Componentes de presentación

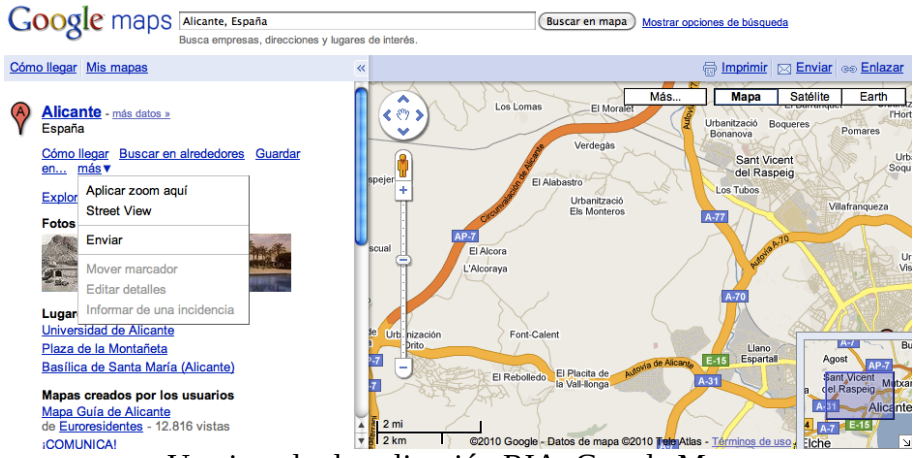
Índice

1. Introducción a JavaServer Faces

1.1. Frameworks RIA basados en el servidor

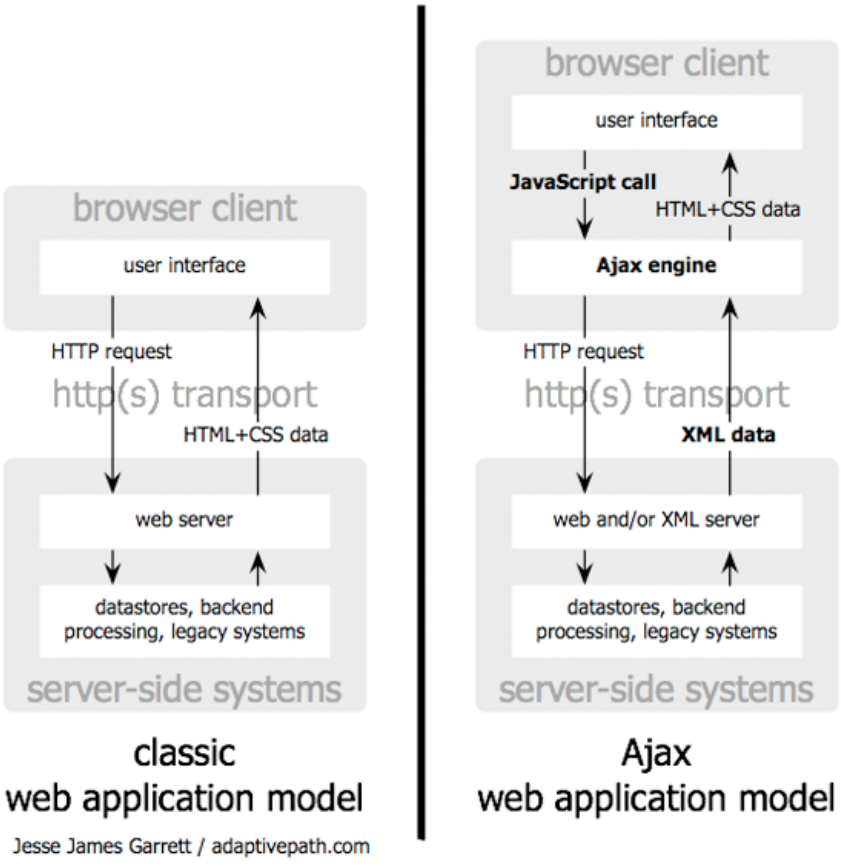
Frente a las aplicaciones web tradicionales basadas en la navegación entre distintas páginas, en los últimos años se han popularizado las aplicaciones web de una única página que intentan simular las aplicaciones de escritorio. Son las denominadas RIA (*Rich Internet Applications*).

Google fue uno de los primeros promotores de este tipo de aplicaciones, con ejemplos como Google Maps. Esta aplicación web utiliza JavaScript de forma intensiva para interactuar con el mapa o para seleccionar opciones (*imprimir*, *enviar*, etc.). Todas estas acciones se realizan en una única página del navegador.



Un ejemplo de aplicación RIA: Google Maps

En las aplicaciones tradicionales cada petición al servidor hace que éste genere una nueva página HTML. En la parte izquierda de la siguiente figura se muestra este modelo.



En la parte derecha de la figura vemos el funcionamiento de las aplicaciones RIA. Se siguen realizando peticiones HTTP al servidor, pero no es el navegador quién lo hace directamente, sino funciones JavaScript residentes en la página. Estas funciones envían la petición y los datos al servidor y éste devuelve los datos de respuesta en algún tipo de formato (texto, XML o JSON). Las funciones JavaScript en la página del navegador formatean estos datos y los muestran en pantalla actualizando el árbol de componentes DOM. El servidor no genera una nueva página sino sólo los datos. Este enfoque se ha denominado Ajax (Asynchronous JavaScript and XML).

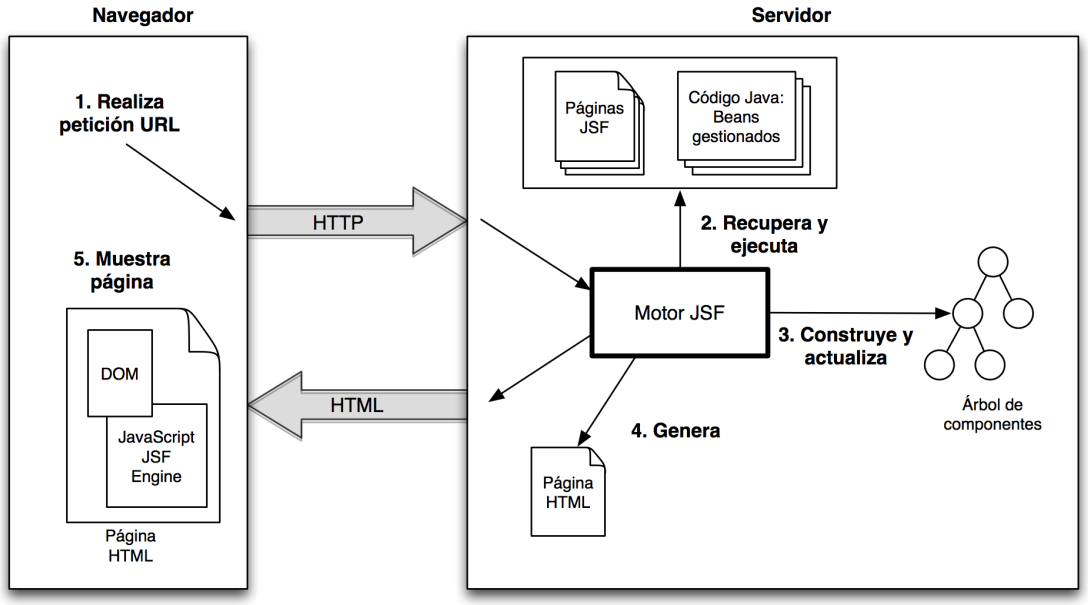
Existen frameworks JavaScript muy populares, como jQuery, que facilitan la tarea de definir la interfaz de usuario de la aplicación. Aun así, el mantenimiento y testeo de este código se hace complicado. Hay muchos elementos implicados: la comunicación con el servidor, las páginas HTML en las que está embebido y el propio código JavaScript. También se hace complicado reutilizar y compartir el código JavaScript entre distintas páginas y desarrolladores.

