

Capítulo 3. JavaServer Faces

3.1. Introducción

JavaServer Faces (JSF) es el framework para aplicaciones Web en Java de Sun Microsystems, liberado apenas en Marzo del 2004, que busca tomar su lugar como estándar entre los muchos de su clase.

JSF es un framework orientado a la interfaz gráfica de usuario (GUI), facilitando el desarrollo de éstas, y que sin embargo, realiza una separación entre comportamiento y presentación, además de proporcionar su propio servlet como controlador, implementando así los principios del patrón de diseño Model-View-Controller (MVC), lo que da como resultado un desarrollo más simple y una aplicación mejor estructurada.

El enfoque mencionado anteriormente no es nada nuevo. Lo que hace a JSF tan atractivo, entre muchas otras cosas más, es que brinda un modelo basado en componentes y dirigido por eventos para el desarrollo de aplicaciones Web, que es similar al modelo usado en aplicaciones GUI *standalone* durante años [Bergsten, 2004], como es el caso de Swing, el framework estándar para interfaces gráficas de Java.

Otra característica muy importante de JavaServer Faces es que, a pesar de que HTML es su lenguaje de marcado por *default*, no está limitado a éste ni a ningún otro, pues tiene la capacidad de utilizar diferentes *renderers* para los componentes GUI y obtener así diferentes salidas para mandar al cliente [Bergsten, 2004]. Así mismo, JSF es suficientemente flexible para soportar diversas tecnologías de presentación [Bergsten, 2004], destacando entre éstas JSP, ya que es una tecnología soportada, requerida y especificada para toda implementación de JavaServer Faces.

3.1.1. Ciclo de vida

Otro aspecto muy importante dentro de JavaServer Faces es su ciclo de vida, el cual es similar al de una página JSP: el cliente hace una petición HTTP y el servidor responde con la página en HTML. Sin embargo, debido a las características que ofrece JSF, el ciclo de vida incluye algunos pasos más [Ball, 2003].

El proceso de una petición estándar incluye seis fases, como se muestra en la Figura 3.1.1, representadas por los rectángulos blancos:

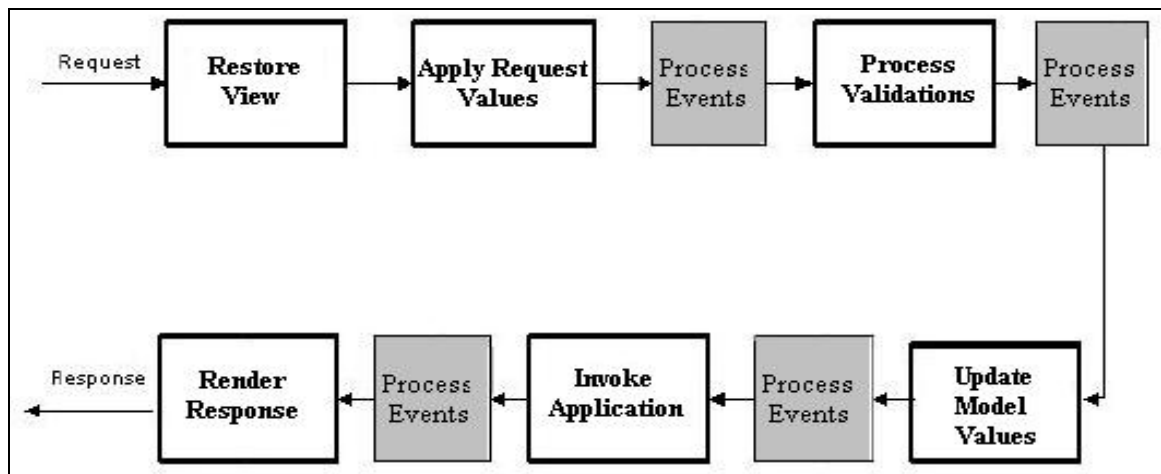


Figura 3.1.1 Ciclo de Vida de JavaServer Faces

Los rectángulos grises etiquetados con la leyenda “*Process Events*” representan la ejecución de cualquier evento producido durante el ciclo de vida.

El funcionamiento de cada etapa se describe brevemente a continuación.

1. *Restore View*: también llamada *Reconstitute Component Tree*, es la primera etapa que se lleva a cabo, e inicia cuando se hace una petición. Su objetivo es la creación de un árbol con todos los componentes de la página en cuestión.
2. *Apply Request Values*: cada uno de los componentes del árbol creado en la fase anterior obtiene el valor que le corresponde de la petición realizada y lo almacena.

3. *Process Validations*: después de almacenar los valores de cada componente, estos son validados según las reglas que se hayan declarado.
4. *Update Model Values*: durante esta fase los valores locales de los componentes son utilizados para actualizar los *beans* que están ligados a dichos componentes [Geary & Geary, 2004]. Esta fase se alcanzará únicamente si todas las validaciones de la etapa anterior fueron exitosas.
5. *Invoke Application*: se ejecuta la acción u operación correspondiente al evento inicial que dio comienzo a todo el proceso.
6. *Render Response*: la respuesta se renderiza y se regresa al cliente.

