Temas avanzados de JAVA SERVER FACES 1.2

Por:
Carmelo Navarro Serna

INDICE

1 Introducción

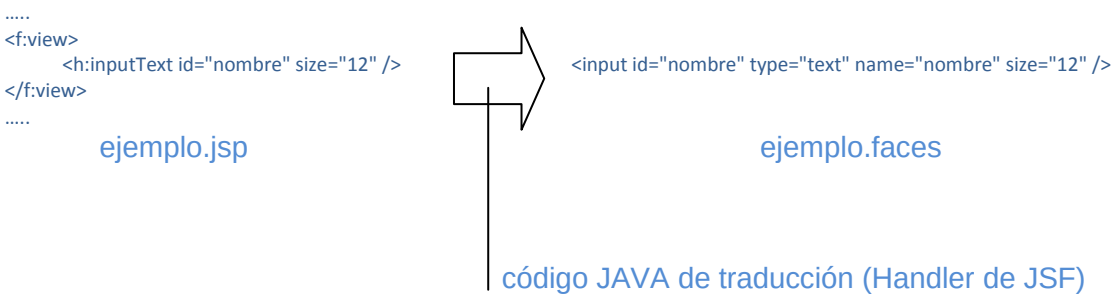
El siguiente artículo está dirigido a personas que tienen un buen conocimiento de JSF 1.2. No voy a explicar que es JSF, para que sirve, como se configura, cual es la mejor implementación, no voy a explicar Javadoc, ni nada por el estilo. Si quieres algo como lo que enumerado anteriormente mejor mira otro artículo y/o tutorial; en la red hay muchos muy buenos.

Mi intención es, partiendo de que el lector ya tiene una buena base, profundizar en aspectos que me parecen muy importantes y que si se conocen bien proporcionan al lector la capacidad de desarrollar una aplicación empresarial con este framework.

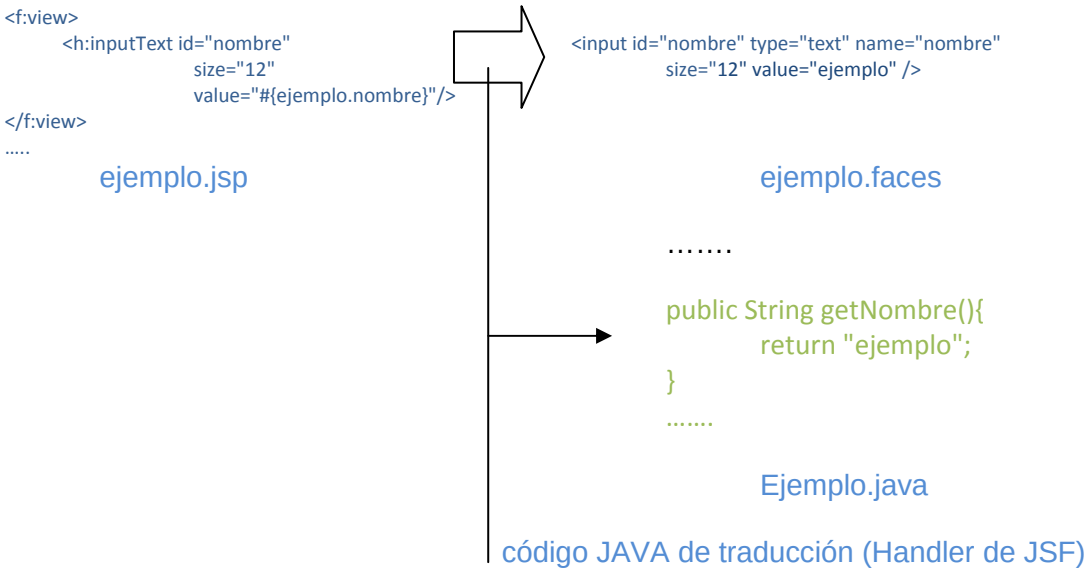
Para hacer más ameno el artículo introduzco pocos conceptos teóricos (intento ir al grano) e intento que el hilo de cada apartado se siga gracias a la explicación de ejemplos de código. Recomiendo que, al leer el artículo, cada vez que se mencione un ejemplo se tenga en código abierto para poder visionarlo antes de seguir leyendo el artículo.

2 Componentes

Una posible definición de los componentes de JSF (commandButton, dataTable) es que son anotaciones que al encontrarlas JSF en el código de una página se traducirán en una o varias etiquetas HTML. Este proceso de traducción es código JAVA que además de traducir un texto (una etiqueta JSF) en otro texto (una o varias etiquetas HTML)



puede llamar a otro código JAVA (métodos públicos de un Managed Bean) en el proceso



Por lo que si hay alguna etiqueta HTML que queremos conseguir que pueda utilizar las funcionalidades que ofrecen los Managed Bean y no se pueda obtener utilizando las etiquetas JSF nativas o tenemos un conjunto de etiquetas HTML que forman una funcionalidad muy común en un entorno web (ejemplo; un input con un List pueden ser un buscador) estamos ante unos ejemplos clarísimos de “Creación de un componente JSF”.

2.1 Mi primer componente JSF

Un crimen, en mi opinión claro, es que no exista un componente nativo de JSF que sea un **div.** Por lo que para ilustrar un ejemplo de cómo crear un componente para obtener una etiqueta HTML que vamos a crear un componente **e:divEjemplo**. El código de ejemplo lo puedes encontrar en \fuentes\etiquetaSimple.rar (mirar la lista de ficheros en el capítulo 9).

1º Crear una librería de etiquetas:

Voy a crear una librería de etiquetas (tld) donde definir la etiqueta divEjemplo y se pueda utilizar en el código fuente de una página.

```
….
<tag>
    <name>divEjemplo</name>
    <tag-class>ejemplo.etiqueta.DivEtiqueta</tag-class>
    <attribute>
       <name>estilo</name>
       <description>estilo</description>
      </attribute>
    <body-content>JSP</body-content>
 </tag>
….
```

- Name: Nombre de la etiqueta por la se identifica la etiqueta
- Tag-class: nombre de la clase java donde se implementa el componente etiqueta.
- Attribute: atributo aceptado por la etiqueta.

2º Crear una clase java que implemente la etiqueta.

Esta clase debe extender de una clase implementación “Tag” (como en el ejemplo javax.faces.webapp.UIComponentELTag). Se deben definir como mínimo los siguientes métodos:

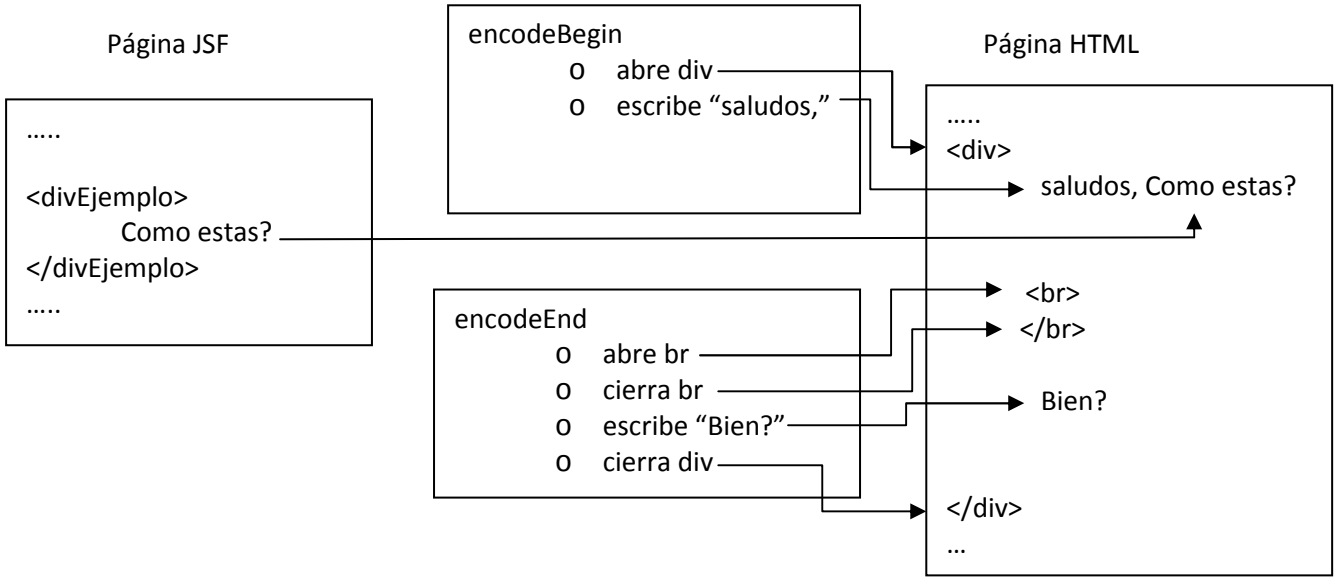
- getComponentType: cadena que identifica el tipo de componente faces que estamos utilizando. Si no se indica uno nativo habrá que definirlo en el faces-config.xml (de hecho es lo que vamos a hacer).
- getRendererType: si es distinto de nulo indica que la etiqueta se pinta de una manera determinada como por ejemplo en texto.
- setProperties: esto es una sobrecarga del método original por lo que lo primero es llamar al objeto super. La finalidad de sobrecargar este método es añadir en la lista de atributos de la etiqueta que estamos creando las propiedades que hemos definido.
- setXXXX y getXXXX: se deben definir los getters y setters de todas las propiedades que se han definido en la nueva etiqueta.