Como alternativa a la codificación en JavaScript aparecen los denominados *frameworks RIA basados en el servidor*, en los que los desarrolladores no escriben directamente JavaScript, sino que utilizan componentes de alto nivel (paneles, menús desplegables,

combos, etc.) que el servidor inserta en las páginas resultantes. Este enfoque es el que utilizan frameworks populares como JSF (*JavaServer Faces*), GWT (*Google Web Toolkit*) o ZK.

En este enfoque el desarrollador compone la interfaz de usuario utilizando un lenguaje de componentes específico. Esta definición reside en el servidor en forma de página de texto. Cuando el servidor recibe una petición realiza un procesamiento de esta página de texto y genera otra que contiene todos los componentes de la interfaz en formato HTML y JavaScript y que se envía de vuelta al navegador.

En el caso de JSF 2 la definición de la interfaz se realiza en forma de páginas XHTML con distintos tipos de etiquetas que veremos más adelante. Estas páginas se denominan *páginas JSF*. La siguiente figura muestra el funcionamiento de JSF para generar una página por primera vez.



Generación de una página JSF

El navegador realiza una petición a una determinada URL en la que reside la página JSF que se quiere mostrar. En el servidor un servlet que llamamos *motor de JSF* recibe la petición y construye un árbol de componentes a partir de la página JSF que se solicita. Este árbol de componentes replica en forma de objetos Java la estructura de la página JSF original y representa la estructura de la página que se va a devolver al navegador. Por ejemplo, si en la página JSF se define un componente de tipo menú con la etiqueta

```
<h:selectOneMenu>
```

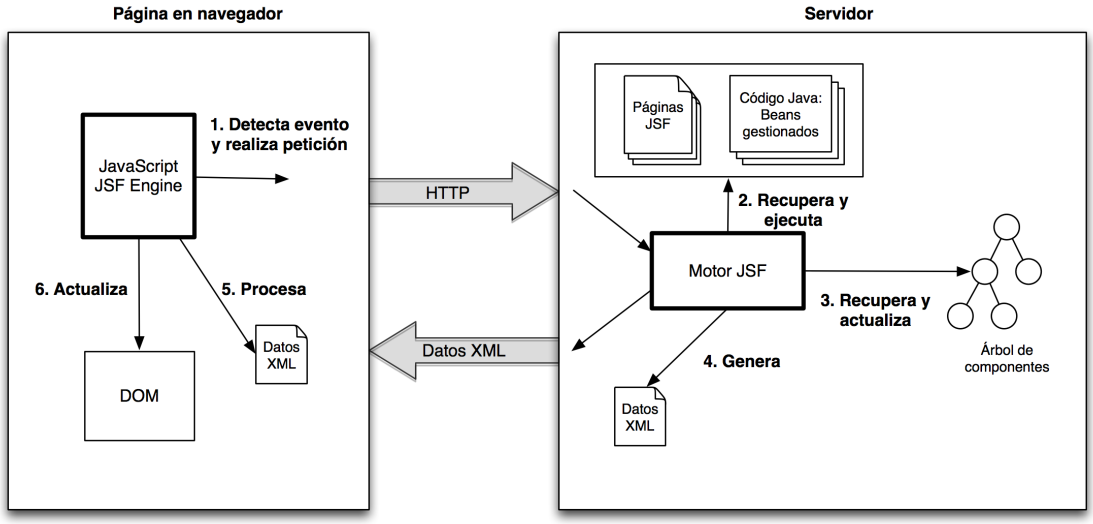
en el árbol de componentes se construirá un objeto Java de la clase

```
javax.faces.component.html.HtmlSelectOneMenu
```

Una vez construido el árbol de componentes, se ejecuta código Java en el servidor para rellenar los elementos del árbol con los datos de la aplicación. Por último, a partir del árbol de componentes se genera la página HTML que se envía al navegador.

Los componentes deben mostrar y recoger datos que se obtienen y se pasan a la aplicación. Este proceso se denomina ligado de datos o *data binding*. En los frameworks basados en el servidor el ligado de datos es sencillo porque la definición de la interfaz de usuario y los datos residen en un mismo sitio. En el caso de JSF 2 la forma de ligar datos con la interfaz es utilizando la anotación `@ManagedBean` en una clase Java para definir lo que se denomina como *bean gestionado*.

Una vez que la página se muestra en el navegador, el usuario interactúa con ella (por ejemplo, pulsando en un botón, escribiendo texto en un campo o pinchando en una opción de un menú desplegable). En este momento es cuando se utiliza el enfoque de Ajax. Los componentes contienen código JavaScript que se encarga de gestionar la interacción y pasar al servidor las peticiones específicas. Estas peticiones no tienen por qué generar una nueva página. Muchas veces es suficiente con actualizar algún elemento del DOM de la página actual, siguiendo un enfoque similar a Ajax. La siguiente figura describe el proceso de actualización de una página JSF.



Actualización de una página JSF

El servidor recoge la petición, la procesa, recupera el árbol de componentes de la página y genera unos datos en formato XML que devuelve al cliente. Allí el código JavaScript los procesa y actualiza el DOM de la página.

Como resumen, entre las ventajas del desarrollo basado en el servidor frente a la codificación directa en JavaScript destacamos:

- Más fácil de portar a distintos navegadores y plataformas (móviles, por ejemplo).
- Más sencillo de utilizar: es muy fácil acceder a los objetos de negocio y de datos

desde la presentación.
- Mayor robustez y seguridad.
- Mayor facilidad de mantener, probar y modificar.

1.2. Características de JSF

JSF es un framework MVC (Modelo-Vista-Controlador) basado en el API de Servlets que proporciona un conjunto de componentes en forma de etiquetas definidas en páginas XHTML mediante el framework Facelets. Facelets se define en la especificación 2 de JSF como un elemento fundamental de JSF que proporciona características de plantillas y de creación de componentes compuestos. Antes de la especificación actual se utilizaba JSP para componer las páginas JSF.

En la siguiente sesión explicaremos con más profundidad las características MVC de JSF. Por ahora basta con adelantar que JSF utiliza las páginas Facelets como vista, objetos Javabean como modelos y métodos de esos objetos como controladores. El servlet FacesServlet realiza toda la tediosa tarea de procesar las peticiones HTTP, obtener los datos de entrada, validarlos y convertirlos, colocarlos en los objetos del modelo, invocar las acciones del controlador y renderizar la respuesta utilizando el árbol de componentes.

Entrando un poco más en detalle, JSF proporciona las siguientes características destacables:

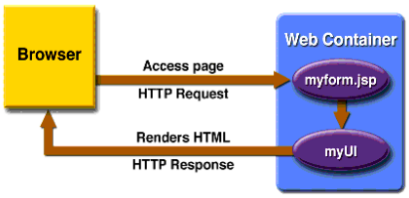
- Definición de las interfaces de usuario mediante **vistas** que agrupan **componentes** gráficos.
- Conexión de los componentes gráficos con los datos de la aplicación mediante los denominados **beans gestionados**.
- Conversión de datos y validación automática de la entrada del usuario.
- Navegación entre vistas.
- Internacionalización
- A partir de la especificación 2.0 un modelo estándar de comunicación Ajax entre la vista y el servidor.

Tal y como hemos comentado, JSF se ejecuta sobre la tecnología de Servlets y no requiere ningún servicio adicional, por lo que para ejecutar aplicaciones JSF sólo necesitamos un contenedor de servlets tipo Tomcat o Jetty.

Para entender el funcionamiento de JSF es interesante compararlo con JSP. Recordemos que una página JSP contiene código HTML con etiquetas especiales y código Java. La página se procesa en una pasada de arriba a abajo y se convierte en un servlet. Los elementos JSP se procesan en el orden en que aparecen y se transforman en código Java que se incluye en el servlet. Una vez realizada la conversión, las peticiones de los usuarios a la página provocan la ejecución del servlet.