Dependiendo del éxito o fracaso de las tareas en cada una de las fases del ciclo de vida, el flujo normal descrito puede cambiar hacia caminos alternos según sea el caso.

3.1.2. Estructura de una aplicación JSF

Se necesitan dos cosas para correr aplicaciones con JavaServer Faces: un contenedor Web para Java y una implementación de la especificación de JSF [Bergsten, 2004]. Debido a esto, la estructura del directorio de una aplicación JSF podría verse de la manera como lo muestra la Figura 3.1.2.

Los elementos propios de JavaServer Faces que conforman una aplicación son típicamente los siguientes:

1. Archivos JSP, que constituyen la interfaz gráfica de la aplicación y que contienen las diversas funcionalidades de JSF, como los *tags* que representan los componentes GUI. Un archivo de este tipo puede verse como lo muestra la Figura 3.1.3.

2. Archivos XML, específicamente el archivo `faces-config.xml` que almacena las diferentes configuraciones y elementos a utilizar en la aplicación. Un ejemplo de este archivo se muestra en la Figura 3.1.4.
3. Archivos Java, típicamente desempeñando el rol de *beans*.
4. Archivos de paquetes de mensajes (*message bundles*).

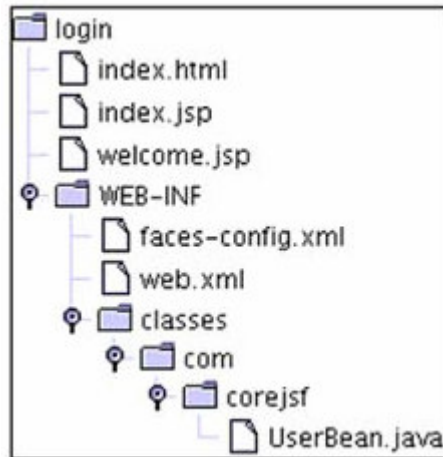
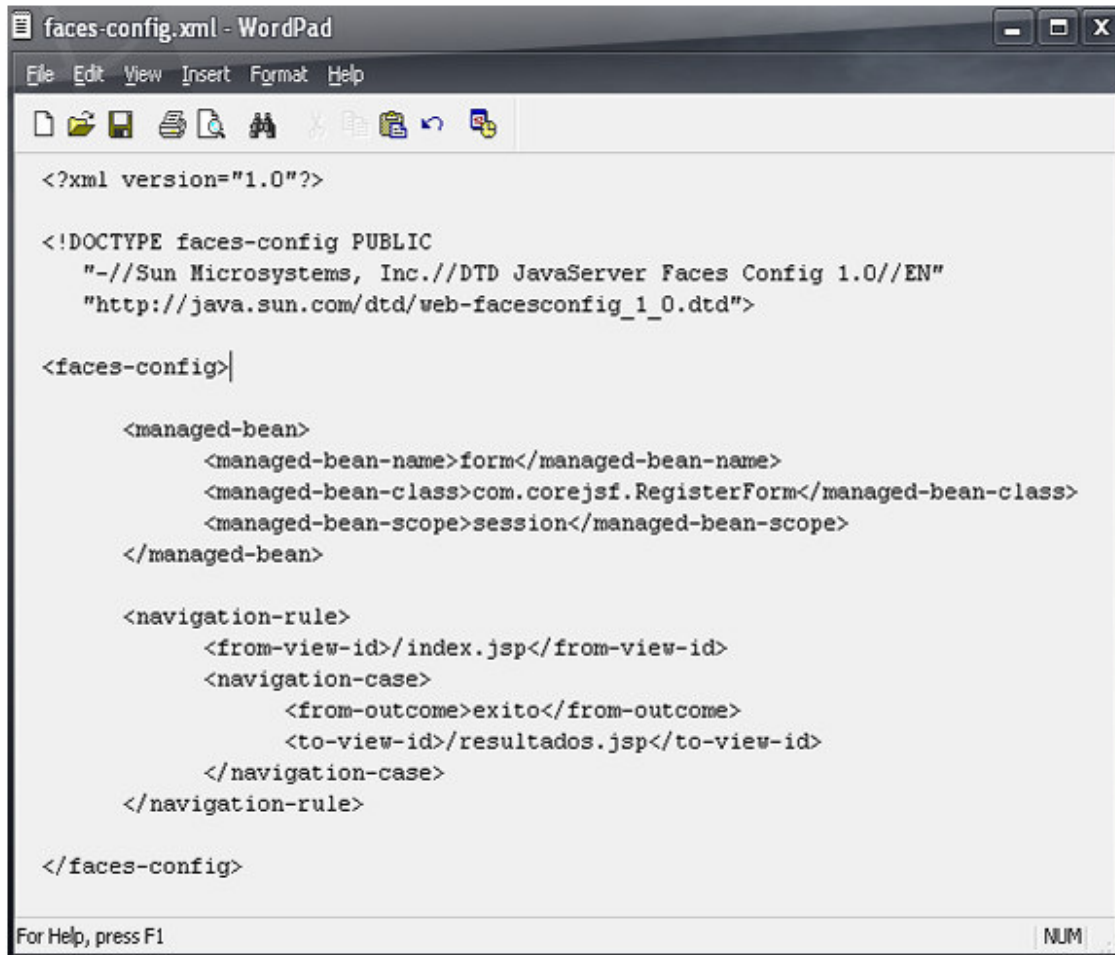


