por qué el Ajax en JSF?

¿Por qué una biblioteca de JSF Ajax específico?
Hay toneladas de las bibliotecas Ajax ya (jQuery, DWR, GWT, etc.) ¿Por qué inventar una nueva para JSF?

10.5.3.1 Ventajas de un enfoque específico JSF Ajax

JSF específico
Del lado del cliente:
- Usted puede actualizar los elementos JSF (h: outputText, h: inputText, h: selectOneMenu, etc).
- Usted no tiene que escribir JavaScript

Del lado del servidor
- Los backed beans están disponible en las llamadas Ajax
- Usted no tiene que escribir servlets y analizar parámetros

10.5.4 Uso del tag <f:ajax>

Descripción general:

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

Render: especificar los elementos a actualizar en el cliente
Execute: especificar elementos para procesar en el servidor
Event: especificar los eventos de usuario que inician la llamada Ajax
onEvent: especificar los scripts secundarios (JavaScript) a iniciar la llamada Ajax

Backed bean utilizado en los ejemplos abajo:

```
package curso.beans;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name = "AjaxDemo")
@SessionScoped
public class AjaxDemo {

    private Integer a = 0;
    private int b=0;
    private String respuesta;

    public AjaxDemo() {
    }
    public void onClick() {
        a += 10;
    }
    public void execute() {
        double rand = Math.random();
        respuesta = "El numero al hazar es " + rand + ", multiplicado por " + a + " = " + (rand * a);
    }
    public Integer getA() {
        return a;
    }
    public void setA(Integer a) {
        this.a = a;
    }
    public String getRespuesta() {
        return respuesta;
    }
    public void setRespuesta(String respuesta) {
        this.respuesta = respuesta;
    }
    public int getB() {
        b+=10;
        return b;
    }
    public void setB(int b) {
        this.b = b;
    }
}
```

10.5.4.1 Ejemplo sencillo

Código:

```
<h:commandButton … action="…">
<f:ajax render="id1"/>
</h:commandButton>
/h:commandButton
…
<h:outputText … id="id1"/>
```

Interpretación:

Cuando se pulsa el botón, vaya al servidor, ejecute la acción, calcular el valor del elemento de JSF, cuyo id es "id1", enviar ese valor al cliente, y luego sustituir ese elemento en el DOM con el nuevo valor.

Ejemplo:

```
        <h:form prependId="false" id="form">
            <h1>Ajax demo 1</h1>
            <h:commandButton action="#{AjaxDemo.onClick}" value="Agrega 10 a A">
            <f:ajax render="id1" />
            </h:commandButton>
            <br/>
            <h3>Valor de A:
            <h:outputText id="id1" value="#{AjaxDemo.a}" />
            </h3>
        </h:form>
```

10.5.4.2 renderer: Especificación de los elementos de actualización en el cliente

Resumen de Código

```
<f:ajax render="formId:elementId" />
```

Idea

Id o lista separada por espacios de las identificaciones de los elementos de JSF cuyos valores separados por espacios deben ser devueltos desde el servidor y se sustituye en DOM

Detalles

Hay cuatro valores especiales: @ este @ formulario, ninguno @ y @ todos. Sin embargo, estos son más frecuentemente utilizados para la ejecución de atributos que la hacen atributo. Ver sección del execute.

A menos que utilice <h:form prependId="false"/>, el REAL ID es FormID: elementId.

10.5.4.3 execute: Especificación de los elementos a procesar en servidor

Resumen de Código

```
<f:ajax render="…" execute="…"… />
```

Idea

Un identificador o lista de identificadores JSF separada por espacio que debe ser enviada al servidor para su ejecución. Pro ejemplo, un h: inputText, que será procesado normalmente (setters, validación, etc).

Detalles

Hay 4 valores especiales: @this @form, @none @all

- @this. El elemento que encierra f:ajax. Predeterminado.
- @form. El h:form incluyendo a f: ajax. Muy conveniente si usted tiene varios campos para enviar.
- @none. Nada de lo enviado. Útil si el elemento que cambia los valores de hacer cada vez que se evalúen.
- @all. Todos los elementos de la interfaz de usuario JSF en la página.