En JSF el funcionamiento es distinto. Una página JSF también contiene etiquetas especiales y código HTML, pero su procesamiento es mucho más complicado. La

diferencia fundamental con JSP es el resultado del procesamiento interno, en el servidor, de la página cuando se realiza la petición. En JSP la página se procesa y se transforma en un servlet. En JSF, sin embargo, el resultado del procesamiento es un árbol de componentes, objetos Java instanciados el servidor, que son los que posteriormente se encargan de generar el HTML.



Con un poco más de detalle, cuando el usuario realiza una petición a la página JSF se realizan las siguientes acciones en orden:

- (1) Se procesa la página de arriba abajo y se crea un **árbol de componentes** JSF en forma de objetos instanciados de clases del framework JSF.
- (2) Se obtienen los valores introducidos por el usuario y se actualizan los beans gestionados con ellos.
- (3) Se actualizan los componentes con los valores procedentes de las propiedades de los beans gestionados.
- (4) Se pide a los componentes que se rendericen, generándose el código HTML que se envía de vuelta al navegador como resultado de la petición.
- (5) El árbol de componentes JSF se guarda en memoria para que posteriores peticiones a la misma página JSF no tengan que crearlo, sino que utilicen el existente.

¿Cómo se genera el HTML final de la respuesta a la petición? En JSP el servlet genera el HTML mediante sentencias embebidas en su código que escriben en el *stream* de salida. En JSF, la página HTML se genera como resultado de llamadas a métodos del árbol de componentes (en JSF se habla de realizar un *render* del componente). Una ventaja del enfoque de JSF es que el renderizado de la interfaz de usuario resultante es más flexible. De hecho, es posible generar con el mismo JSF distinto código para distintos dispositivos. Por ejemplo, es posible utilizar la misma aplicación JSF para servir páginas a navegadores web y dispositivos móviles.

1.3. Evolución de JSF

1.3.1. Especificaciones de JSF

JavaServer Faces es el framework oficial de Java Enterprise para el desarrollo de interfaces de usuario avanzadas en aplicaciones web. La especificación de JSF ha ido evolucionando desde su lanzamiento en 2004 y se ha ido consolidando, introduciendo nuevas características y funcionalidades.

La especificación original de JavaServer Faces (1.0) se aprobó en marzo del 2004, con la *Java Specification Request* JSR 127. En esta especificación se define el funcionamiento básico de JSF, introduciéndose sus características principales: uso de beans gestionados el lenguaje JSF EL, componentes básicos y navegación entre vistas.

La especificación JavaServer Faces 1.2 se aprobó en mayo del 2006. La JSR donde se define es la JSR 252. En esta especificación se introducen algunas mejoras y correcciones en la especificación 1.1. Una de las principales es la introducción del lenguaje unificado de expresiones (*Unified Expression Language* o *Unified EL*), que integra el lenguaje JSTL de expresiones de JSP y el lenguaje de expresiones de JSF en una única especificación.

La especificación actual de JSF (JavServer Faces 2.1) se aprobó en octubre de 2010 (JSR 314) junto con el resto de especificaciones que definen Java EE 6.0. En esta especificación se rompe definitivamente con JSP como forma de definir las vistas JSF y se introduce un formato independiente de JSP. De hecho, la implementación de referencia de Oracle/Sun se integra con Facelets, un sistema de definición de plantillas para las páginas web que sustituye a JSP y que se ha popularizado con JSF 1.2.

Otras características importantes de esta especificación son :

- Soporte para Ajax
- Componentes múltiples
- Integración con Facelets
- Gestión de recursos (hojas de estilo, imágenes, etc.)
- Facilidad de desarrollo y despliegue

Por la cantidad de comentarios que está generando, parece ser que esta nueva versión va a representar un salto cualitativo importante y va a conseguir popularizar definitivamente JSF. Esta es la especificación con la que vamos a trabajar en este módulo.

Las siguientes URLs son muy útiles cuando se está programando en JSF.

- JavaDoc del API de JSF 2: http://javaserverfaces.java.net/nonav/docs/2.1/javadocs/
- Anotación `@ManagedBean`: http://javaserverfaces.java.net/nonav/docs/2.0/managed-bean-javadocs/index.html
- Etiquetas de JSF y Facelets: http://javaserverfaces.java.net/nonav/docs/2.1/vdldocs/facelets/index.html

1.3.2. JSF y otros frameworks de presentación

En la actualidad JSF se ha convertido una alternativa ya madura digna de ser tenida en cuenta si necesitamos que nuestra aplicación web tenga un interfaz rico.

Algunas de las características más importantes de JSF en la actualidad:

- Framework estándar definido en la especificación Java EE.
- Soporte en todos los servidores de aplicaciones y en las herramientas de desarrollo:

Eclipse, GlassFish, etc.
- Entornos gráficos para desarrollar rápidamente aplicaciones JSF.
- Gran variedad de implementaciones de componentes.
- Fácil integración con frameworks en la capa de negocio y de persistencia: Spring, JPA, etc.
- Comunidad muy activa en la actualidad; podemos encontrar fácilmente soporte en foros, artículos, tutoriales, etc.

Todo esto, sumado a la madurez demostrada con la especificación 2.0 hace que JSF sea una tecnología que tenemos que conocer obligatoriamente y que plateemos seriamente su utilización en proyectos Java EE.

Ahora bien, como con cualquier otra tecnología y framework novedoso, antes de lanzarse a ello hay que tomar ciertas precauciones. JSF es una tecnología complicada, con una curva de aprendizaje bastante pronunciada, y debemos meditar bien si es apropiado utilizarla o no.

Lo primero que debemos plantearnos es si nuestro proyecto necesita una interfaz de usuario rica. Si estamos desarrollando una aplicación web de gestión de contenidos y páginas HTML, lo más probable es que JSF nos venga grande. Sería suficiente alguna herramienta para gestionar plantillas como Velocity, Tiles o el mismo Facelets junto con JSP.

Si nuestro proyecto va a necesitar formularios, rejillas de datos, gráficas estadísticas generadas dinámicamente, árboles de navegación, pestañas, validación de los datos introducidos por el usuario, internacionalización y demás características avanzadas, entonces sí que debemos considerar JSF.

Uno de los problemas que nos podemos encontrar en JSF es que necesitemos algún componente específico que no exista en el conjunto de componentes que estemos utilizando. Puede ser que queramos definir alguna funcionalidad específica en nuestra web (un canvas, por ejemplo, en el que el usuario dibuja algunos sencillos trazos) que no esté soportada por ningún conjunto de componentes, o que queramos integrar en ella algún servicio Ajax de terceros, como Google Maps. Dado que JSF se basa en componentes definidos previamente que insertamos en las páginas web de forma declarativa (utilizando etiquetas XML), tendríamos que construir ese componente nosotros mismos. Hay bastante información sobre cómo hacerlo, pero es un tema avanzado que no vamos a ver en este curso.

Entre las alternativas más sólidas a JSF en la actualidad se encuentran GWT (Google Web Toolkit), Java FX y Flex. Todas ellas se pueden comunicar perfectamente con la aplicación Java en el servidor, si hemos utilizado una correcta arquitectura y hemos separado nuestra aplicación en capas. Algunas de las características de estas tecnologías son las siguientes:

- GWT (Google Web Toolkit) es un proyecto open source de Google para desarrollar aplicaciones de internet ricas en Java basadas en Ajax. El enfoque de GWT es

radicalmente distinto al de JSF. La interfaz de usuario se desarrolla en Java (en lugar de definirse estáticamente en un fichero XML) como si estuviéramos programando en Swing. La conexión con el servidor se realiza utilizando un API de GWT de llamadas asíncronas a procedimientos remotos. El código que define la interfaz de usuario cliente se compila a código JavaScript compatible con todos los navegadores con las herramientas suministradas por Google. Es una alternativa a JSF muy interesante.