Figura 3.1.2 Estructura de Directorio de una Aplicación JSF



```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>
  <managed-bean>
    <managed-bean-name>form</managed-bean-name>
    <managed-bean-class>com.corejsf.RegisterForm</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
  <navigation-rule>
    <from-view-id>/index.jsp</from-view-id>
    <navigation-case>
      <from-outcome>exito</from-outcome>
      <to-view-id>/resultados.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>
```

Figura 3.1.4 Archivo faces-config.xml

3.2. JavaBeans

De acuerdo a la especificación de JavaBeans, estos se definen como “componentes de software reusables que pueden ser manipulados en una herramienta de desarrollo” [JavaBeans, 2006]. De una manera más simple, un JavaBean se puede ver como una clase Java tradicional o un POJO (Plain Old Java Object) que sigue ciertas especificaciones de programación.

Las características más importantes de un *bean* son las propiedades que ofrece. Una propiedad es cualquier atributo de un bean que tiene:

- un nombre;
- un tipo;
- métodos para obtener (leer) y/o establecer (escribir) el valor de la propiedad.

Enseguida se muestra un ejemplo de un JavaBean, con dos atributos (nombre, apellido) y sus correspondientes métodos para obtener sus valores (getNombre, getApellido) y para establecer nuevos valores (setNombre, setApellido).

```
package com.jsf;
public class BeanUsuario {
    private String nombre;
    private String apellido;

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

    // PROPERTY: apellido
    public String getApellido() { return apellido; }
    public void setApellido(String nuevoValor){ apellido = nuevoValor; }
}
```

En JavaServer Faces se usan *beans* cuando se necesita conectar clases Java con páginas Web; en otras palabras, se utilizan para los datos que necesitan ser accedidos desde una página. Los beans son el medio para conectar la interfaz gráfica con la lógica aplicativa [Geary, 2004].

En JSF los *beans* son utilizados en las páginas que representan la interfaz gráfica, típicamente JSP, específicamente en los componentes. Para poder hacer uso de estos beans, se deben declarar en el archivo `faces-config.xml`. A continuación se muestra el ejemplo para el caso del *bean* presentado anteriormente.

```
<managed-bean>
  <managed-bean-name>usuario</managed-bean-name>
  <managed-bean-class>com.jsf.BeanUsuario</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Lo anterior se puede traducir de la siguiente manera: construye un objeto de la clase `com.jsf.BeanUsuario`, llámalo `usuario` y mantenlo activo mientras dure la sesión.

Una vez declarado el bean, éste puede ser utilizado por los componentes JSF a través de expresiones de referencia de valor, mejor conocidas como *value binding expressions*, y cuya sintaxis es la siguiente:

```
<h:inputText id=nombre value="#{NombreBean.NombrePropiedad}"/>
```

Lo que se logra con estas expresiones es que el valor del componente estará mapeado a la propiedad especificada del bean en cuestión. De esta manera, cuando el componente sea renderizado, se llamará al método *get* respectivo; y cuando la respuesta al usuario sea procesada, se invocará al método *set* [Geary, 2004], de manera automática ambas operaciones. Es decir, el valor del componente será almacenado en la variable de la clase del bean, y así se podrá utilizar para las diversas operaciones que deban realizarse en la lógica aplicativa, y el valor a desplegar en la página Web será a su vez obtenido de la misma fuente (la variable de la clase del bean).

3.3. Paquetes de Mensajes

Mejor conocidos como *message bundles*, estos paquetes agrupan todos los mensajes a desplegar en una página Web, es decir, la parte estática de éstas, con el objetivo de facilitar la internacionalización.

Los mensajes de la aplicación deberán ser guardados en un archivo con el formato “properties” mostrado en la Figura 3.3.1.

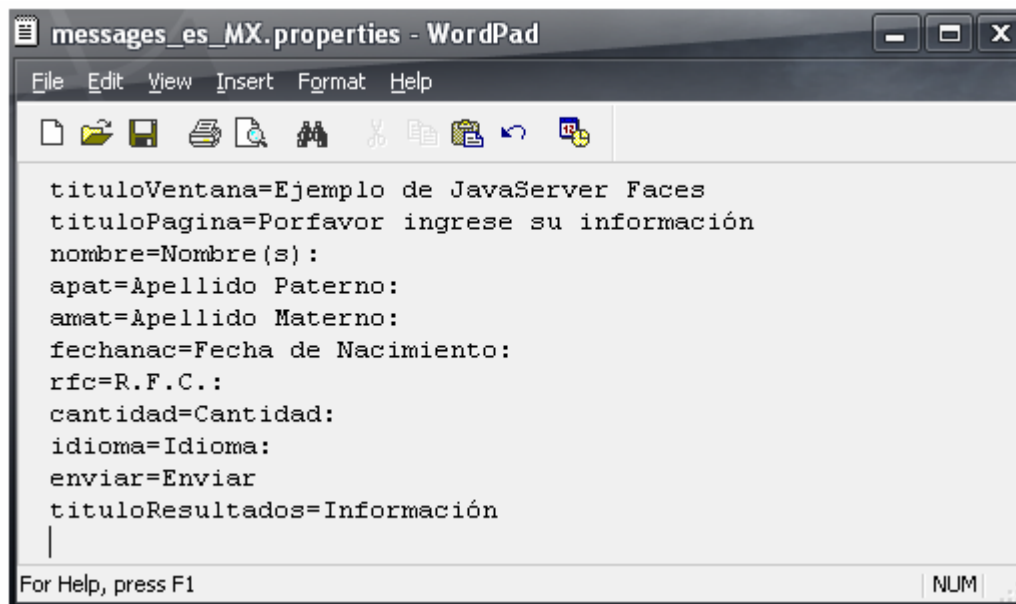


Figura 3.3.1 Archivo .properties

El archivo deberá guardarse en el mismo directorio de las clases para mayor comodidad, aunque puede almacenarse en cualquiera, sin embargo, la extensión tendrá que ser siempre `.properties`.

Para poder hacer uso de los mensajes, se debe cargar el archivo correspondiente en la página JSF a través de la sentencia siguiente:

```
<f:loadBundle basename="com.jsf.messages" var="msgs"/>
```

Una vez cargados los mensajes, estos se podrán utilizar a través de *value binding expressions*, haciendo referencia a la variable en la que se guardaron:

```
<h:outputText value="#{msgs.nombre}"/>
```

De esta manera, todos los mensajes que se quieran desplegar de manera estática, estarán almacenados en un archivo, reduciendo así las modificaciones a nuestras páginas JSF.

La página resultante que hace uso de los ejemplos presentados, se muestra en la Figura 3.3.2.

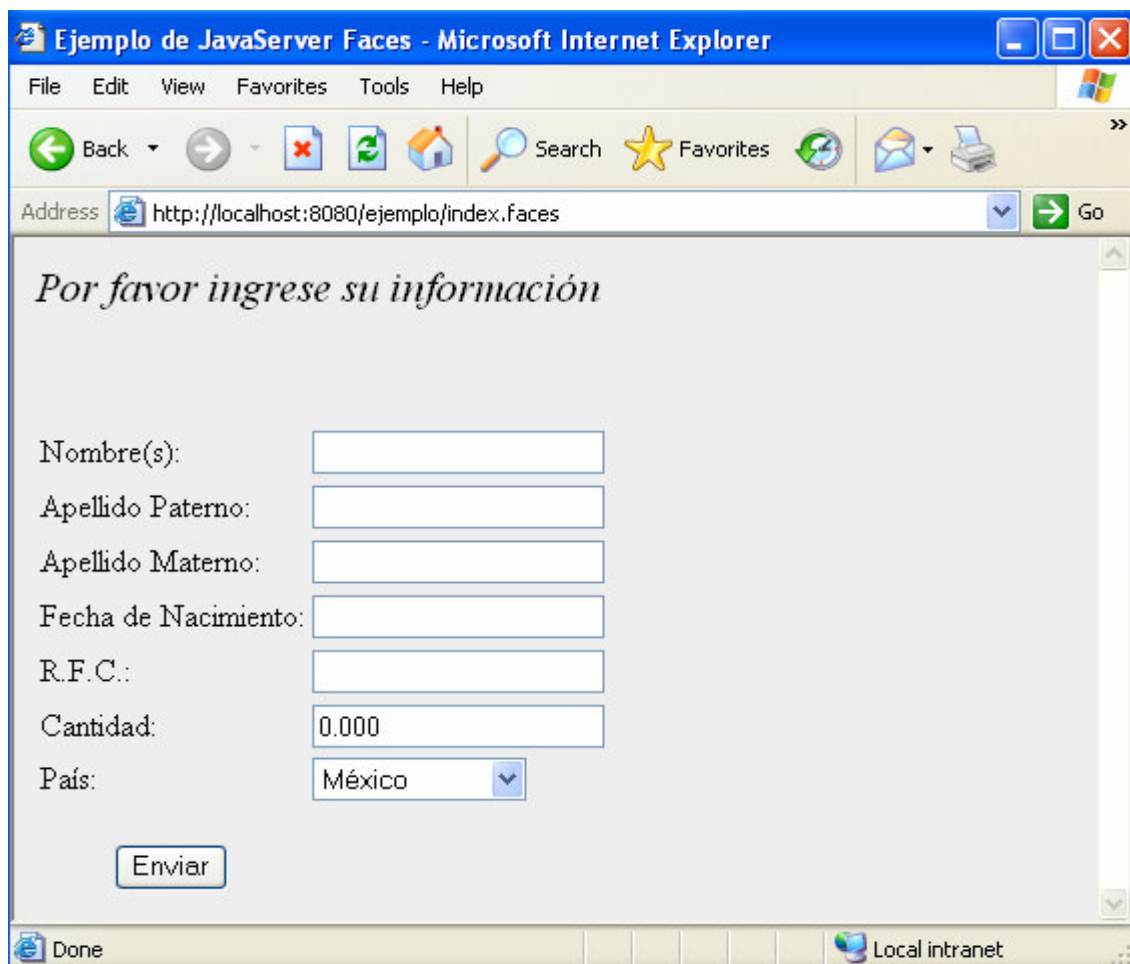


Figura 3.3.2 Página Resultante

3.4. Navegación

La navegación se refiere a la forma en que se desplegarán las diferentes páginas de la aplicación a través de los eventos producidos por el usuario.

La navegación se determina por medio de reglas declaradas en el archivo `faces-config.xml`, que serán evaluadas con la respuesta obtenida del evento generado en la página actual para determinar la siguiente vista a desplegar. Es decir, cuando el usuario hace clic en un botón, y éste origina un *submit* de la forma, la aplicación debe analizar los datos mandados y determinar qué página desplegar a continuación. Para determinar la vista a mostrar, la aplicación toma la respuesta generada por el atributo `action` del componente que originó el evento, y en base a ésta y a las reglas establecidas, selecciona la página adecuada.

A continuación se muestra el uso del atributo `action` en un componente, en este caso un botón:

```
<h:commandButton value="{msgs.enviar}" action="exito" />
```

En este caso se ejemplifica la navegación estática, ya que el valor de `action` es simplemente una cadena de texto que no será afectada por los datos que ingrese el usuario.

Sin embargo esto no es muy útil, ya que lo ideal sería analizar la información proporcionada por el usuario y en base a esto generar una respuesta adecuada para determinar la siguiente página a mostrar, lo que se conoce como navegación dinámica. Por lo tanto, para lograr lo mencionado, en lugar de asignarle una cadena de texto al atributo `action`, se hará referencia a un método de un bean a través de un *value binding expression*, de tal forma que dicho método analice los datos del usuario y genere una respuesta:

```
<h:commandButton value="#{msgs.enviar}" action="#{control.verificar}" />
```

Todo método utilizado con el atributo `action` deberá regresar un *String* y no tener parámetros. El método `verificar` utilizado tendría la siguiente estructura:

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

Como se mencionó anteriormente, las reglas de navegación serán declaradas en el `faces-config.xml`, siguiendo el siguiente formato:

```
<navigation-rule>  
    <from-view-id>/index.jsp</from-view-id>  
    <navigation-case>  
        <from-outcome>login</from-outcome>  
        <to-view-id>/welcome.jsp</to-view-id>  
    </navigation-case>  
</navigation-rule>
```

Este ejemplo se puede traducir de la siguiente manera: si estando en la página `/index.jsp` se obtiene de un evento la respuesta “logout”, se desplegará entonces la página llamada `/logout.jsp`.

Se pueden modificar las reglas y hacer diferentes combinaciones para que se ajusten a nuestras necesidades; por ejemplo, en el ejemplo anterior si se omite la línea `<from-view-id>/index.jsp</from-view-id>`, entonces todas las páginas que generen en su atributo `action` una salida “logout” mandarán el flujo a `/logout.jsp`, ya que no se ha definido ninguna página inicial.

Así mismo, se pueden definir varias reglas para una misma vista inicial, de tal manera que de una misma vista se puedan obtener varias rutas diferentes:

```
<navigation-rule>
  <from-view-id>/index.jsp</from-view-id>
  <navigation-case>
    <from-outcome>login</from-outcome>
    <to-view-id>/welcome.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>signup</from-outcome>
    <to-view-id>/newuser.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Otro elemento importante en la definición de las reglas de navegación es `from-action`, el cual se emplea cuando diferentes métodos utilizados en el atributo `action` arrojan una misma respuesta y se hace necesario identificar cuál de ellos se ha invocado para obtener la página adecuada a desplegar, como se ejemplifica a continuación:

```
<navigation-case>
  <from-action>#{quiz.answerAction}</from-action>
  <from-outcome>again</from-outcome>
  <to-view-id>/again.jsp</to-view-id>
</navigation-case>
<navigation-case>
  <from-action>#{quiz.startOverAction}</from-action>
  <from-outcome>again</from-outcome>
  <to-view-id>/index.jsp</to-view-id>
</navigation-case>
```

En estos casos los *value binding expression* (`#{quiz.answerAction}`, `#{quiz.startOverAction}`) no son invocados, pues ya se han llamado desde el atributo `action`, sino que funcionan a modo de llave para encontrar la regla de navegación adecuada [Geary, 2004].

3.5. Conversión y Validación

Antes de que los datos ingresados por el usuario sean almacenados en los beans correspondientes del modelo, estos deben pasar por dos procesos necesarios de JSF: conversión y validación, siempre en este orden.

La conversión es el proceso a través del cual se transforman los datos ingresados por el usuario en la forma Web en los tipos de datos que mapean en sus correspondientes variables de su bean. Es decir, los datos que proporciona el usuario, manejados como cadenas de texto, se transforman en `int`, `Date` o según sea el tipo de dato que se ha definido para cada uno de ellos en el bean especificado.