3º Definir el tipo de componente en el faces-config.xml y el tipo ha de coincidir con la cadena que devuelve el método getComponentType de la implementación de la nueva etiqueta.

4º Crear una clase java que implemente el componente de la etiqueta.

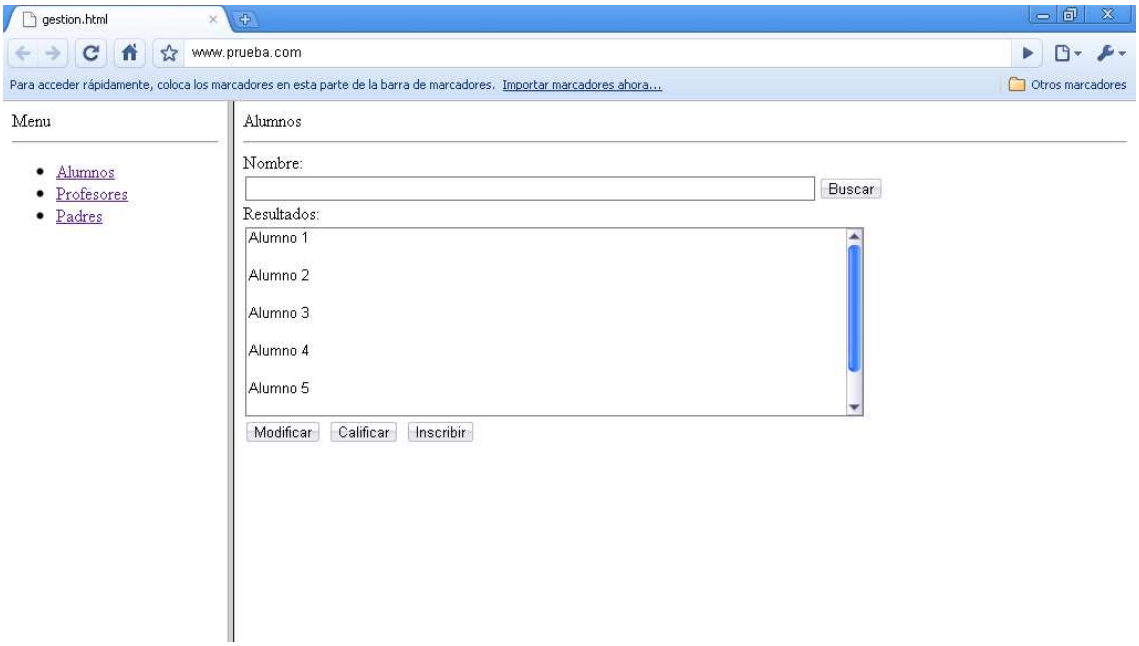
Esta clase debe extender de la clase UIOutput. Se deben definir como mínimo los siguientes métodos:

- encodeBegin: Defines que se va a renderizar antes de que se renderice el contenido de la nueva etiqueta.
- encodeEnd: Defines que se va a renderizar después de que se renderice el contenido de la nueva etiqueta.



2.2 Componentes complejos

Ahora voy a definir un componente más complejo. Voy a suponer que estoy desarrollando una aplicación web de gestión de un colegio donde en la parte izquierda tenemos un menú para acceder a los diversos individuos que están relacionados con el colegio (alumnos, profesores y padres). Cada vez que pinchamos en una opción de menú aparece en el frame central un buscador por nombre, el resultado de la búsqueda nos dará una lista de individuos que se pueden seleccionar (pinchando encima) y realizar sobre cualquiera de ellos una de las acciones que se ejecutan al pulsar los botones de la parte inferior.



Como la búsqueda es igual siempre y solo cambian las acciones y el origen de datos de las búsquedas voy a definir un etiqueta e:buscador que tiene como propiedades:

- datos: una función que me realiza la búsqueda y devuelve una lista de objetos “BuscadorResultado”.
- selección: id del objeto seleccionado en formato cadena.
- criterio: cadena insertada para que sea el filtro de búsqueda.

El código de ejemplo lo puedes encontrar en \fuentes\etiquetaCompleja.rar. (mirar la lista de ficheros en el capítulo 9).

La forma de crear el componente va a ser un componenteBase que incluye:

- Texto: Titulo de la búsqueda.
- Inpunt: Criterio de búsqueda.
- Texto: Titulo del resultado.
- Lista: Lista de resultados.
- Boton: Botón buscar.

Hasta el punto 4, el componente complejo es como el simple solo que en lugar de incluir solo un atributo se incluyen tres (datos, selección, y criterio).

Crear una clase java que implemente el componente de la etiqueta.

La clase debe extender de UIComponentBase porque lo voy a plantear como un componente que incluye varios componentes básicos.

Solo implementamos el método encodeBegin.

Preguntamos si hay un formulario, tiene que haberlo ya que nuestra componente tiene acciones relacionadas (buscar).

Creo el input del filtro de búsqueda como un objeto HtmlInputText y le asocio el atributo del ManagedBean “BuscadorBean” con la línea de código

```
cuadro.setValueExpression("value", dameExpresion(contexto, criterio));
```

con esto lo que consigo dos efectos:

- Cada vez que se haga un submit del formulario el valor que se indique en el cuadro de texto se le asocia a esta propiedad.
- El valor del texto en el cuadro de texto tiene el mismo ciclo de vida que tiene la propiedad.

Creo una lista cuyo valor se rellena invocando a un método del ManagedBean

```
MethodExpression funcion = dameMetodo(contexto, datos);
Object parms [] = new Object[1];
parms[0] = valor.getValue(contexto.getELContext());
List<BuscadorResultado> salida = (List) funcion.invoke(contexto.getELContext(), parms);
```

El valor del parámetro es el filtro del cuadro de texto anterior por lo que tomo la expresión de la referencia a la propiedad del ManagedBean

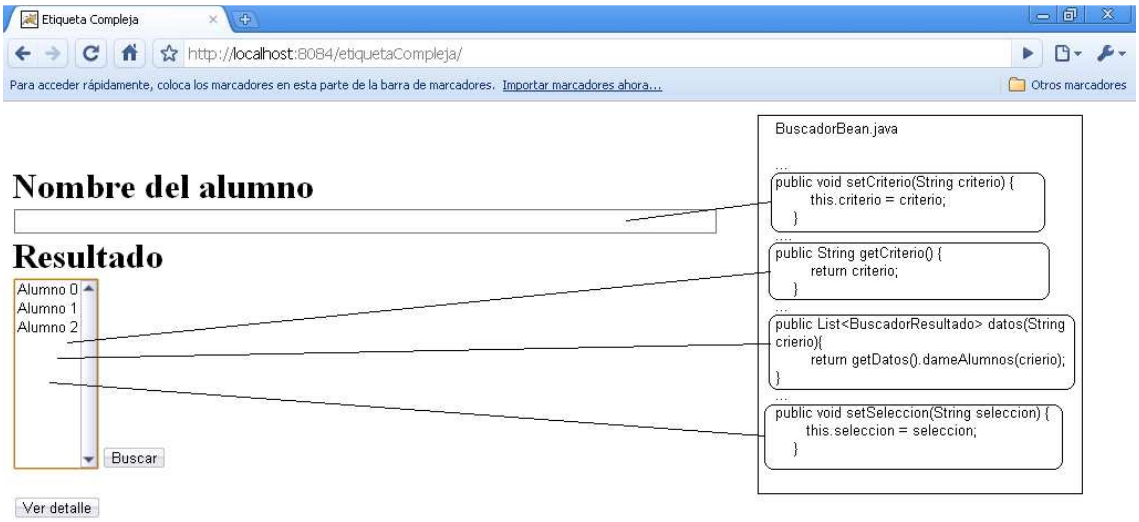
```
ValueExpression valor =  dameExpresion(contexto, criterio);
```

Por último el botón que como solo tiene que hacer sumbit y no ir a ninguna otra pagina pues es suficiente con incluirlo en el componente base.