Ejemplo:

```
<h:form>
            <h:panelGrid columns="2">
                Numero a multiplicar :
                <h:inputText  id="a"  size="5" value="#{AjaxDemo.a}">
                </h:inputText>
                <h:commandButton action="#{AjaxDemo.execute}" value="Clic">
                    <f:ajax render="respuesta" execute="@form"></f:ajax>
                </h:commandButton>
            </h:panelGrid>
        <hr></hr>
        <h:outputText id="respuesta" value="#{AjaxDemo.respuesta}" />
 </h:form>
```


10.5.4.4 event: mencionar a cual evento de usuario debe disparar la llamada Ajax

Resumen de Código

```
<f:ajax Render="…" event="…"… />
```

Idea

Nombre del evento Ju JavaScript para responder. No incluya "on", por lo que es mouseover, keyup, Blur, etc...

Detalles

Por defecto

Si el evento no especificado, por defecto utilizado.

Alto nivel de eventos

JSF añade 2 Extras: accion y valueChange.

Ajuste f:ajax en torno a elementos

<f:ajax render="…"> un montón de componentes </ f: Ajax> añade el comportamiento Ajax en el evento por defecto para cada componente envolvió.

Acciones por defecto:

action: h:CommandButton, h:commandLink

- Tenga en cuenta que la "acción" es parte de JSF, y no un nombre de Javascript nativo evento. Significa que el botón ha sido invocada en cualquier forma (haciendo clic en él, ENTER, si tiene el foco, el teclado, acceso directo, etc).

valueChange: h:inputText, h:inputSecret, h:inputTextarea, todos los botones de opción, casilla de verificación, y los elementos de menú (h:selectOneMenu, etc),

- Una vez más, este evento es añadido por JSF y no es un nombre Javascript nativo evento. Diferentes navegadores manejan un "cambio" de manera diferente, por lo que este unifica el comportamiento.
- También tenga en cuenta que es "valueChange" no ", valuechange". Los acontecimientos nativas JavaScript son minúsculas (mouseover, keyup, etc)

Ejemplo:

```
<h:form prependId="false">
    <h1>Ajax Event</h1>
    <h:commandButton action="#{AjaxDemo.onClick}" value="Agrega 10 a B">
        <f:ajax render="id1" event="mouseover" />
    </h:commandButton>
    <br/>
    <h3>Valor de B:
    <h:outputText id="id1" value="#{AjaxDemo.b}" />
    </h3>
</h:form>
```


11Hibernate

11.1Introducción

Hibernate es un marco de trabajo para persistencia de datos (en Java).

Hibernate permite manejar la persistencia de los datos en Java, haciendo abstracción de la base de datos. Hibernate se encargará de dar los datos al código Java, y se encargará de mantener la base de datos al día, si se utiliza una base de datos. Es una implementación de JPA.

11.2Instalación

Hibernate viene preinstalado en los servidores Jboss y Glassfish.

11.3Configuración

Para poder usar Hibernate, se requiere configurar un archivo llamado hibernate.cfg.xml que se encuentra en WEB-INF/clases (→ en la raíz de las clases Java).

Aquí va un ejemplo de configuración para conectarse con una base de datos MySQL:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
                                        "http://www.hibernate.org/dtd/hibernate-configuration-
3.0.dtd">
<hibernate-configuration>
 <session-factory name="curso">
  <property name="hibernate.connection.driver_class">org.gjt.mm.mysql.Driver</property>
  <property name="hibernate.connection.url">jdbc:mysql://localhost/curso</property>
  <property name="hibernate.connection.username">curso</property>
  <property name="hibernate.connection.password">123</property>
  <property name="hibernate.dialect">org.hibernate.dialect.MySQL5InnoDBDialect</property>
  <!--  thread is the short name for
      org.hibernate.context.ThreadLocalSessionContext
      and let Hibernate bind the session automatically to the thread
    -->
  <property name="current_session_context_class">thread</property>
  <!-- this will show us all sql statements -->
  <property name="hibernate.show_sql">true</property>
  <property name="hibernate.connection.autocommit">true</property>
 </session-factory>
</hibernate-configuration>
```

11.4Utilización

Para utilizar Hibernate necesitamos crear un Entity EJB3 y referenciarlo en el archivo de configuración de Hibernate.

Existen herramientas para crear las clases a partir de la base de datos.

11.4.1 EJB3 para la table city

City.java:

```
package curso.hibernate;

import java.io.Serializable;
import javax.persistence.*;
import java.util.Set;

@Entity
@Table(name="city")
```



```
public class City implements Serializable {
	private static final long serialVersionUID = 1L;

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	@Column(name="NO_CITY")
	private int noCity;

	@Column(name="CITY_NAME")
	private String cityName;

	@Column(name="REGION")
	private String region;

	//bi-directional many-to-one association to Patient
	@OneToMany(mappedBy="city1")
	private Set<Patient> patients1;

	//bi-directional many-to-one association to Patient
	@OneToMany(mappedBy="city2")
	private Set<Patient> patients2;

    public City() {
    }
	public int getNoCity() {
		return this.noCity;
	}
	public void setNoCity(int noCity) {
		this.noCity = noCity;
	}
	public String getCityName() {
		return this.cityName;
	}
	public void setCityName(String cityName) {
		this.cityName = cityName;
	}
	public String getRegion() {
		return this.region;
	}
	public void setRegion(String region) {
		this.region = region;
	}
	public Set<Patient> getPatients1() {
		return this.patients1;
	}
	public void setPatients1(Set<Patient> patients1) {
		this.patients1 = patients1;
	}
	public Set<Patient> getPatients2() {
		return this.patients2;
	}
	public void setPatients2(Set<Patient> patients2) {
		this.patients2 = patients2;
	}
}
```

Luego agregamos una línea en hibernate.cfg.xml:

```
...
 <session-factory name="curso">
...
        <mapping class="curso.hibernate.City"  />
 </session-factory>
...
```

11.4.2 HibernateHelper

Para recuperar y manejar la sesión de Hibernate, se necesita una clase, como por ejemplo la clase HibernateHelper.

HibernateHelper.java:

```
package curso.hibernate;
```

```
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class HibernateHelper {
	/**
	 * Reference to SessionFactory.
	 */
	private static SessionFactory sf;
	public static final ThreadLocal session = new ThreadLocal();
	public static synchronized void init() {
		if (sf != null) return;
		System.out.println("Initializing Hibernate");
		try {
			Configuration cfg = new Configuration().configure();
			sf = cfg.buildSessionFactory();
		} catch (Exception he) {
			System.err.println("Unable to create session factory from configuration");
				he.printStackTrace();
				throw new RuntimeException("Unable to create session factory
from configuration", he);
		}
		System.out.println("Hibernate initialized");
	}
	/**
	 * Return the SessionFactory.
	 * @return The SessionFactory for this application session
	 */
	public static SessionFactory sessionFactory() {
		if (sf == null) init();
		return sf;
	}
	public static void destroy() {
		if (sf != null) {
			try {
				sf.close();
			} catch (HibernateException he) {
				he.printStackTrace();
			}
		}
		sf = null;
		System.out.println("Hibernate resources released");
	}
	/**
	 * Closes an hibernate {@link Session}, releasing its resources.
	 * @throws HibernateException if an hibernate error occurs
	 */
	public static void closeSession() throws HibernateException {
		Session s = (Session)session.get();
		session.set(null);
		if (s != null) {
			s.close();
		}
	}
	/**
	 * Returns an hibernate {@link Session} from the session factory.
	 * @return an hibernate {@link Session}
	 * @throws HibernateException if an error occurs
	 */
	public static Session openSession() throws HibernateException {
		if (sf == null) init();
		Session s = (Session)session.get();
		if (s == null) {
			s = sf.openSession();
			session.set(s);
		}
		return (s);
	}
}
```

11.4.3 Prueba de uso de Hibernate desde Java

Vamos a conectarnos desde Java y utilizar Hibernate para recuperar datos en la base de datos.

TestExample.java:

```
package curso.hibernate;
import java.util.Iterator;
import java.util.List;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class TestExample {
	final static Logger logger = LoggerFactory.getLogger(TestExample.class);
	public static void main(String[] args) {
		City forestHoney = new City();
		forestHoney.setCityName("forest city");
		forestHoney.setRegion("very sweet");
		City countryHoney = new City();
		countryHoney.setCityName("country city");
		countryHoney.setRegion("tasty");
		createCity(forestHoney);
		createCity(countryHoney);
		System.out.println("listing...");
		listCity();
		deleteCity(countryHoney);
		listCity();
		forestHoney.setCityName("Norther Forest City");
		updateCity(forestHoney);
		HibernateHelper.closeSession();
	}
	private static void listCity() {
		Transaction tx = null;
		System.out.println("listing1...");
		Session session = HibernateHelper.openSession();
		try {
			System.out.println("listing.2..");
			tx = session.beginTransaction();
			List citys = session.createQuery("select h from curso.hibernate.City as h")
			.list();
			System.out.println("listing.3..");
			System.out.println(citys.size());
			for (Iterator iter = citys.iterator(); iter.hasNext();) {
				City element = (City) iter.next();
		    System.out.println("City = " + element.getCityName());
			}
			tx.commit();
		} catch (RuntimeException e) {
			if (tx != null && tx.isActive()) {
				try {
					// Second try catch as the rollback could fail as well
					tx.rollback();
				} catch (HibernateException e1) {
					logger.debug("Error rolling back transaction");
				}
				// throw again the first exception
				throw e;
			}
		}
	}
	private static void deleteCity(City city) {
		Transaction tx = null;
		Session session = HibernateHelper.openSession();
		try {
			tx = session.beginTransaction();
			session.delete(city);
			tx.commit();
		} catch (RuntimeException e) {
			if (tx != null && tx.isActive()) {
				try {
					// Second try catch as the rollback could fail as well
					tx.rollback();
				} catch (HibernateException e1) {
					logger.debug("Error rolling back transaction");
				}
				// throw again the first exception
				throw e;
```

```
                        }
                }
        }
        private static void createCity(City city) {
                Transaction tx = null;
                Session session = HibernateHelper.openSession();
                try {
                        tx = session.beginTransaction();
                        session.save(city);
                        System.out.println("created...");
                        tx.commit();
                } catch (RuntimeException e) {
                        if (tx != null && tx.isActive()) {
                                try {
                                        // Second try catch as the rollback could fail as well
                                        tx.rollback();
                                } catch (HibernateException e1) {
                                        logger.debug("Error rolling back transaction");
                                }
                                // throw again the first exception
                                throw e;
                        }
                }
        }
        private static void updateCity(City city) {
                Transaction tx = null;
                Session session = HibernateHelper.openSession();
                try {
                        tx = session.beginTransaction();
                        session.update(city);
                        tx.commit();
                } catch (RuntimeException e) {
                        if (tx != null && tx.isActive()) {
                                try {
                                        // Second try catch as the rollback could fail as well
                                        tx.rollback();
                                } catch (HibernateException e1) {
                                        logger.debug("Error rolling back transaction");
                                }
                                // throw again the first exception
                                throw e;
                        }
                }
        }
}
```

11.5 Hibernate Query Language

Hibernate utiliza un lenguaje de consulta potente (HQL) que se parece a SQL. Sin embargo, comparado con SQL, HQL es completamente orientado a objetos y comprende nociones como herencia, polimorfismo y asociación.

Para obtener más información sobre el uso del HQL, ver en
http://docs.jboss.org/hibernate/core/3.6/reference/es-ES/html/queryhql.html

11.6 Hibernate con JSF

Para usar Hibernate con JSF, vamos a crear un objeto DAO para acceder a la clase City. Luego podremos manejar el objeto desde JSF.

11.6.1 Clase DAO

Generico.java:

```
package dao;
/**
*
* @author JavAdicto
```



```
*/
import java.util.List;

public interface Generico {
   public void guardar(Object obj);
   public void actualizar( Object obj);
   public void borrar(Object obj);
   public void borrarPorId(Class<?> clase, java.io.Serializable id);
   public Object buscarPorId( Class<?> clase, java.io.Serializable id);
   public List buscarTodos(Class clase);
   public void inicializar(Object obj);
   public void inicializarColeccion(java.util.Collection coleccion);
   public void guardarColeccion(java.util.Collection coleccion);
   public void flush();
   public void merge(Object obj);
}
```

CityDao.java:

```
package dao;
import java.io.Serializable;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;
import org.hibernate.Hibernate;
import org.hibernate.Query;
import org.hibernate.Session;
import curso.hibernate.City;
import curso.hibernate.HibernateHelper;

public class CityDao implements Generico {
        Session session;
        public CityDao() {
         //session = SessionFactoryUtil.getInstance().getCurrentSession();
                session=HibernateHelper.openSession();
        }
        public List  buscarTodos(Class clase){
                List<City> cityList;
         session.beginTransaction();
         Query query = session.createQuery("from "+clase.getSimpleName());
         cityList = query.list();
         session.getTransaction().commit();
         return cityList;
        }
        @Override
        public void actualizar(Object obj) {
         session.beginTransaction();
                session.refresh(obj);
         session.getTransaction().commit();

        }
        @Override
        public void borrar(Object obj) {
         session.beginTransaction();
                session.delete(obj);
         session.getTransaction().commit();
        }
        @Override
        public void borrarPorId(Class<?> clase, Serializable id) {
         session.beginTransaction();
         Object obj = buscarPorId(clase, id);
                session.delete(obj);
         session.getTransaction().commit();
        }
        @Override
        public Object buscarPorId(Class<?> clase, Serializable id) {
         Object instance = null;
         try
         {
             session.beginTransaction();
             instance = session.get(clase, id);
             session.getTransaction().commit();
         }
         catch( RuntimeException e )
         {
             e.printStackTrace();
```

```
        }
        return instance;
        }
        @Override
        public void flush() {
                session.flush();
        }
        public Object find (Class clase, int id){
        session.beginTransaction();
                Object obj = session.load(clase, id);
        session.getTransaction().commit();
                return obj;
        }
        @Override
        public void guardar(Object obj) {
                try
        {
                 session.beginTransaction();
           session.saveOrUpdate(obj);
           session.getTransaction().commit();
        }
        catch( RuntimeException e )
        {
            e.printStackTrace();
        }
        }
        @Override
        public void guardarColeccion(Collection coleccion) {
        session.beginTransaction();
                session.saveOrUpdate(coleccion);
        session.getTransaction().commit();
        }
        @Override
        public void inicializar(Object obj) {
                Hibernate.initialize(obj);
        }
        @Override
        public void inicializarColeccion(Collection coleccion) {
                try
        {
            Hibernate.initialize(coleccion);
            Iterator it=coleccion.iterator();
            while(it.hasNext())
            {
                Hibernate.initialize(it.next());
            }
        }
        catch( RuntimeException e )
        {
            e.printStackTrace();
        }
        }
        @Override
        public void merge(Object obj) {
        session.beginTransaction();
                session.merge(obj);
        session.getTransaction().commit();
        }
}
```

11.6.2 Acceso desde JSF

CityList1.java:

```
package curso.beans;
import java.util.List;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.bean.ViewScoped;
import curso.hibernate.City;
import dao.CityDao;

@ManagedBean(name="cityList1")
@ViewScoped
public class CityList1 {
```

```
	private List<City> all;

	public List<City> getAll() {
		this.all = new CityDao().buscarTodos(City.class);
		return all;
	}
	public String initCity(){
		String outcome = "details";
		return outcome;
	}
}
```

city_list.xhtml:

```
<!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"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets">