- Java FX es la última propuesta de Oracle/Sun para desarrollar aplicaciones RIA (*Rich Internet Applications*) que podemos ejecutar en el navegador o en el escritorio. Sun invertió mucho esfuerzo en el desarrollo del framework y ahora en su divulgación y promoción. Los resultados y las aplicaciones ejemplo que muestran en el sitio web son bastante espectaculares. Es una tecnología que merece la pena conocer si se quiere desarrollar una aplicación totalmente interactiva, que se salga del navegador y que necesite conexión a servicios web externos proporcionados por un servidor. Con la actualización 10 de Java 6 se intenta dar un nuevo empujón a los applets, introduciendo la posibilidad de arrastrar y soltar applets del navegador al escritorio.
- Flex es la propuesta de Adobe basada en Flash para realizar aplicaciones RIA. Es posible integrarlo con código Java en el servidor. Es posible usarlo tanto para desarrollar aplicaciones completas, como para definir pequeños componentes con efectos interactivos en nuestras páginas web. Una de las ventajas de Flex frente a Java FX es que Flash está instalado casi en la totalidad de navegadores, lo que hace la aplicación muy portable.

1.3.3. Implementaciones y componentes JSF

Una vez que hemos decidido utilizar JSF para nuestro proyecto tenemos que tomar una decisión complicada. ¿Qué conjunto de componentes utilizar? Existen multitud de implementaciones de componentes.

JSF es una especificación y, como tal, existen distintas implementaciones. Sun siempre proporciona una implementación de referencia de las tecnologías Java, que incluye en el servidor de aplicaciones GlassFish. En el caso de JSF, la implementación de referencia es las dos implementaciones más usadas son:

- Mojarra es la especificación de referencia realizada por Sun. Es una de las implementaciones más populares y se incluye en distintos servidores de aplicaciones Java Enterprise, entre los que se encuentran GlassFish y JBoss. Sun la mantiene activamente y se puede descargar desde esta página.
- MyFaces es la implementación abierta de la Fundación Apache. Está desarrollada por la comunidad Apache y está incluida en el servidor de aplicaciones Geronimo. Se puede descargar desde esta página.

Las implementaciones de la especificación JSF proporcionan el framework y los componentes básicos (etiquetas `h:` y `f:`). Para utilizar JSF en una aplicación avanzada necesitaremos componentes más elaborados.

En la página web www.jsfmatrix.net se encuentra una tabla resumen muy útil en la que se

comparan las características de distintos conjuntos de componentes para JSF (más de 20). Los autores de la página escribieron un interesante artículo en The Server Side comparando tres conjuntos de componentes específicos. Este artículo dio origen a la tabla comparativa.

Podemos destacar los siguientes:

- RichFaces. Desarrollado por Exadel y comprado por JBoss para su servidor de aplicaciones JBoss. Componentes Ajax muy elaborados y con una detallada documentación. Muy recomendable. Licencia open source LGPL.
- Icefaces. Conjunto de componentes muy maduro, usado por gran cantidad de aplicaciones web de importantes empresas. Ha sido escogido por Sun como conjunto de componentes adicional a los básicos, sustituyendo un proyecto interno de Sun recién cancelado (Proyecto Woodstock). Licencia open source Mozilla Public License, version 1.1.
- MyFaces Tomahawk. Desarrollado por el proyecto MyFaces de Apache. También maduro y estable. Licencia open source Apache Software License, version 2.0.

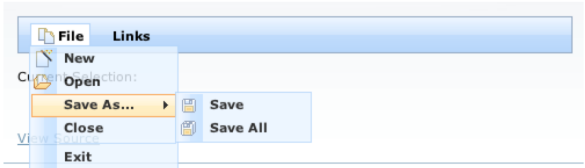
En las siguientes imágenes, se puede observar un ejemplo de la apariencia de algunos componentes de RichFaces y de Icefaces. Hay que tener en cuenta que la apariencia final del componente en una misma implementación depende también del *skin* escogido, que, a su vez, se implementa con hojas de estilo CSS.

Richfaces:

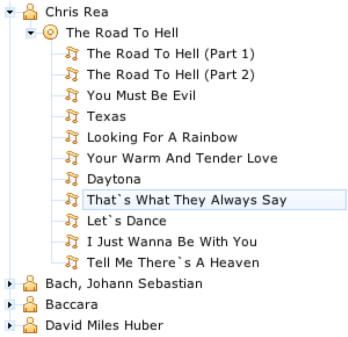
Implementación RichFaces de JBoss de JSF

	Expenses			subtotals
	Meals	Hotels	Transport	
San Jose				
25-Aug-97	$37.74	$112.00	$45.00	
26-Aug-97	$27.28	$112.00	$45.00	
	$65.02	$224.00	$90.00	$379.02
Seattle				
27-Aug-97	$96.25	$109.00	$36.00	
28-Aug-97	$35.00	$109.00	$36.00	
	$131.25	$218.00	$72.00	$421.25
Totals	**$196.27**	**$442.00**	**$162.00**	**$800.27**

Componente **Grid**

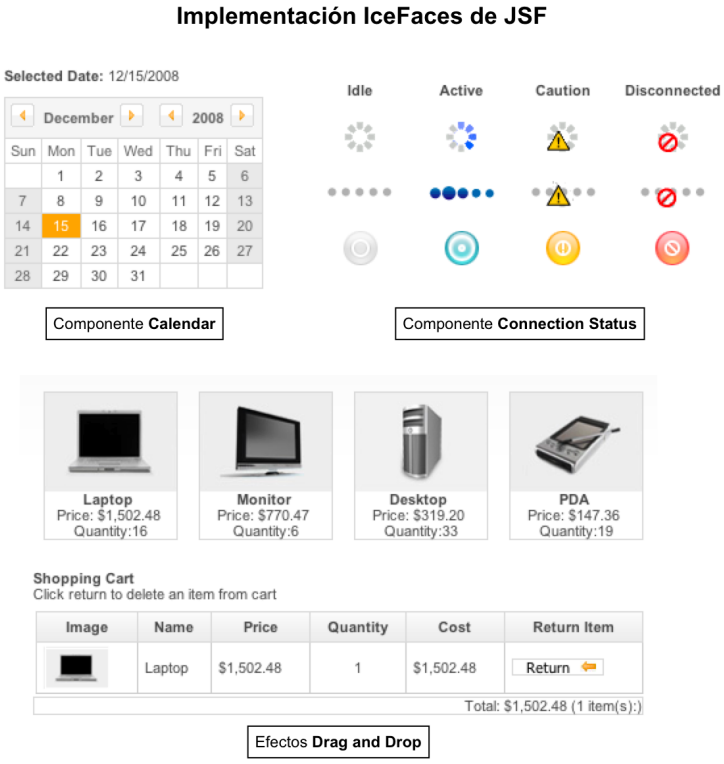


Componente **Drop Down Menu**



Componente **Tree**

Icefaces:



1.4. El lenguaje de definición de vistas JSF

Al igual que JSP, las vistas JSF se construyen mediante etiquetas específicas que declaran elementos y componentes. Un ejemplo de página JSF es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
      <title>Welcome</title>
   </h:head>
   <h:body>
      <h:form>
         <h3>Please enter your name and password.</h3>
         <table>
            <tr>
               <td>Name:</td>
               <td><h:inputText value="#{user.name}"/></td>
            </tr>
            <tr>
               <td>Password:</td>
               <td><h:inputSecret value="#{user.password}"/></td>
            </tr>
         </table>
         <p><h:commandButton value="Login" action="welcome"/></p>
```