Incluir los objetos en un componente no es complicado y es algo muy común en Java (como por ejemplo cuando estamos creando una aplicación SWING), lo realmente interesante es ver como con los métodos

```
private ValueExpression dameExpresion(FacesContext contexto, String expresion){

    Application app = contexto.getApplication();
    ELContext elContext = contexto.getELContext();
    ExpressionFactory exprFactory = app.getExpressionFactory();

    return exprFactory.createValueExpression(elContext, expresion, String.class);
 }

private MethodExpression  dameMetodo(FacesContext contexto, String expresion){

    Application app = contexto.getApplication();
    ELContext elContext = contexto.getELContext();
    ExpressionFactory exprFactory = app.getExpressionFactory();
    Class[] argumentTypes = new Class[1];
    argumentTypes[0] = String.class;

    return exprFactory.createMethodExpression(elContext, expresion, List.class,
argumentTypes);
   }
```

estoy consiguiendo asociar el valor de los objetos a propiedades y funciones de un ManagedBean



3 ViewHandler

La definición de componente del apartado anterior es algo simplista, todo hay que decirlo, pero por extensión nos da como conclusión un concepto también algo simplista pero muy interesante, que una implementación de JSF consiste en: “Ante una petición, evaluar el estado y valor de los objetos de la página JSF que hace la petición (Restore View), procesarla (validaciones, errores, llamar a las funciones de los ManagedBean, etc..) y generará una respuesta (Render Response)”. Voy a desarrollar este concepto:

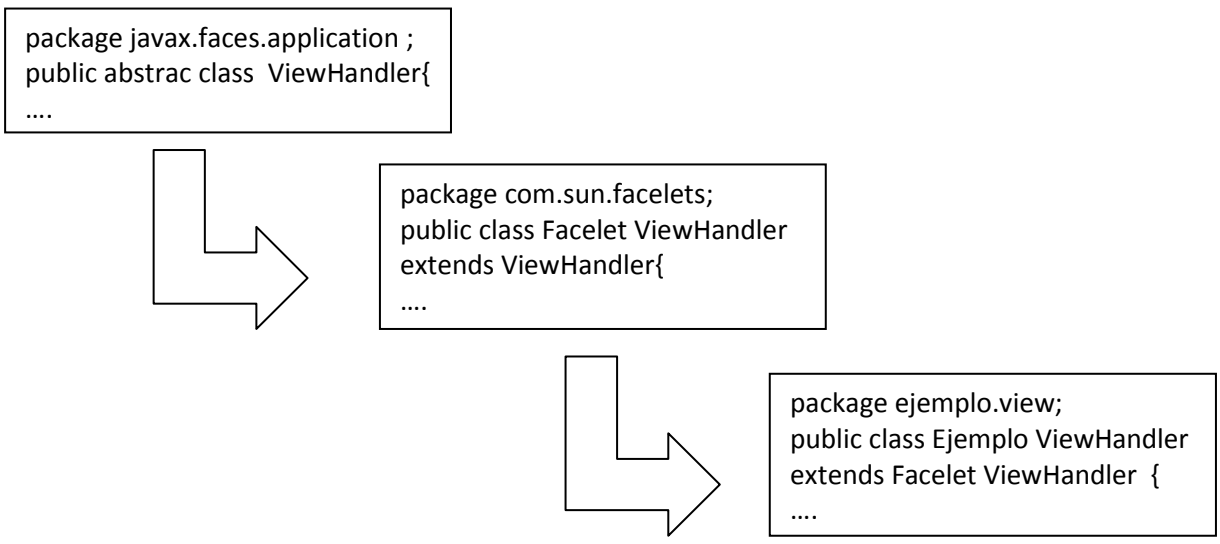
1º Premisa: La clase que se encarga de implementar las fases de “Restore View” y “Render Response” es el ViewHandler de la aplicación y da la *casualidad* que puedo saber cual es ya que se define en el faces-config.xml

```
<faces-config>
 <application>
  <!—Por ejemplo,el de la implementacion Facelets -->
  <view-handler>com.sun.facelets.FaceletViewHandler</view-handler>
 </application>
```

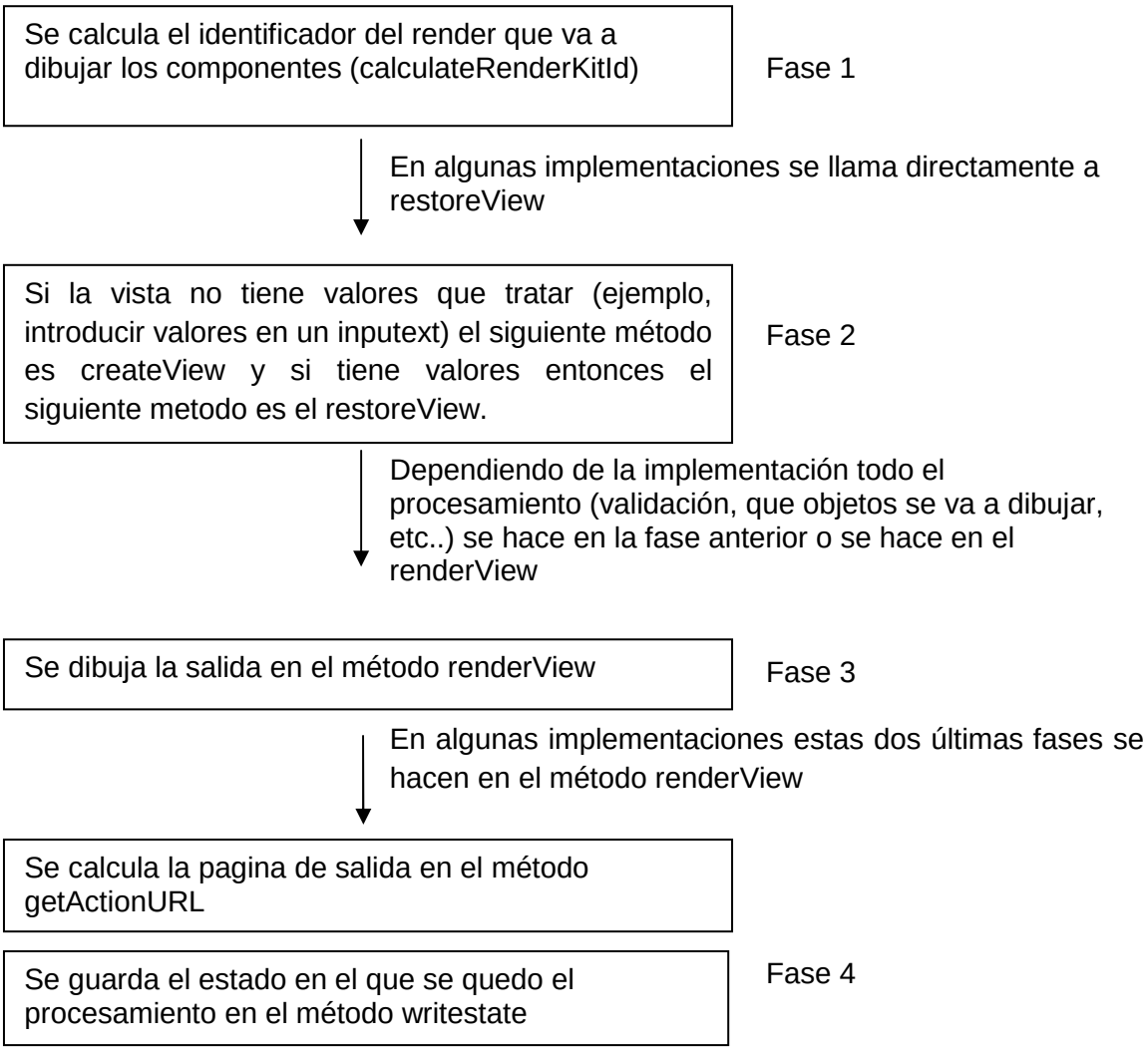
2º Premisa: Se cuál es la forma que tiene la clase, al menos los métodos más importantes, porque tiene que extender de la clase **javax.faces.application.ViewHandler.**

Conclusión: Por lo que si customizo esta clase entonces controlare lo que se va a procesar (los componentes que recogen y muestran información y dicha información) de la pantalla y lo que una vez procesado se va a renderizar en la pantalla. Es casi como tener nuestra propia implementación de JSF.

La mejor forma de customizar es crear una clase que herede del ViewHandler y empezar a sobrecargar los métodos. Por ejemplo, sigamos con Facelets



Todas las implementaciones extienden de javax.faces.application.ViewHandler pero la forma de implementar los distintos métodos y, sobre todo, la secuencia lógica que siguen los métodos depende de cada implementación. La secuencia de llamadas a los métodos de javax.faces.application.ViewHandler depende mucho de la implementación pero en general siempre es la misma cuando haces una petición:

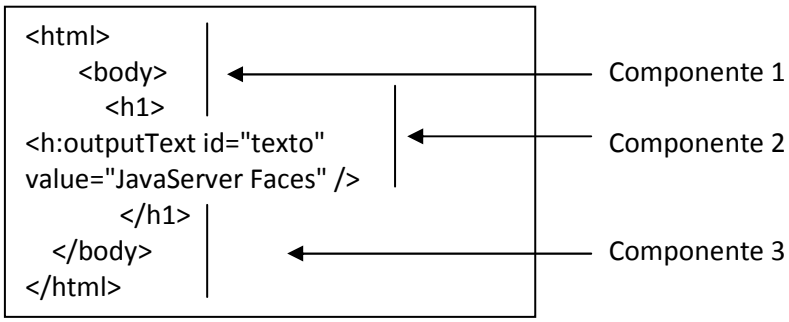


En mi opinión, a la hora de customizar el ViewHandler de una implementación los factores a tener en cuenta son:

- Como trata los ciclos de vida de JSF: El ViewHandler es el encargado de ir avanzando los ciclos de vida de JSF desde la petición hasta la respuesta y puede avanzar el ciclo cuando quiera. Por ejemplo, una implementación podría decidir, por cualquier motivo, ir a la fase de “Render View” al crear la vista y no realizaría las otras fases intermedias como la validación.
- Cuál es la persistencia de los componentes y cuál es la forma de tratar los componentes: El ViewHandler tiene que leer los componentes de una página, analizarlos y dar una respuesta. La forma en que lee esos componentes y los tiene presentes en cada una de las funciones que tiene que implementar porque le obliga la interfaz javax.faces.application.ViewHandler (que al final es lo que nos interesa

porque es por donde le vamos a atracar al ViewHandler para que se comporte como queremos) dependen de la implementación.