La validación es el proceso mediante el cual JSF analiza los datos del usuario de manera semántica, de acuerdo a ciertas reglas especificadas para cada uno de estos, como pueden ser la longitud mínima y máxima para un campo de la forma.

Ambos procesos tienen como objetivo filtrar los datos del usuario, de tal manera que los valores que se guardan en los beans sean coherentes y aceptados por el proceso llevado en la lógica aplicativa.

3.5.1. Conversión estándar

JavaServer Faces proporciona algunos convertidores para los tipos de datos básicos. Así mismo, JSF realiza la conversión de manera automática para los tipos de datos `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`, `BigDecimal`, `BigInteger`, y para sus tipos primitivos, donde apliquen.

JSF permite realizar la conversión a números, independientemente del tipo, pudiendo darles el formato deseado, como cantidad de dígitos en la parte entera o decimal.

Para lograr esto, lo único que debe hacerse es añadir al componente GUI el convertidor adecuado con sus diferentes atributos, según se necesite, como lo muestra el siguiente ejemplo, donde se usa el convertidor `f:convertNumber` y su atributo `minFractionDigits` para determinar el número mínimo de dígitos en la parte decimal:

```
<h:inputText value="#{bean.cantidad}">  
  <f:convertNumber minFractionDigits="2"/>  
</h:inputText>
```

El otro convertidor que proporciona JSF, además de los que realiza de manera automática, se encuentra `f:convertDateTime`, encargado de convertir y formatear datos de tipo `Date`:

```
<h:inputText value="#{bean.fechadate}">  
  <f:convertDateTime pattern="MM/yyyy"/>  
</h:inputText>
```

Así mismo se ha utilizado además su atributo `pattern` para determinar el formato que debe seguir.

Tanto `f:convertNumber` como `f:convertDateTime` disponen de varios atributos con diferentes propósitos para darle formato a los datos deseados; para conocer más acerca de ellos es recomendable consultar la librería de *tags* de JSF, disponible en la página <http://java.sun.com/j2ee/javaserverfaces/1.1/docs/tlddocs/>.

Para dar a conocer los errores que se han generado en el proceso de conversión, se debe añadir al componente GUI en cuestión el *tag* apropiado con el nombre del componente a manera de llave:

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

La página con el mensaje desplegado se presenta en la Figura 3.5.1.

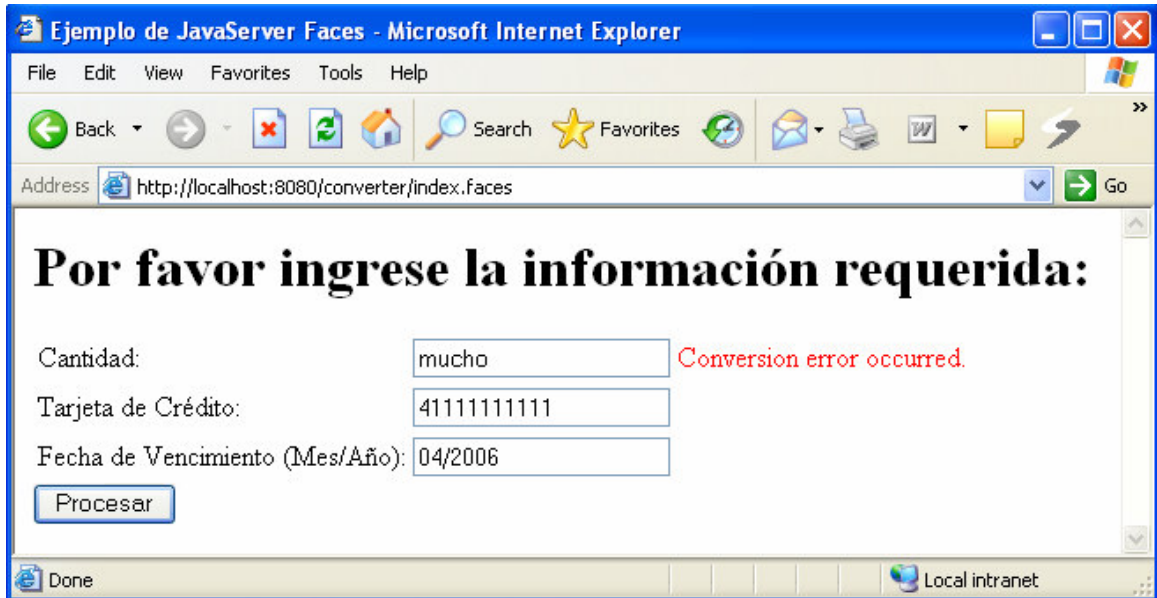


Figura 3.5.1 Mensaje Desplegado Cuando la Conversión Falla

3.5.2. Validación estándar

JavaServer Faces proporciona validación para longitud de cadenas de texto y para rangos de números.