```
      </h:form>
   </h:body>
</html>
```

El ejemplo muestra una pantalla de login en la se pide al usuario el nombre y la contraseña. El aspecto de la página en el navegador es el siguiente:

En las especificaciones anteriores a JSF 2.0 (la actual es la versión 2.1), las páginas de JSF se construían utilizando páginas JSP con etiquetas específicas de JSF. Pronto se comprobó que el enfoque no era el correcto. La tecnología JSP no tenía la potencia sufiente para las funcionalidades que se intentaban implementar en JSF. Además, ambos conjuntos de etiquetas tenían incompatibilidades y era complicado combinarlas de forma sencilla.

La especificación 2 soluciona el problema utilizando XHTML como el lenguaje en el que se definen las páginas. Repasemos rápidamente qué es el XHTML y cuáles son sus ventajas frente al HTML tradicional.

El lenguaje XHTML es una normalización del HTML orientada a hacerlo compatible con el formato XML. Expresándolo en pocas palabras, un documento XHTML es un documento HTML formateado en forma de documento XML. Por ejemplo, en los documentos XML toda etiqueta debe empezar y terminar. En HTML, no sucede siempre así. El ejemplo más común es la etiqueta de fin de línea `<br>`. En HTML es una etiqueta correcta. Sin embargo en XHTML se transforma añadiéndole la terminación (`<br/>`) para cumplir el estándar XML.

Debido a que están escritos en XML, los documentos XHTML son más fáciles de validar y de procesar. Se pueden editar con herramientas genéricas orientadas a XML. Y se pueden transformar utilizando hojas de estilo y otras características de XML.

Una de las características del XHTML es la posibilidad de utilizar en un mismo documento distintos lenguajes de etiquetas utilizando espacios de nombres. Por ejemplo, un documento XHTML puede contener dibujos definidos en SVG o ecuaciones definidas en MathML. Cada lenguaje tiene su propio espacio de nombres definido con un prefijo

distinto. Es la forma de evitar conflictos entre etiquetas iguales de distintos lenguajes. Esta característica es la que usa la especificación JSF para permitir que en las páginas XHTML coexistan distintos tipos de etiquetas.

Los espacios de nombres se especifican al comienzo del documento XHTML definiendo el prefijo que utilizan las etiquetas de cada uno de los lenguajes. Utilizando distintos prefijos para distintos lenguajes se elimina el problema de la posible ambiguedad a la hora de procesar el documento.

La siguiente tabla muestra las distintas librerías de etiquetas que se utilizan en las páginas JSF, incluyendo las etiquetas de la implementación RichFaces que es la que utilizaremos en el módulo.

Librería de etiquetas	Descripción	Ejemplo
JSTL Core	Etiquetas estándar JSP	`<c:forEach>` `<c:catch>`
JSTL Functions	Funciones estándar JSTL	`<fn:toUpperCase>` `<fn:toLowerCase>`
Facelets	Lenguaje de plantillas	`<ui:component>` `<ui:insert>`
JSF HTML	Componentes estándar JSF basados en HTML	`<h:body>` `<h:inputText>`
JSF Core	Componentes específicos de JSF	`<f:actionListener>` `<f:attribute>`
RichFaces	Componentes específicos de RichFaces	`<rich:dataTable>` `<rich:panel>`
Ajax RichFaces	Funciones Ajax de RichFaces	`<a4j:poll>` `<a4j:commandButton>`

Cada librería se declara con un prefijo distinto. La siguiente tabla muestra los distintos prefijos y URIs.

Librería de etiquetas	Prefijo	URI
JSTL Core	`c:`	`http://java.sun.com/jsp/jstl/core`
JSTL Functions	`fn:`	`http://java.sun.com/jsp/jstl/functions`
JSF Facelets	`ui:`	`http://java.sun.com/jsf/facelets`
JSF HTML	`h:`	`http://java.sun.com/jsf/htm`
JSF Core	`f:`	`http://java.sun.com/jsf/core`
RichFaces	`rich:`	`http://richfaces.org/rich`

Ajax RichFaces	a4j:	http://richfaces.org/a4j

Estos lenguajes de etiquetas se definen al comienzo del documento XHTML. Sólo es necesario declarar los que se utilizan en el documento. El siguiente ejemplo muestra una cabecera de un documento en el que se declaran todos los lenguajes de etiquetas que se pueden utilizar en las páginas RichFaces.

```
<!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"
  xmlns:a4j="http://richfaces.org/a4j"
  xmlns:rich="http://richfaces.org/rich">
```

1.4.1. Facelets

Facelets es un framework basado en el servidor que permite definir la estructura general de las páginas (su *layout*) mediante plantillas. Se trata de un framework similar a Tiles, Velocity, Tapestry o Sitemesh.

Facelets se adapta perfectamente al enfoque de JSF y se incorpora a la especificación en la última revisión 2.1. Anteriormente, las páginas JSF se definían utilizando etiquetas específicas de JSP, lo que generaba cierta confusión porque se trata de enfoques alternativos para un mismo problema. La sustitución de JSP por Facelets como lenguaje básico para definir la disposición de las páginas permite separar perfectamente las responsabilidades de cada parte del framework. La estructura de la página se define utilizando las etiquetas Facelets y los componentes específicos que deben presentar los datos de la aplicación utilizando etiquetas JSF.

Entre las características de Facelets destacan:

- Definición de plantillas (como en Tiles)
- Composición de componentes
- Etiquietas para definir funciones y lógica
- Desarrollo de páginas amistoso para el diseñador
- Posibilidad de crear librerías de componentes

Para usar los tags de facelets, habrá que añadir la siguiente declaración de namespace en nuestra página JSF:

```
xmlns:ui="http://java.sun.com/jsf/facelets"
```

La siguiente tabla explica brevemente las etiquetas definidas por Facelets. Por falta de tiempo en el módulo nos centraremos únicamente en las etiquetas que se utilizan para definir plantillas: `<ui:include>`, `<ui:composition>`, `<ui:define>` y `<ui:insert>`.

Etiqueta	Descripción

`<ui:include>`	Incluye contenido de otro fichero XHTML. Permite la reutilización de contenido para múltiples vistas
`<ui:composition>`	Cuando se usa sin el atributo `template` una composición es una secuencia de elementos que se pueden insertar en algún otro lugar (mediante la etiqueta `<ui:include>` por ejemplo). La composición puede tener partes variables especificadas con el elemento hijo `<ui:insert>`. Cuando se usa con el atributo `template`, se carga la plantilla especificada. Los elementos hijos (etiquetas `<ui:define>`) determinan las partes variables de la plantilla. El contenido de la plantilla reemplaza esta etiqueta.
`<ui:decorate>`	Cuando se usa sin el atributo `template` especifica una página en la que se pueden insertar otras partes. Las partes variables se especifican con el elemento hijo `<ui:insert>`.
`<ui:define>`	Define el contenido que se inserta en una plantilla con un `<ui:insert>` que empareja.
`<ui:insert>`	Define un lugar en el que se va a insertar contenido en una plantilla. El contenido se define en la etiqueta que carga la plantilla utilizando la etiqueta `<ui:define>`.
`<ui:param>`	Especifica un parámetro que se pasa a un fichero incluido o a una plantilla.
`<ui:component>`	Esta etiqueta es idéntica a `<ui:composition>`, excepto en que crea un componente que se añade al árbol de componentes.
`<ui:fragment>`	Es idéntica a `<ui:decorate>` excepto que crea un componente que se añade al árbol de coponentes.
`<ui:debug>`	Permite mostrar al usuario una ventana de depuración que muestra la jerarquía de componentes de la página actual y las variables en el ámbito de la aplicación.
`<ui:remove>`	JSF elimina todo lo que hay dentro de una etiqueta `<ui:remove>`.
`<ui:repeat>`	Itera sobre una lista, array, result set o un objeto individual.

1.4.2. Definiendo plantillas con Facelets

Facelets nos permite usar un mecanismo de plantillas para encapsular los comoponentes comunes en una aplicación. Así también podremos modificar el aspecto de nuestra página aplicando cambios sobre la plantilla, y no individualmente sobre las páginas.

Veamos ahora cómo definir la plantilla de una aplicación. Supongamos que queremos definir una plantilla con la siguiente estructura: un menú a la izquierda, una cabecera, un pie de página y un contenido variable, en función de la opción del menú que se ha

pulsado. Para el usuario siempre se va a estar en la misma página y sólo va a cambiar el contenido del centro de la pantalla. El aspecto de la página que queremos construir es el que aparece en la siguiente figura:

La plantilla la definimos en el fichero `resources/templates/principal.xhtml`. Es una página XHTML que utiliza una tabla para componer la disposición y que contiene la etiqueta `<ui:insert>` para definir el contenido variable. La cabecera y el pie de página se definen en la propia página. Por modularidad el menú se define en una página separada que se incluye con la instrucción `<ui:include>`.

La página se define en el fichero `templates/principal.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:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">

    <h:head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"
/>
        <link href="./resources/css/default.css" rel="stylesheet"
type="text/css" />
        <link href="./resources/css/cssLayout.css" rel="stylesheet"
type="text/css" />
        <title>
          <ui:insert name="pageTitle">JSF Twitter</ui:insert>
```

```
        </title>
    </h:head>

    <h:body>

        <div id="top">
            <ui:insert name="top">
                <h1>JSF Twitter</h1>
            </ui:insert>
        </div>
        <div>
            <div id="left">
                <ui:insert name="left">
                    <ui:include
src="/resources/components/leftMenu.xhtml"></ui:include>
                </ui:insert>
            </div>
            <div id="content" class="left_content">
                <ui:insert name="content">
                    Aquí va el contenido. Si no hay contenido, se mostrará
este texto por defecto
                </ui:insert>
            </div>
        </div>
        <div class="clearboth"></div>
        <div id="bottom">
            &copy; Experto en Desarrollo de Aplicaciones y Servicios con
Java Enterprise
        </div>

    </h:body>

</html>
```