Basándome en el desarrollo de este concepto, voy a utilizar la implementación de Facelets (com.sun.facelets.FaceletViewHandler) frente a otras como la propia de Sun Faces (com.sun.faces.application.ViewHandlerImpl) ya que, en mi opinión, la forma de avanzar por las fases es muy rígida en la implementación de Sun Faces y su forma de leer y tratar los componentes es muy poco versátil. La única pega que le he visto a Facelets (respecto a persistir y tratar componentes) es el hecho de que para Facelets los objetos HTML que no etiquetas de JSF no son componentes. Si examinas como deduce los objetos veras que



En otras implementaciones como IceFaces todos los objetos HTML son componentes ya sean etiquetas JSF, etiquetas HTML o etiquetas IceFaces. Los componentes que se forman a partir de etiquetas HTML son componentes sin mucha funcionalidad pero tienen los metodos basicos como “setRendered” o “getAttributes” que son muy útiles.

Y ahora que ya se como customizar un ViewHandler, de que van los métodos del ViewHandler y que implementación vamos a utilizar, voy a ver para que nos puede servir customizar el ViewHandler mientras voy viendo los métodos de la clase ViewHandler.

3.1 Seguridad de la aplicación Web

Voy a utilizar el ViewHandler para implementar la seguridad en una aplicación web a nivel de vista. A cada usuario, tras logarse, se le aplica la seguridad a tres niveles:

- Pantallas que puede ver o no ver un usuario.
- Dentro de una pantalla, que objetos puede ver un usuario.

El código de este ejemplo lo tienes en fuente\seguridad.rar.

La seguridad la voy a configurar de forma fija en el código que, de forma algo rudimentaria pero muy simple, me va a rellenar el objeto que voy a guardar en sesión para representar al usuario que se conecta y contendrá la siguiente información útil:

- Usuario: Contiene la lista de usuarios y las contraseñas.
- Pantallas: Contiene la lista de pantallas que puede ver cada usuario.

- Compontes: Contiene la lista de componentes que puede ver cada usuario.

Voy a controlar solo el acceso a las paginas que están en los directorios “admin” y “publico”. En el ejemplo he pasado por alto muchos criterios de estilo de cómo se debería programar una aplicación web para centrarme más en lo que importa pero cabe destacar que a la hora de cómo estructurar las paginas en una aplicación web Java se debe montar en una estructura de directorios ya que si se desea introducir seguridad nativa de Java (JAAS) los patrones “url-pattern” tienen la peculiaridad de que si quieres que la seguridad afecte a las paginas que estén en el raíz de la aplicación web o les afecta a todas o a ninguna mientas que si están metidas en directorios puedes hacer lo que quieras.

El acceso a las paginas lo voy a controlar en el punto antes de construir la pagina en la Fase 2, por eso en el ejemplo se incluye la llamada al método validarDestino en los metodos “createview” y “restoreview”. En este método se ignora la extensión de la pagina destino ya que puede ser confusa

```
…
String vista=viewId.substring(1, viewId.lastIndexOf("."));
…
```

Y no compruebo los privilegios de acceso en las páginas que no hacen falta

```
….
if (vista.indexOf("welcomeJSF")==-1
        && vista.indexOf("error/")==-1
….
```

Si no se puede acceder entonces se envía una señal de prohibido

```
..
response.sendError(HttpServletResponse.SC_FORBIDDEN);
..
```

Y se renderiza la pagina de error correspondiente indicada en el web.xml

```
….
<error-page>
                <error-code>403</error-code>
                <location>/error/sinPermiso.faces</location>
</error-page>
…
```

El control de los objetos se hace a la hora de renderizar la página. El método renderView llama internamente al método buildView antes de pintar por lo que sobrecargo este método para que si un objeto se llama “controlXXXX” y no tengo permisos para verlo entonces no se pinte el objeto.

```
protected void buildView(FacesContext arg0, UIViewRoot arg1) throws IOException,
FacesException {
      super.buildView(arg0, arg1);
   .....
revisaTodos(c,usuario);
...
```

Ojo, se hace después de que se ejecute el método buildView, si se hiciera antes entonces no habría ningún objeto que revisar. Si entro en la página admin/alta.faces no podre ver el botón "controlModifica" porque empieza por "control" y no tengo permiso para verlo.

3.2 Ampliar funcionalidad de etiquetas JSF

Las etiquetas tienen una serie de atributos como "value" para indicar el valor, "required" para indicar que el campo es obligatorio, etc.. Voy a mostrar cómo utilizar el viewHandler para añadir un nuevo atributo que de una funcionalidad genérica.

Voy seguir utilizando el ejemplo de seguridad. Supongamos que nos interesa que un usuario siempre pueda entrar a la aplicación, por lo que si no se sabe ningún usuario y contraseña se le dé una explicación de lo que puede hacer. Voy a incluir el atributo "explica".

Se incluye el atributo en los componentes que se quiere explicar

```
...
<h:outputText id="texto1" value="Usuario:" /><br/>
      <h:inputText id="usu" value="#{beanSeguridad.user}" explica="Usuarios de
prueba:usuario1"/><br/>
      <h:outputText id="texto2" value="Contraseña:" /><br/>
      <h:inputText id="pass" value="#{beanSeguridad.pass}" explica="Constraseñas de
prueba:pu1"/><br/>
      <h:messages                    style="color:                    red"/>
...
```

y el viewHandler se incluye una función que recorra los atributos de los componentes y en caso de que exista algún atributo "explica" se actúa en consecuencia, es decir incluyendo un mensaje.

3.3 Incluir un componente

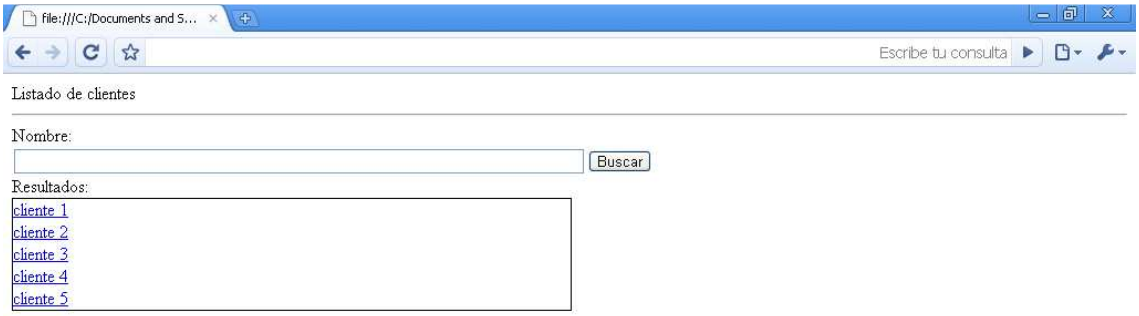
Otra caso muy interesante es el de incluir componentes. Supongamos que nos interesa meter publicidad y en cualquier pagina de nuestra aplicación, una vez que entras, se muestre aleatoriamente o un enlace a una página o un texto. La estrategia que voy a seguir es incluir un "outputText" o un "form" con un "commandlink" a otra página que se incluirá al crear la vista (createView) y antes de pintar nada (renderView).

```
…
   UIViewRoot salida = super.createView(arg0,arg1);   ←——  Primero se crea la vista

      incluyeObjetos(salida);

      return salida;   ←————————  Pero se incluye antes de terminar la fase
…..
```

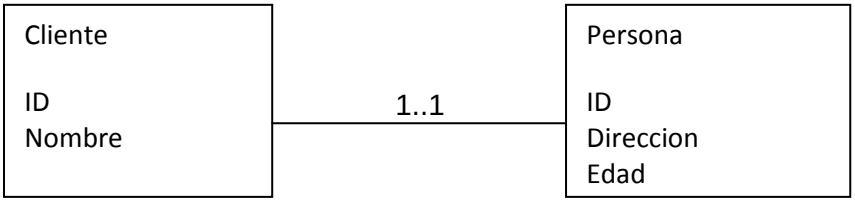
Todos estos casos de customizar un ViewHandler no tienen como objetivo que cada vez que desarrolles una aplicación JSF intentes utilizar alguno de ellos. Lo que pretendo es mostrar las entrañas de un ViewHandler y las unidades de información que utiliza para que cuando tengas que solucionar un problema en tu aplicación JSF que no pueda resolverse de forma basica, cosa muy normal que pase en un desarrollo, pudas pensar ¿lo puedo solucionar customizando el ViewHandler?.

4 JPA

Un error muy grave cuando se está desarrollando una aplicación web es consultar datos (normalmente accesos a bases de datos) que no se muestran al usuario. Pongamos como ejemplo una aplicación de gestión de clientes que permite introducir un nombre y devuelve todos los clientes con ese nombre y que al pinchar sobre el nombre nos permita ver el detalle del cliente.