<h:head></h:head>
<body>
<h:form>
<h:dataTable id="dt1" value="#{cityList1.all}"
var="item" bgcolor="#F1F1F1" border="10" cellpadding="5" cellspacing="3" rows="4" width="50%"
dir="ltr" frame="hsides"
rules="all" summary="Es un codigo JSF para crear un dataTable." >
<h:column >
<f:facet name="header">
<h:outputText value="Cuidad" />
</f:facet>
<h:commandLink value="#{item.cityName}" action="#{cityList1.initCity}" >
<f:setPropertyActionListener value="#{item}"
target="#{cityDetails.city}"></f:setPropertyActionListener>
</h:commandLink>
</h:column>
<h:column>
<f:facet name="header">
<h:outputText value="Region"/>
</f:facet>
<h:outputText value="#{item.region}"></h:outputText>
</h:column>
</h:dataTable><br />
</h:form>
</body>
</html>
```

CityDetails.java:

```
package curso.beans;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
import javax.faces.event.AbortProcessingException;
import javax.faces.event.ActionEvent;
import curso.hibernate.City;
import dao.CityDao;

@ManagedBean(name="cityDetails")
@RequestScoped
public class CityDetails {
	private City city;
	private String cityName;
	private int noCity;
	private CityDao dao;

	public CityDetails() {
		dao=new CityDao();
	}
	public void onSubmit(ActionEvent e) {
		System.out.println("submiting !!!");
		if (cityName.equalsIgnoreCase("")) {
			throw new AbortProcessingException("Cuidad invalida");
		}
	}
	public String submit(){
		System.out.println("Updating !!! noCity="+noCity);
		city=(City) dao.find(City.class, noCity);
```



```
                city.setCityName(cityName);
                dao.guardar(city);
                System.out.println("Updated !!! + "+cityName);
                System.out.println("submit2222 !!!");
                String outcome = "success";
                return outcome;
        }
        public City getCity() {
                System.out.println("ZZZZZ - "+city.getCityName());
                return city;
        }
        public void setCity(City city) {
                System.out.println("XXXXXXXXXX");
                this.city = city;
        }
        public String getCityName() {
                if (city!=null)
                cityName=city.getCityName();
                System.out.println("AAA "+cityName);
                //System.out.println("AAA "+city.getCityName());
                return cityName;
        }
        public void setCityName(String cityName) {
                System.out.println("bbb "+cityName);
                this.cityName = cityName;
        }
        public int getNoCity() {
                if (city!=null)
                this.noCity=city.getNoCity();
                System.out.println("ccc "+noCity);
                return noCity;
        }
        public void setNoCity(int noCity) {
                System.out.println("ddd "+noCity);
                this.noCity = noCity;
        }
}
```

CityDetails.xhtml:

```
<!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:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">

   <h:head></h:head>
   <h:body>
   <h:form >
   <h:inputHidden value="#{cityDetails.noCity}"  id="noCity"/>
   <h1>Detalle de la cuidad de <h:outputText value="#{cityDetails.cityName}" /></h1>
   <h:panelGrid bgcolor="yellow" columns="1">
   <h:column><h:inputText value="#{cityDetails.cityName}" autocomplete="off" id="cityName"
   required="true" validatorMessage="Minimo 2 caracteres!"
   requiredMessage="La cuidad no puede estar vacia"  >
   <f:validateLength minimum="2" /></h:inputText>
   </h:column>
   <h:column><h:commandButton value="Enviar" action="#{cityDetails.submit}"
actionListener="#{cityDetails.onSubmit}" /> </h:column>
   </h:panelGrid>
   </h:form>
   </h:body>
</html>
```

Agregar a faces-config.xml:

…

```
 <navigation-rule>
  <from-view-id>/cityDetails.xhtml</from-view-id>
  <navigation-case>
   <from-outcome>success</from-outcome>
   <to-view-id>/city_list.xhtml</to-view-id>
  </navigation-case>
 </navigation-rule>
```



```
 <navigation-rule>
  <from-view-id>/city_list.xhtml</from-view-id>
  <navigation-case>
   <from-outcome>details</from-outcome>
   <to-view-id>/cityDetails.xhtml</to-view-id>
  </navigation-case>
 </navigation-rule>
...
```