Ya tenemos definida nuestra plantilla con sus cuatro elementos: cabecera, menú, contenido y pie

Dentro de cada fragmento, podemos especificar un contenido por defecto, cosa que hemos hecho tanto en el menú como en el contenido. Sin embargo, la manera de añadir contenido por defecto se ha hecho de dos formas distintas:

- En el caso del menú, hemos hecho uso de la etiqueta `ui:include` para incluir contenido de otra página
- En el caso del bloque de contenido, hemos incluido directamente el código HTML del mismo

Ambas maneras son permitidas por Facelets. Sin embargo, el uso de la etiqueta `ui:include` nos permite una mayor independencia de la plantilla a la hora de modificar fragmentos.

El siguiente fichero `resources/components/leftMenu.xhtml` contiene el menú de la aplicación. Utiliza la directiva `<ui:composition>` para definir un bloque de código que es insertado desde otra página con la directiva `<ui:include>`. Dentro utilizamos el componente RichFaces `<rich:panel>` para definir un panel con el título *Menú*

```
<?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:f="http://java.sun.com/jsf/core"
    xmlns:rich="http://richfaces.org/rich">

    <ui:composition>
      <div class="left_menu">
        <div>
          <div class="foto_usuario">
            <img src="#{user.profile_image_url}" /><br/>
            user.name}<br/>(#{user.screenName})
          </div>
          <div class="clearboth"></div>
        </div>
        <div>
          <ul>
            <li>Tweets: #{user.tweets}</li>
            <li>Siguiendo a: #{user.following}</li>
            <li>Seguidores: #{user.followers}</li>
          </ul>
        </div>
      </div>
    </ui:composition>

</html>
```

Por último, definimos el fichero principal `timeline.xhtml`: que incluye la plantilla con la instrucción `<ui:composition>` y el atributo `template` en el que se indica la ruta de la plantilla. Con la directiva `<ui:define>` definimos el contenido del panel principal, utilizando el atributo `name` con un valor que debe emparejar con el mismo atributo de la directiva `<ui:insert>` en el que se coloca.

```
<?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:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:rich="http://richfaces.org/rich"
    xmlns:c="http://java.sun.com/jsp/jstl/core">

  <body>
    <ui:composition
template="./resources/templates/twitterTemplate.xhtml">
      <ui:define name="content">
        <h:form>
          <div>
            <h:commandButton
                action="#{timelineBean.getPrevPage()}"
                value="Más recientes" />
            <ui:repeat
                var="tweet"
                value="#{timelineBean.getTweets()}"
                offset="#{timelineBean.currentPage}"
                size="#{timelineBean.perPage}">
              <div class="tweet">
                <div class="tweet-wrapper">
                  <div class="userpic">
                    <img src="#{tweet.user.profile_image_url}" />
                  </div>
```

```
                    <div>#{tweet.user.name}
(@#{tweet.user.screenName})</div>
                    <div>#{tweet.text}</div>
                  </div>
                </div>
              </ui:repeat>
            </div>
            <h:commandButton
                action="#{timelineBean.getNextPage()}"
                value="Siguiente página" />
          </h:form>
        </ui:define>
      </ui:composition>
    </body>
</html>
```

Se utiliza una hoja de estilo CSS en la que se define el tamaño de fuente del título y del pie de página y las dimensiones y separación del menú y la zona principal. Lo hacemos en el fichero `resources/css/detault.css`:

```
body {
    background-color: #ffffff;
    font-size: 12px;
    font-family: Verdana, Arial, sans-serif;
    color: #000000;
    margin: 10px;
}

h1 {
    font-family: Verdana, Arial, sans-serif;
    border-bottom: 1px solid #AFAFAF;
    font-size:  16px;
    font-weight: bold;
    margin: 0px;
    padding: 0px;
    color: #D20005;
}

/* RESTO DEL CSS*/
```

Para acceder a la página debemos utilizar el prefijo `faces` accediendo a *http://localhost:8080/jsf-ejemplos/faces/timeline.xhtml*.

También podemos definir un fichero `index.jsp` en la raíz de la aplicación web que haga una redirección a la página anterior utilizando la directiva `jsp:forward`:

```
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html>
<head></head>
<body>
   <jsp:forward page="/faces/timeline.xhtml" />
</body>
</html>
```

Debemos incluir el fichero `index.jsp` en el primer lugar de la lista de ficheros de bienvenida en `WEB-INF/web.xml`:

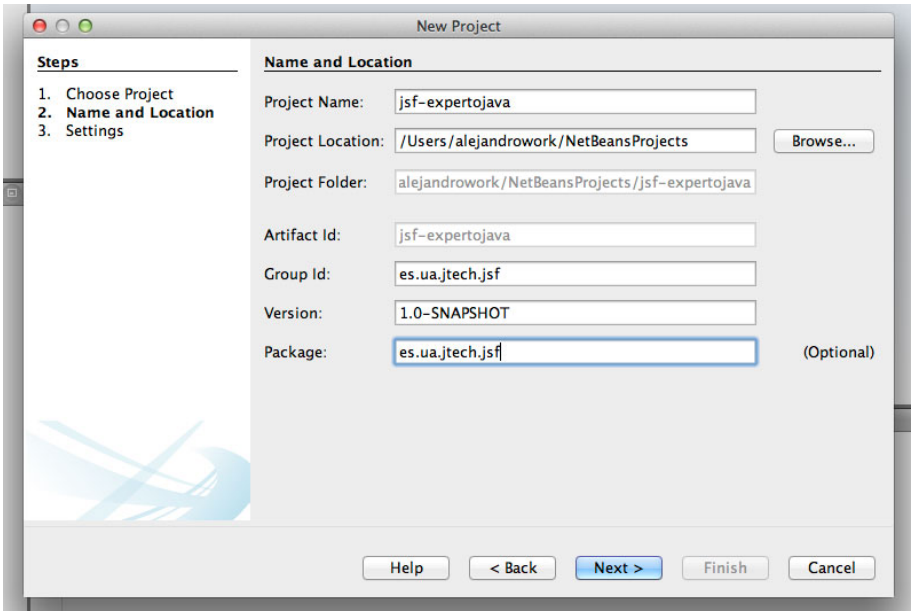
```
...
<welcome-file-list>
   <welcome-file>index.jsp</welcome-file>
```

```
   <welcome-file>faces/timeline.xhtml</welcome-file>
</welcome-file-list>
...
```

1.5. Creando un primer proyecto JSF

En esta sección vamos a detallar cómo crear un proyecto JSF utilizando Maven y la implementación de JSF RichFaces. En el momento de escribir estos apuntes, JBoss está terminando el desarrollo de la versión 4.2, la versión que soporta completamente la especificación 2.1 de JSF.