Con el siguiente esquema de base de datos:

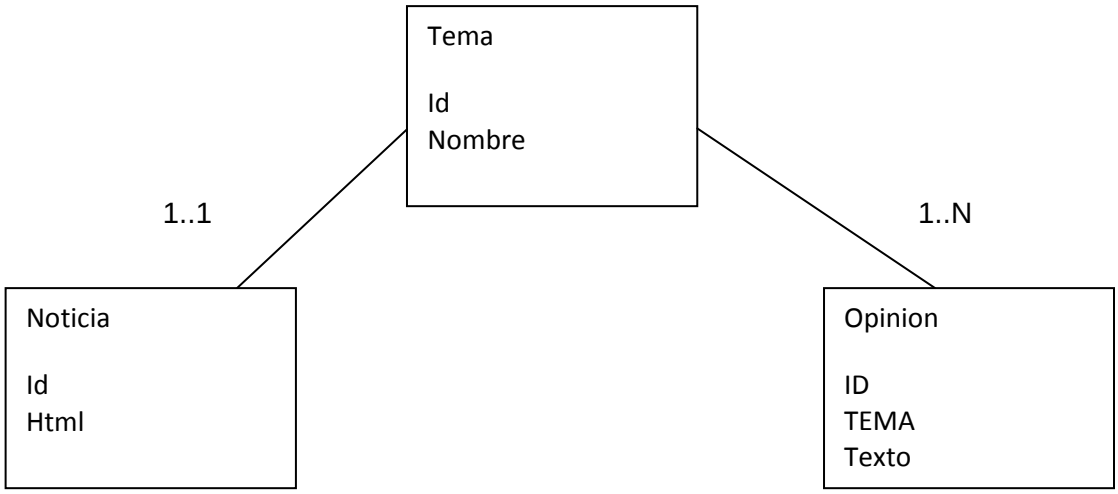


Al buscar los clientes solo es necesario el nombre de los clientes pero si traen todos los detalles (que pueden ser datos como dirección, edad, dni, población, nacionalidad, etc) y los guardamos en sesión no será necesario volver a consultar por lo que será **más rápido** (tiempo de respuesta de la aplicación desde que se pincha en el nombre del cliente hasta que se ve el detalle) y el **código que utilizaremos será mucho más simple**. Lo malo es que se está usando demasiada memoria y cuando accedan varios usuarios influirá muy negativamente en el rendimiento del servidor.

JPA utilizado en aplicaciones web permite simplificar el código a utilizar no solo en la envoltura de la base de datos, sino también en el código que se utiliza en el Managed Bean para manejar las paginas ya que el código JPA da a entender que se ha consultado toda la información pero hasta que no se acceda explícitamente al detalle no se accedería a base de datos (Modo Lazy). Esto no es tan rápido como tener los objetos en sesión pero elimina el problema del gasto excesivo de memoria y la velocidad es bastante buena.

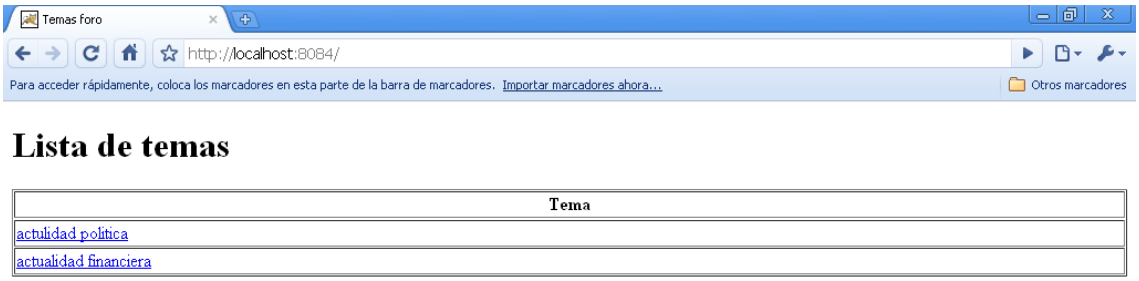
Voy a utilizar el ejemplo de un foro para ilustrar la forma de utilizar JPA. La aplicación foro muestra en la pantalla inicial una lista de temas. Cada elemento de la lista de temas es un link que permite ver el detalle del tema que consiste en un campo que almacena todo el HTML que se verá por pantalla y una lista de opiniones de los usuarios del foro para cada tema. Todo el código del ejemplo lo tienes en fuente\foro.rar(mirar la lista de ficheros en el capítulo 9).

El esquema de base de datos es el siguiente:



El código de ejemplo lo puedes encontrar en \fuentes\foro.rar(mirar la lista de ficheros en el capítulo 9). Para que funcione hay que definir una librería que se llame “Open Jpa” con todos los ficheros jar de OpenJpa y poner la ruta local donde están esos ficheros en el fichero build-impl, en el target “-post-compile” en el valor que actualmente vale c:\Java\librerias. También se debe crear una base de datos en MySQL cuyo script esta en el fichero foro.sql.

La pantalla inicial muestra la siguiente información:



Tenemos dos registros temas y si vemos el log de JPA aparece la consulta se puede comprobar que solo se accede a los datos necesarios:

```
3187  foroPU  TRACE  [http-8084-3] openjpa.jdbc.SQL - <t 25277396, conn 31520656>
executing prepstmnt 14440243
SELECT t0.Id, t0.Nombre   FROM tema t0 LIMIT ?, ? [params=(long) 0, (long) 10]
```

Como se puede ver no se ha accedido a los datos de las tablas noticia y opinión y si vemos el objeto depurando vemos que los campos de la entidad Tema noticia y opinión no tienen valor sino un socket de JPA.

Si pinchamos en uno de los temas, en el bean “detalleBean” accedemos a los campos html de Noticia

```
public String getHtml() {
    return temaActual.getNoticia().getHtml();
}
```

y a al campo Texto de Opinión

```
public List<OpinionDto> getOpiniones(){

    List<Opinion>  datosOpinion = temaActual.getOpinion();
  ….
```

En este momento JPA rellena esos campos y en el log de JPA aparece ahora las consultas correspondientes:

```
foroPU  TRACE  [http-8084-5] openjpa.jdbc.SQL - <t 1622177, conn 18550448> executing
prepstmnt 21570811
SELECT t0.Html, t0.id  FROM noticia t0     WHERE t0.id = ? [params=(int) 0]
 foroPU  TRACE  [http-8084-5] openjpa.jdbc.SQL - <t 1622177, conn 18550448> [16 ms] spent
 foroPU  TRACE  [http-8084-5] openjpa.jdbc.SQL - <t 1622177, conn 7894673> executing
prepstmnt 19825275
SELECT t0.Id, t0.Texto   FROM opinion t0  WHERE t0.TEMA = ? [params=(int) 0]
```

En la pantalla vemos el resultado

Ahora los campos Noticia y Opinión de la entidad Tema ya están rellenos de tal manera que los hemos ido a buscar a base de datos solo cuando los hemos necesitado y sin escribir ni una sola línea código expresamente para este fin.

Cabe destacar que hemos utilizado Enhaner de JPA debido a que si no se utiliza la relación “OneToOne” no se puede cargar en modo LAZY. Para utilizar Enhancer de JPA hemos modificado el fichero build-impl de tal manera que una vez compilado ejecute el siguiente objetivo:

```
<target name="-post-compile">
<!--Mensaje-->
      <echo message="inicio enhancer"/>
<!--Classpath. Deben estar las librerias de JPA y los .class de las entidades-->
      <path id="jpa.enhancement.classpath">
        <pathelement location="build/web/web-inf/classes"/>
        <fileset dir="c:\java\librerias">
          <include name="**/*.jar"/>
        </fileset>
      </path>
      <!--Esto es por si no se ha incluido el persistence.xml-->
      <copy includeemptydirs="false" todir="build/web/web-inf/classes">
        <fileset dir="src/conf" excludes="**/*.launch, **/*.java"/>
      </copy>
      <!--Se define el comando que va a realizar el Enhancer-->
      <taskdef name="openjpac" classname="org.apache.openjpa.ant.PCEnhancerTask">
        <classpath refid="jpa.enhancement.classpath"/>
      </taskdef>
        <!--Ejecutar el Enhancer-->
      <openjpac>
        <classpath refid="jpa.enhancement.classpath"/>
      </openjpac>
       <echo message="Fin enhancer"/>
  </target>
```

5 Diseño de una aplicación WEB

Una vez ya visto como plantear los componentes, la potencia que puede dar el ViewHandler y cómo afecta al planteamiento de la aplicación el uso de framework como puede ser JPA ya se tiene una buena base para diseñar e implementar una aplicación con JSF. En este apartado vamos a enumerar otras reglas que en general se pueden aplicar en cualquier aplicación web a la hora de diseñarla.