Para el caso de validación de longitud, JSF utiliza el elemento `f:validateLength`:

```
<h:inputText value="#{bean.direccion}">
  <f:validateLength minimum="13"/>
</h:inputText>
```

Para rangos de números hace uso de `f:validateLongRange`:

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



```
</h:inputText>
```

Así mismo, JSF permite realizar validación para campos que son necesarios o requeridos, añadiendo únicamente el atributo `required` al componente GUI deseado:

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

En algunos casos podría desearse que el proceso de validación fuera omitido; por ejemplo, si se agrega un botón “Cancelar” a una página que contiene campos requeridos, al hacer clic en éste, la página mandará un error, pues no se han llenado los campos necesarios [Geary, 2004]. Para solucionar esto, JSF utiliza el atributo *immediate* en el componente GUI deseado con el objetivo de dejar a un lado la validación y pasar a la página requerida por el componente.

```
<h:commandButton value="Cancel" action="cancel" immediate="true"/>
```

Para desplegar los errores de validación se procede exactamente de la misma forma que en la conversión. El ejemplo se muestra en la Figura 3.5.2.

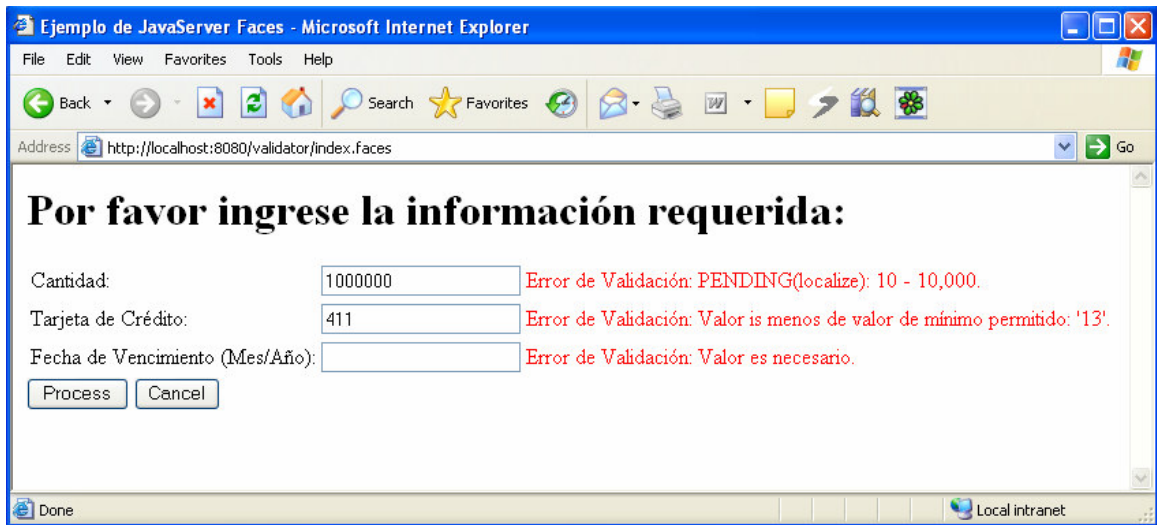


Figura 3.5.2 Mensaje de Error que se Despliega Cuando la Validación Falla

3.5.3. Conversión personalizada

La conversión personalizada se refiere a la creación de clases para que realicen dicho proceso, lo cual puede ser necesario cuando la aplicación desarrollada haga uso de tipos de datos complejos, como clases mismas.

Las clases destinadas a realizar conversiones deberán implementar la interfaz `Converter`, que tiene dos métodos:

- `Object getAsObject(FacesContext context, UIComponent component, String newValue)`
- `String getAsString(FacesContext context, UIComponent component, Object value)`

El primer método convierte una cadena de texto en un objeto del tipo deseado, arrojando una excepción del tipo `ConverterException` si el proceso falla:

```
if (foundInvalidCharacter) {
    FacesMessage message = com.jsf.util.Messages.getMessage(
        "com.jsf.messages", "badCharacter",
        new Object[] { new Character(invalidCharacter) });
```

```
message.setSeverity(FacesMessage.SEVERITY_ERROR);
throw new ConverterException(message);
}
```

El segundo método convierte el objeto en una cadena de texto para que pueda desplegarse al usuario [Geary, 2004].

Una vez creada la clase, se debe registrar en el archivo `faces-config.xml`:

```
<converter>
  <converter-id>com.jsf.Rfc</converter-id>
  <converter-class>com.jsf.RfcConverter</converter-class>
</converter>
```

Y para utilizarla:

```
<h:inputText value="#{bean.rfc}">
  <f:converter converterId="com.jsf.Rfc"/>
</h:inputText>
```

3.5.4. Validación personalizada

Al igual que en la conversión personalizada, en la validación personalizada las clases deberán extender de una interfaz, sólo que ahora ésta será `javax.faces.validator.Validator`, la cual define un único método: `void validate(FacesContext context, UIComponent component)`. Así mismo, si el proceso falla, se arrojará una excepción, ahora del tipo `ValidatorException`:

```
if (validation fails) {
  FacesMessage message = ...;
  message.setSeverity(FacesMessage.SEVERITY_ERROR);
  throw new ValidatorException(message);
}
```

Al igual que en la conversión personalizada, las nuevas clases deberán ser declaradas en el `faces-comfig.xml`:

```
<validator>
```

```
<validator-id>com.jsf.Rfc</validator-id>  
<validator-class>com.jsf.RfcValidator</validator-class>  
</validator>
```

Y para hacer uso de ellas se hará de la misma forma:

```
<h:inputText value="#{bean.rfccard}" required="true">  
  <f:converter converterId="com.jsf.Rfc"/>  
  <f:validator validatorId="com.jsf.Rfc"/>  
</h:inputText>
```

3.6. Manejo de Eventos

Los eventos juegan un papel muy importante dentro de JavaServer Faces, como se ha mencionado. Los componentes GUI de JSF responden a las acciones del usuario disparando eventos, que serán manejados por código de la aplicación, llamados *listeners*, que han sido registrados para ser notificados de la ocurrencia del evento [Bergsten, 2004].