Para crear nuestro primer proyecto JSF, lo primero que haremos será crear un proyecto web en Netbeans con Maven, del tipo *Web Application*. Las características de nuestro proyecto serán las que se muestran en el siguiente pantallazo:



Proyecto Netbeans

Posteriormente, tendremos que editar el fichero pom.xml para añadir las rutas del repositorio de RichFaces, así como las dependencias con sus librerías y las de JSR 303

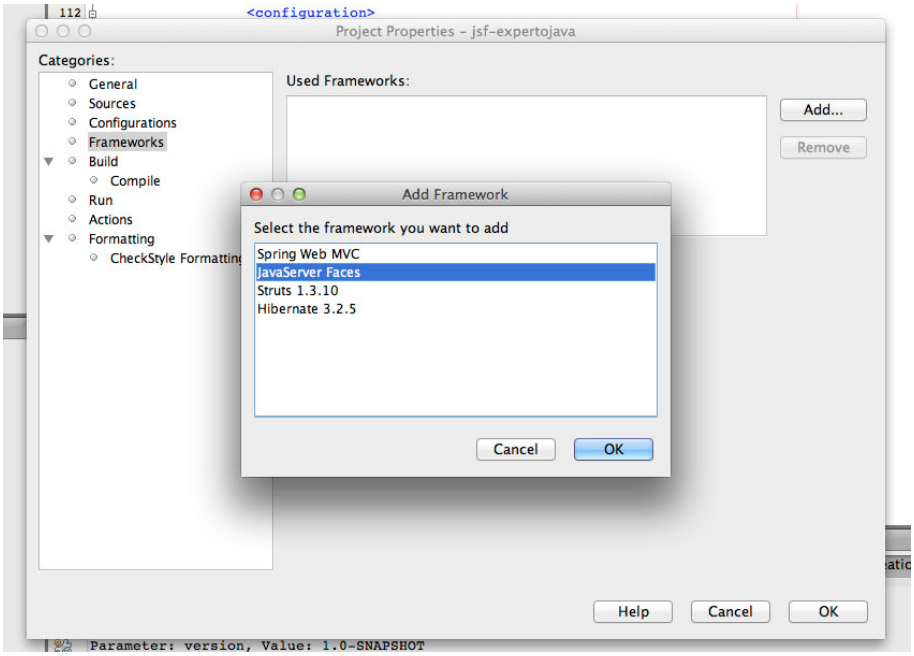
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>es.ua.jtech.jsf</groupId>
  <artifactId>ModJSF</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>

  <name>ModJSF</name>
```

```
  <!-- AÑADIR REPOSITORIOS DE JSF -->
  <repositories>
    <repository>
      <id>richfaces</id>
      <name>Richfaces repository</name>
      <layout>default</layout>
<url>https://repository.jboss.org/nexus/content/groups/public-jboss/</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>

  <properties>
    <endorsed.dir>${project.build.directory}/endorsed</endorsed.dir>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <!-- DEPENDENCIAS RICHFACES -->
    <dependency>
      <groupId>org.richfaces.core</groupId>
      <artifactId>richfaces-core-impl</artifactId>
      <version>4.2.0.Final</version>
    </dependency>
    <dependency>
      <groupId>org.richfaces.ui</groupId>
      <artifactId>richfaces-components-ui</artifactId>
      <version>4.2.0.Final</version>
    </dependency>
    <dependency>
      <groupId>org.richfaces.core</groupId>
      <artifactId>richfaces-core-api</artifactId>
      <version>4.2.0.Final</version>
    </dependency>
    <dependency>
      <groupId>org.richfaces.ui</groupId>
      <artifactId>richfaces-components-api</artifactId>
      <version>4.2.0.Final</version>
    </dependency>
    <dependency>
      <groupId>com.sun.faces</groupId>
      <artifactId>jsf-api</artifactId>
      <version>2.1.1-b04</version>
    </dependency>
    <dependency>
      <groupId>com.sun.faces</groupId>
      <artifactId>jsf-impl</artifactId>
      <version>2.1.1-b04</version>
    </dependency>

    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>jstl</artifactId>
      <version>1.1.2</version>
    </dependency>
    <dependency>
      <groupId>taglibs</groupId>
      <artifactId>standard</artifactId>
      <version>1.1.2</version>
    </dependency>
        <!-- DEPENDENCIAS SCRIBE -->
    <dependency>
      <groupId>org.scribe</groupId>
      <artifactId>scribe</artifactId>
```

```
    <version>1.3.2</version>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1-b02</version>
          <scope>provided</scope>
  </dependency>

  <!-- DEPENDENCIAS JSR 303 -->
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.3.0.Final</version>
  </dependency>
  <dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.2.2</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
          <compilerArguments>
            <endorseddirs>${endorsed.dir}</endorseddirs>
          </compilerArguments>
        </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <version>2.1.1</version>
      <configuration>
        <failOnMissingWebXml>false</failOnMissingWebXml>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <version>2.1</version>
      <executions>
        <execution>
          <phase>validate</phase>
            <goals>
              <goal>copy</goal>
          </goals>
          <configuration>
            <outputDirectory>${endorsed.dir}</outputDirectory>
            <silent>true</silent>
            <artifactItems>
              <artifactItem>
                <groupId>javax</groupId>
                <artifactId>javaee-endorsed-api</artifactId>
                <version>6.0</version>
                <type>jar</type>
              </artifactItem>
            </artifactItems>
```

```
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Una vez hecho esto, editaremos las propiedades del proyecto, y nos iremos al apartado Frameworks. Añadiremos JavaServer Faces 2.1 y aceptaremos



Proyecto Netbeans

Volviendo a editar, verificamos que, además, se nos han añadido las librerías de RichFaces. En caso contrario, añadiremos las dependencias faltantes y actualizaremos nuestro proyecto.

1.6. Una aplicación de ejemplo en detalle

Las aplicaciones JSF utilizan la tecnología de servlets y pueden ser ejecutadas en cualquier contenedor de aplicaciones web, como Tomcat 6.x. No es necesario un servidor de aplicaciones compatible con toda la especificación Java EE 5 o 6.

Una aplicación web JSF contiene los siguientes elementos específicos:

- Librerías de implementación de JSF
- Páginas JSF
- Directivas en el fichero `web.xml`
- Fichero `faces-config.xml`

- Clases Java con los beans gestionados

Vamos a detallar todos estos elementos, creando una sencilla aplicación de ejemplo que hace uso de RichFaces.

1.6.1. Librerías de implementación de JSF

Contienen el conjunto de clases que realizan la implementación del framework y las librerías de apoyo. Como en cualquier aplicación web, se guardan en el directorio `WEB-INF/lib` del WAR.

Las librerías dependen de la implementación específica de JSF. Por ejemplo, en el caso de la implementación de RichFaces (*RichFaces 4.2*) la lista de librerías es la siguiente:

```
cssparser-0.9.5.jar
ehcache-1.6.0.jar
guava-r05.jar
jsf-api-2.0.3-b03.jar
jsf-impl-2.0.3-b03.jar
jstl-1.2.jar
richfaces-components-api-4.0.0.20101110-M4.jar
richfaces-components-ui-4.0.0.20101110-M4.jar
richfaces-core-api-4.0.0.20101110-M4.jar
richfaces-core-impl-4.0.0.20101110-M4.jar
sac-1.3.jar
```

Vemos que existen librerías de apoyo (*cssparser*, *ehcache*, *guava* y *sac*) y librerías propias de JSF.