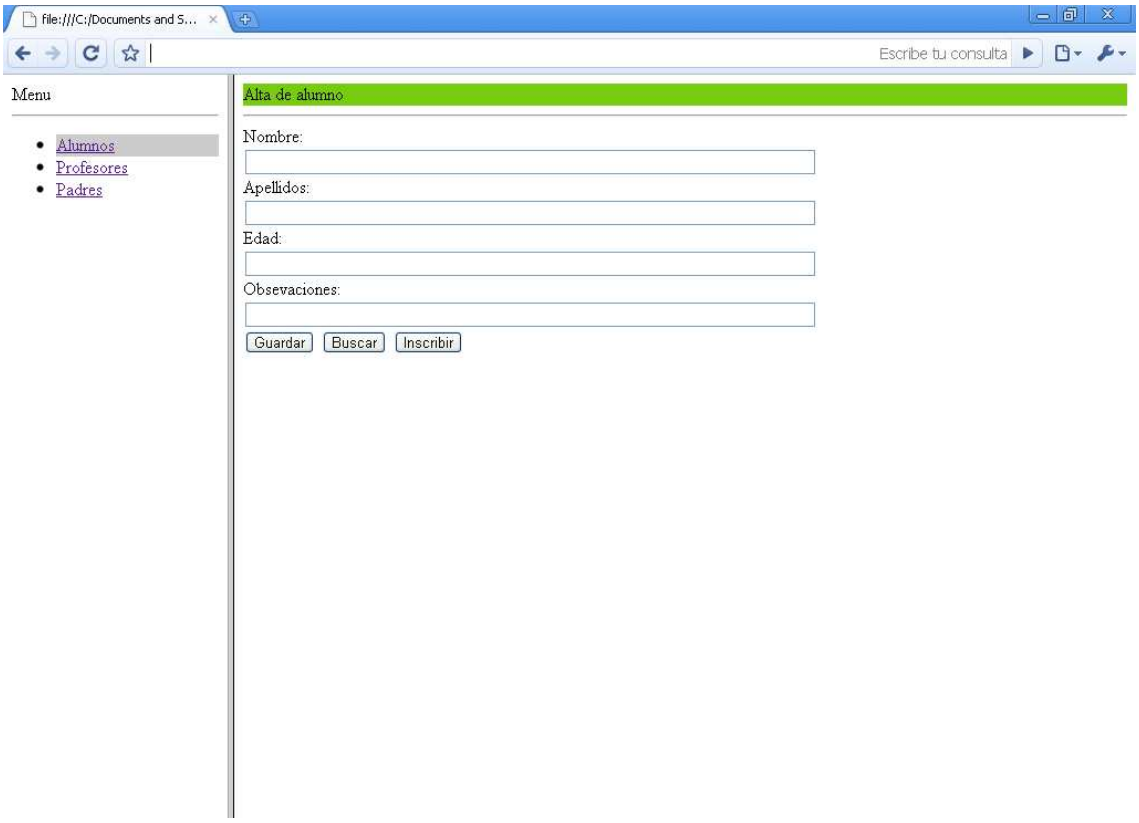
5.1 Diseño de páginas

Hay una regla del buen diseño de aplicaciones web que me gustaría especificar deforma coloquial: "**Cuando diseñes una aplicación web NO diseñes cada página con su propia lógica, establece una base común para toda la aplicación y adapta la lógica de la pantalla a esa base común**".

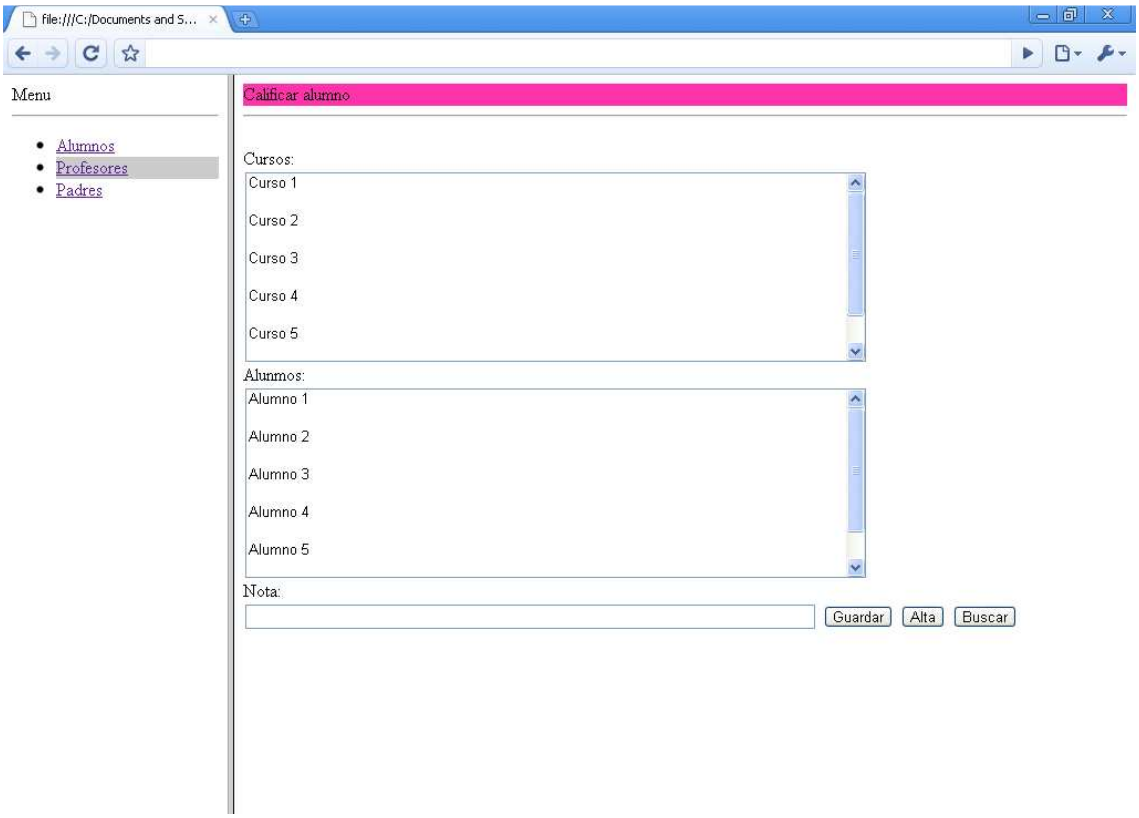
Me explico, pongamos como ejemplo la aplicación de "Gestión Colegio". El cliente nos pide la aplicación de Gestión Colegio y al tomar los requisitos se ve que en el apartado de alumnos el alta es lo más importante, en el apartado de profesores el calificar es lo más importante y en el apartado de padres lo más importante es ver hijos de los padres son alumnos.

Empieza el diseño de las páginas y al basarse en lo que vemos más importante:

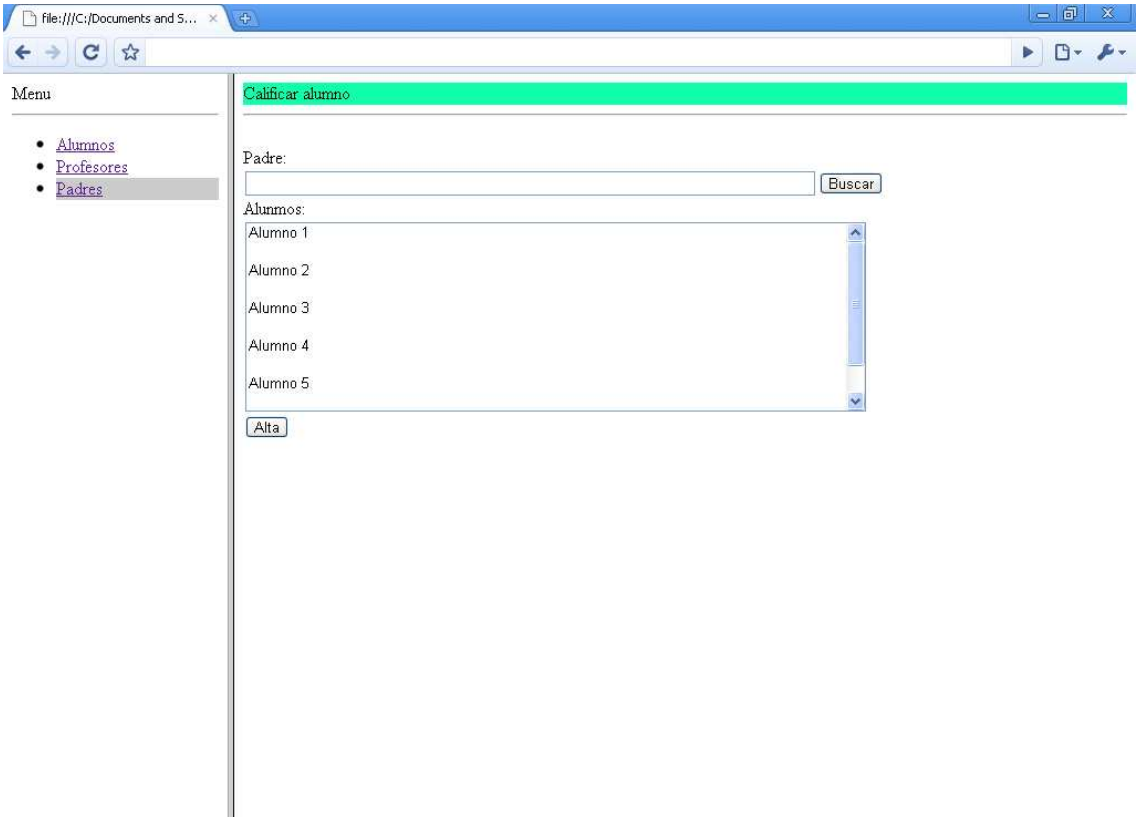
Alumnos



Profesores



Padres

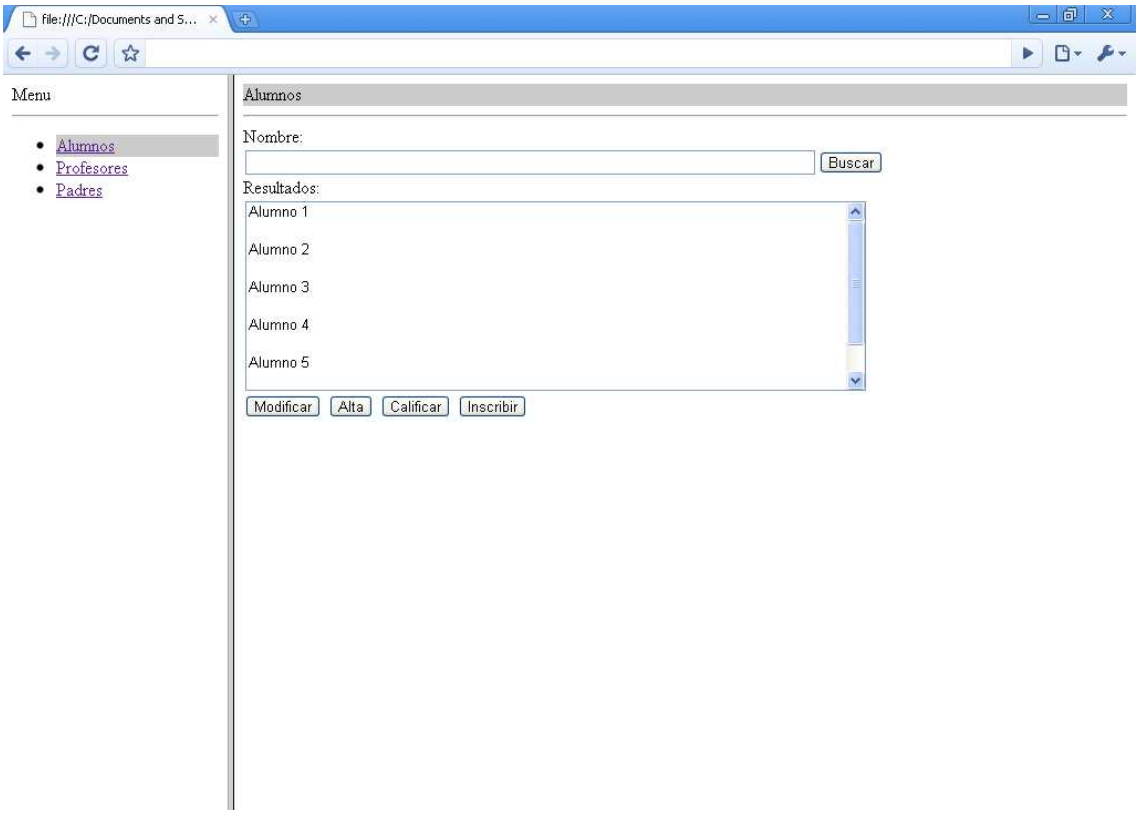


Funcionalmente puede servir pero:

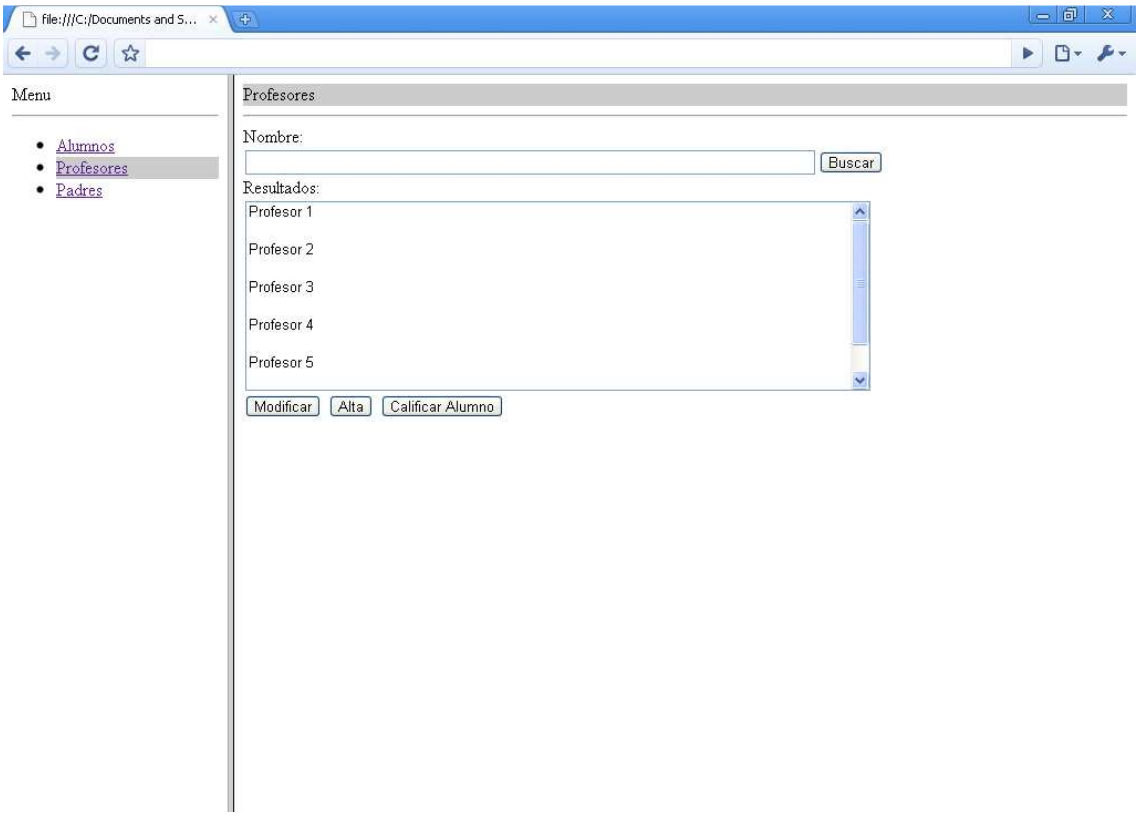
- Cada opción principal del menú me lleva a una página completamente nueva con una funcionalidad completamente nueva, lo cual resulta muy estresante para un usuario.
- Al preocuparme solo de la funcionalidad es fácil descuidar la sincronía de objetos gráficos como botones, colores, etc..
- Si quisiéramos mostrar un “camino de hormiga” para ver en que profundidad de la pagina en la que estamos nos daríamos cuenta que para buscar alumnos seria Alta > Buscar y para buscar profesores seria Calificar > Buscar.

Pero si establecemos una simple base común en la cual decidimos que cada opción de menú empieza con una búsqueda del elemento de la opción de menú (alumno, profesor o padre) seguido de una botonera con las acciones que puede realizar las pantallas serian así:

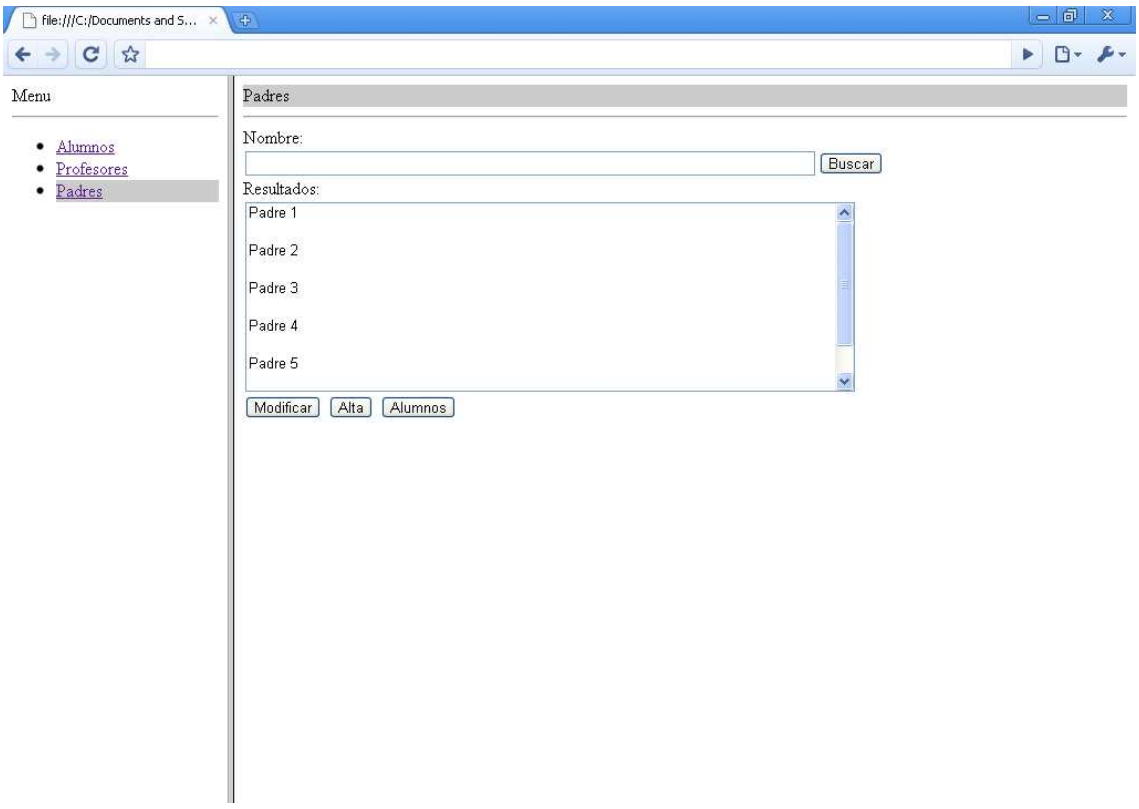
Alumnos



Profesores



Padres



Con una simple decisión inicial vemos que:

- Todas las opciones principales del menú me llevan a paginas en las que solo cambia el titulo y la botonera.
- Al preocuparme más por seguir una base en las paginas difícil descuidar la sincronía de objetos gráficos como botones, colores, etc..
- Si quisiéramos mostrar un “camino de hormiga” para ver en que profundidad de la pagina en la que estamos siempre empezaría igual.