Java Server Faces soporta dos tipos de eventos para los componentes GUI:

- eventos *Value Change* y
- eventos *Action*.

La idea central del manejo de eventos es registrar los *listeners* adecuados para cada componente GUI, de tal manera que se realicen las operaciones apropiadas y deseadas.

3.6.1. Eventos *Value Change*

Estos eventos son generados por componentes de entrada, tales como `h:inputText`, `h:selectManyMenu` y `h:selectOneRadio`, cuando su valor cambia y la forma en que están contenidos es mandada (*submit*).

Para agregar un *listener* del tipo *Value Change*, además de asociarlo con el correspondiente componente GUI y a través de un *value binding expression* indicar el

método que se invocará al dispararse el evento, debe añadirse el atributo `onchange` para forzar que se haga el *submit* de la forma [Geary, 2004], ya que el cambio de valor no realiza esta acción:

```
<h:selectOneMenu value="#{bean.pais }" onchange="submit () "  
    valueChangeListener="#{bean.cambioPais}">  
    <f:selectItems value="#{bean.nombresPaises }"/>  
</h:selectOneMenu>
```

De esta forma, cuando el usuario cambia el valor del componente GUI, se hace un *submit* de la forma y posteriormente se llama al método `countryChanged` del bean `form`; este método debe tener como parámetro una variable `ValueChangeEvent`, con la cual podrá obtener el nuevo valor del componente GUI:

```
private static final String US = "United States";  
...  
public void cambioPais (ValueChangeEvent event) {  
    FacesContext context = FacesContext.getCurrentInstance();  
    if(US.equals((String)event.getNewValue()))  
        context.getViewRoot().setLocale(Locale.US);  
    else  
        context.getViewRoot().setLocale(Locale.CANADA);  
}
```

En el ejemplo anterior el *listener* se ha agregado como un atributo del componente, y el método que invoca pertenece a un bean. Existe otra alternativa que ofrece JSF y consiste en añadir el listener como un *tag*, en lugar de un atributo:

```
<h:selectOneMenu value="#{bean.pais}" onchange="submit () ">  
    <f:valueChangeListener type="com.jsf.PaisListener"/>  
    <f:selectItems value="#{form.nombresPaises }"/>  
</h:selectOneMenu>
```

Y como se puede observar, en este caso se hace referencia a una clase Java, en lugar de un método de un bean, la cual debe implementar la interfaz

ValueChangeListener y definir únicamente el método void processValueChange(ValueChangeEvent):

```
public class PaisListener implements ValueChangeListener {
    private static final String US = "United States";

    public void processValueChange(ValueChangeEvent event) {
        FacesContext context = FacesContext.getCurrentInstance();

        if (US.equals((String) event.getNewValue()))
            context.getViewRoot().setLocale(Locale.US);
        else
            context.getViewRoot().setLocale(Locale.CANADA);
    }
}
```

3.6.2. Eventos *Action*

Estos eventos son disparados por componentes tipo *command*, tales como `h:commandButton` y `h:commandLink`, cuando el botón o el link son activados [Geary, 2004].

A diferencia de los eventos *Value Change*, los eventos *Action* realizan el *submit* de manera automáticamente, por lo cual sólo debe agregarse el atributo `actionListener`, y la referencia al método a invocar, al componente GUI deseado, sin necesidad del atributo `onchange`:

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

Así mismo, el método `linkUsado` debe ahora tener como parámetro una variable de tipo `ActionEvent`.

Para agregar el *listener* a manera de *tag*, se procede de la misma forma que para los eventos *Value Change*, con sus cambios respectivos:

```
<h:commandButton image="mountrushmore.jpg" action="#{rushmore.act}">
    <f:actionListener type="com.jsf.ImagenListener"/>
</h:commandButton>
```

La clase a la que hace referencia, `RushmoreListener`, ahora implementa la interfaz `ActionListener` y el método `void processAction(ActionEvent e)`:

```
public class ImagenListener implements ActionListener {

    public void processAction(ActionEvent e) {
        FacesContext context = FacesContext.getCurrentInstance();
        Map requestParams =
context.getExternalContext().getRequestParameterMap();
        String locale = (String) requestParams.get("locale");

        if ("english".equals(locale))
            context.getViewRoot().setLocale(Locale.UK);
        else if ("german".equals(locale))
            context.getViewRoot().setLocale(Locale.GERMANY);
    }
}
```