5.2 Diseño de clases

La otra regla es muy parecida a la anterior pero de diseño de bajo nivel, es decir de desarrollo. Como va ser la estructura de clases. **Al diseñar la estructura de clases de una aplicación NO utilices una clase de cualquier manera cuando lo necesites, decide que componentes (conjuntos de clases) debes tener en la aplicación y utiliza tanas clases de cada componente como necesites hasta que obtengas lo que necesitas**.

1º Decidir que patrón vas a utilizar.

“Una posible definición de patrón, que vaya casualidad nos viene muy bien, es un conjunto de **componentes** que iteractúan entre si para resolver un problema de una **manera determinada**.” Puff, que ha querido decir?.

Definamos los conceptos más importantes:

- Componentes: Es un conjunto de clases que según la lógica que queramos dar al conjunto (al componente) son clases que sirven para lo mismo. Por ejemplo, si definimos un componente como “De acceso a bd” entonces todas las clases que accedan a base de datos como “ClienteBD.java”, “CompraBD.java”,etc.. tienen que estar en este mismo componente.
- De una manera determinada. Es la forma de relacionar los componentes. Si ya has definido los componentes (y suponemos que de forma correcta) la forma de interactuar entre dichos componentes es lo que define al patrón. Por ejemplo, si los componentes son Modelo, Vista y Controlador entonces si las relacionamos de tal manera que la vista es la parte que interactúa con el usuario, el controlador es la parte que reacciona a eventos que comunica el componente Vista y devuelve información al componente Vista según dicta el componente Modelo entonces el patrón es el MVC.

JSF está basado en MVC. En general para cualquier tecnología que consista en una página web el patrón MVC es la mejor opción.

2º Aplicar y extender el patrón.

Pongamos como ejemplo la aplicación “foro” del apartado JPA. Al aplicar el patrón obtengo

MODELO	CONTROLADOR	VISTA
entidad.Noticia.java entidad.Tema.java entidad.Opcion.java dto.OpcionDto.java dto.TemaDto.java	bean.DetalleBean.java bean.TemaBean.java	Inicio.jsp Detalle.jsp

Y al extender el patrón para que se ajuste mejor a la problemática de la aplicación obtengo

DATOS
datos.BDFuente.java datos.TemaGestion.java

Componente que contiene todas las clases con lógica de acceso a datos.

Ya está la solución montada. No hay que olvidar que un gran fallo, y que se da en muchos desarrollos web, es saltarse el patrón aplicado y poner clases en componentes donde no deben estar o funcionalidad en clases donde no deben estar. Este fallo se produce por diversos motivos:

- No hay tiempo para desarrollar de forma correcta y solo se quiere algo que funcione.
- Mala documentación y mal traspaso de conocimientos.
- Desconocimiento de la tecnología.

En JSF, el fallo más común que he visto es sobrecargar el ManagedBean con acceso a base de datos, con operaciones muy complejas, etc.. No debemos olvidar que el ManagedBean es un controlador y que sirve para el manejo de eventos. Si en un ManagedBean necesito un dato de BD hago una llamada a un método de una clase que se encarga del acceso a datos, si necesito que se resuelva una operación compleja llamo a un método de una clase que resuelve esa operación compleja pero NO PICO EL CODIGO EN EL MANAGEDBEAN. Veámoslo en el ejemplo de la aplicación “foro”, cuando se inicia la aplicación (evento) y el ManagedBean TemaBean (controlador) tiene que obtener los datos NO accede a base de datos sino que llama al método “recuperarTodosTema” y transforma esa información de entidad a dto (Modelo) para poder pasársela a la página inicio.jsp (Vista).

6 Conclusiones

- Al desarrollar una página con JSF, un desarrollo estandar consiste en utilizar los componentes nativos de JSF siempre que sea posible y las etiquetas de HMTL cuando no se puedan utilizar los componentes de JSF. Esta es una buena práctica pero si somos capaces de diseñar las pantallas de la aplicación de manera que los objetos de la pantalla sigan algún patrón que nos permita más fácilmente identificar componentes que podemos crear y utilizar en muchas de las páginas de nuestra aplicación entonces podremos comprobar cómo la potencia de JSF no solo nos quita trabajo sino que además nos ayuda a mejorar el diseño de nuestra aplicación.
- JSF es un framework muy versátil y si utilizas otro framework, como JPA o Spring, no es un error guiar el diseño de la aplicación para obtener mejor rendimiento de ese otro framework. La experiencia me ha demostrado que si no haces algo que sea muy descabellado JSF se adecua muy bien al uso de otros frameworks.
- Cada implementación de JSF hace las cosas a su manera, pero si no tiene viene bien modifica la forma de procesar los objetos (viewHandler) y hazte tus nuevos componentes si fuera necesario. Pero mucho cuidado, no confundas el adecuar JSF a tus necesidades con el mal uso del framework. La forma correcta de actuar es que antes de modificar JSF a tu conveniencia tienes que asegurarte de que la implementación JSF no puede hacerlo que necesitas (Lee la documentación).

7 Documentación recomendada

Te recomiendo los siguientes enlaces:

- Tutorial J2EE1.4 de Sun http://download.oracle.com/docs/cd/E17477_01/javaee/1.4/tutorial/doc/J2EETutorial.pdf
- Comunidad JSF Central http://jsfcentral.com/
- Pagina general de proyectos, eventos, artículos sobre el mundo java http://www.java.net/

8 Anotaciones técnicas

Todo el artículo está basado en JSF 1.2.

Los códigos de ejemplo los he desarrollado con NETBEANS 6.0.1 con un servidor TOMCAT 6.0.14.

La versión de Open JPA es la 1.2.2.

La versión de Facelets es la 1.1.15.

La versión de la implementación de Java Sun JSF es la 1.2.

9 Código de ejemplo

Estructura de ficheros de cada código de ejemplo.

etiquetaCompleja.rar

```
etiquetaCompleja
        build.xml
        nbproject
                ant-deploy.xml
                build-impl.xml
                faces-config.NavData
                genfiles.properties
                project.xml
                private
                        private.properties
                        private.xml
        src
                conf
                        MANIFEST.MF
                java
                        ejemplo
                                bean
                                        BuscadorBean.java
                                datos
                                        Datos.java
                                dto
                                        BuscadorResultado.java
                                etiqueta
                                        BuscadorEtiqueta.java
                                excepcion
                                        Control.java
                                ui
                                        BuscadorComponente.java
        web
                forwardToJSF.jsp
                resultado.jsp
                welcomeJSF.jsp
                ejemplo.tld
                WEB-INF
                        faces-config.xml
                        web.xml
                META-INF
                        context.xml
```

etiquetaSimple.rar

```
etiquetaSimple
        build.xml
        nbproject
                ant-deploy.xml
                build-impl.xml
                faces-config.NavData
                genfiles.properties
                project.properties
                project.xml
                private
                        private.properties
        src
                conf
                        MANIFEST.MF
                java
                        ejemplo
                                etiqueta
                                        DivEtiqueta.java
                                ui
                                        DivComponente.java
        web
                forwardToJSF.jsp
                welcomeJSF.jsp
                ejemplo.tld
                WEB-INF
                        faces-config.xml
                        web.xml
                META-INF
                        context.xml
```

foro.rar

```
foro
    build.xml
    foro.sql
    nbproject
        ant-deploy.xml
        faces-config.NavData
        genfiles.properties
        project.properties
        project.xml
        private
            private.properties
            private.xml
    src
        conf
            MANIFEST.MF
            persistence.xml
        java
            ejemplo
                bean
                    DetalleBean.java
                    TemaBean.java
                datos
                    BDFuente.java
                    TemaGestion.java
                dto
                    OpinionDto.java
                    TemaDto.java
                entidad
                    Noticia.java
                    Opinion.java
                    Tema.java
    web
        detalle.jsp
        forwardToJSF.jsp
        inicio.jsp
        WEB-INF
            faces-config.xml
            web.xml
        META-INF
            context.xml
```

seguridad.rar

- seguridad
  - build.xml
  - nbproject
    - ant-deploy.xml
    - build-impl.xml
    - faces-config.NavData
    - genfiles.properties
    - project.properties
    - project.xml
    - private
      - private.properties
      - private.xml
  - src
    - conf
      - MANIFEST.MF
    - java
      - ejemplo
        - bean
          - AdminBean.java
          - SeguridadBean.java
        - negocio
          - UsuarioNegocio.java
        - pojo
          - Usuario.java
        - view
          - EjemloViewHandler.java
  - web
    - forwardToJSF.jsp
    - menu.xhtml
    - welcomeJSF.xhtml
    - admin
      - alta.xhtml
      - baja.xhtml
      - detalle.xhtml
    - error
      - sinPermiso.xhtml
    - META-INF
      - context.xml
    - publico
      - alta.xhtml
    - WEB-INF
      - faces-config.xml
      - web.xml