Entre las librerías de JSF podemos diferenciar dos tipos, las que realizan la implementación básica del estándar y las que definen componentes adicionales que se añaden a los componentes estándar de JSF. Las implementaciones más usadas de las librerías básicas son las del proyecto Mojarra de GlassFish de Oracle/Sun y las del subproyecto MyFaces Core de Apache. Ambas implementan la última especificación JSF 2.1. En ambos casos se definen dos módulos: `jsf-api-xxx.jar` y `jsf-impl-xxx.jar`.

Los componentes adicionales los proporcionan las distintas librerías de componentes desarrolladas por una gran cantidad de empresas. Las más populares son RichFaces, PrimeFaces y IceFaces, aunque cada día aparecen nuevas librerías muy interesantes como OpenFaces. Algunas de estas librerías de componentes son compatibles, y es posible utilizarlas conjuntamente. Por ejemplo, podemos desarrollar una aplicación en RichFaces y añadir algún componente adicional desarrollado por OpenFaces.

1.6.2. Páginas JSF

La aplicación ejemplo contendrá una única página JSF, que permite al usuario escribir en un campo y que muestra lo que va escribiendo a su derecha. La página (la llamaremos `sample.xhtml`) es la siguiente:

```
<!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"
   xmlns:a4j="http://richfaces.org/a4j">

<h:head>
   <title>RichFaces Sample</title>
   <meta http-equiv="content-type" content="text/xhtml; charset=UTF-8" />
</h:head>

<h:body>
   <h:form prependId="false">
      <h:outputLabel value="Name:" for="nameInput" />
      <h:inputText id="nameInput" value="#{richBean.name}">
         <a4j:ajax event="keyup" render="output" />
      </h:inputText>
      <h:panelGroup id="output">
         <h:outputText value="Hello #{richBean.name}!"
            rendered="#{not empty richBean.name}" />
      </h:panelGroup>
   </h:form>
</h:body>
</html>
```

Destacamos lo siguiente:

- El texto introducido por el usuario se guarda en el bean gestionado `richBean`, en concreto en su campo `name`.
- El componente `<h:form>` genera una etiqueta `<form>` de HTML en la que se incluye el campo de entrada `nameInput`. El atributo `prependId=false` hace que el motor de JSF no añada ningún prefijo a los identificadores de los componentes dentro del formulario. Esto reduce el tamaño de la página HTML generada.
- El texto de salida se encuentra dentro de un *PanelGroup* con el identificador `output`. El componente `<a4j:ajax>` hace que cada vez que se produzca el evento `keyup` en el campo de entrada se redibuje el panel y todos los componentes incluidos.
- El atributo `rendered` de un componente permite definir si se muestra o no. En este caso depende del valor devuelto por la expresión EL `#{not empty richBean.name}`.

1.6.3. Directivas en el fichero web.xml

El motor de JSF es el servlet que procesa las peticiones de los navegadores y realiza el procesamiento de las páginas JSF para generar el HTML resultante. Es necesario declarar este servlet en el fichero `web.xml` y definir ciertas propiedades de la aplicación web, como el mapeado de nombres o las páginas de bienvenida.

En concreto, el fichero `WEB-INF/web.xml` de la aplicación ejemplo es el siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0"
  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-app_3_0.xsd">
    <context-param>
        <param-name>javax.faces.PROJECT_STAGE</param-name>
```

```
        <param-value>Development</param-value>
    </context-param>
    <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>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
        <welcome-file>faces/index.xhtml</welcome-file>
    </welcome-file-list>
</web-app>
```

Podemos destacar los siguientes elementos:

- El servlet que procesa todas las peticiones es `javax.faces.webapp.FacesServlet`
- Se definen dos patrones de nombres de petición que se redirigen a ese servlet: `*.jsf` y `/faces/*`. Cuando se realiza una petición (por ejemplo, */faces/login/registro.xhtml* o */login/registro.jsf*) el servlet elimina el prefijo o la extensión *.jsf* y procesa la página XHTML definida en la ruta de la petición (*/login/registro.xhtml*).
- La página de bienvenida es la `faces/index.xhtml`. El navegador realiza la petición a esa ruta y se carga la página JSF `index.xhtml`

1.6.4. Fichero faces-config.xml

El fichero `faces-config.xml` define la configuración de la aplicación JSF:

- *Beans* de la aplicación, sus nombres y propiedades iniciales.
- Reglas de validación de los componentes de entrada.
- Reglas de navegación entre distintas páginas de la aplicación.
- Ficheros de recursos para la internacionalización de la aplicación.

Sin embargo, nos aprovecharemos de las anotaciones que JSF2 provee para no tener la necesidad de declarar nuestros Beans en este fichero, con lo que en este momento no es ni siquiera necesaria su existencia.

1.6.5. Clase Java con el bean gestionado

La clase Java que define el bean gestionado *richBean* es un sencillo JavaBean con la propiedad `name`:

A partir JSF 2.0 se hace todavía más sencillo la definición de un bean gestionado. Como hemos dicho, no es necesario hacerlo en el fichero `faces-config.xml`. Basta utilizar las anotaciones `@ManagedBean` y `@ViewScoped` en la propia clase. La primera define el

carácter del bean como gestionado y la segunda su ámbito (otros posibles valores son: `@RequestScoped`, `@SessionScoped` o `@ApplicationScoped`).

El nombre por defecto que se utilizará en las páginas JSF será el de la clase del bean gestionado, cambiando la primera letra mayúscula por una minúscula. Es posible definir un nombre distinto utilizando el atributo `name` de la anotación `@ManagedBean`.

```
package es.ua.jtech.jsf.ejemplo;

import java.io.Serializable;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.ViewScoped;

@ManagedBean
@ViewScoped
public class RichBean implements Serializable {
   private static final long serialVersionUID = -2403138958014741653L;
   private String name;

   public RichBean() {
      name = "John";
   }

   public String getName() {
       return name;
   }

   public void setName(String name) {
       this.name = name;
   }
}
```

1.6.6. Probando la aplicación

Ejecutaremos la aplicación para probar el código escrito (se encuentra en http://localhost:8080/jsf-expertojava/faces/sample.xhtml)

2. Ejercicios sesión 1 - Introducción a JSF

Dentro de las sesiones prácticas de JSF, vamos a realizar una pequeña aplicación de gestión de tareas. Y dentro de ésta, lo que realizaremos será:

- Login
- Registro
- Administración de tareas

Haz un *fork* del repositorio inicial `java_ua/jsf-expertojava`, desactivando la opción de heredar los permisos de acceso. Añade el usuario `java_ua` con permiso de lectura.

Clona el repositorio en tu disco duro y crea un nuevo espacio de trabajo en el repositorio recién clonado.

2.1. Creando nuestro proyecto con Maven (1 punto)

Dado que la práctica va a consistir en un único ejercicio, lo que vamos a hacer es crear un proyecto JSF + RichFaces, que ya contiene todos los elementos que vamos a necesitar a lo largo de las sesiones.

Crearemos el proyecto según las indicaciones establecidas en los apuntes de teoría

2.2. Mini-aplicación de ejemplo (1 punto)

Para probar que la importación ha sido correcta, crearemos la aplicación de ejemplo del final de los apuntes de teoría

2.3. Pantalla de login (1 punto)

Con lo que hemos visto hasta ahora, podemos crear la pantalla de login en nuestro gestor de tareas.

De momento, tendremos una estructura de dos páginas: una con el formulario de acceso, y otra de error que nos devolverá de nuevo a la página de login.

En primer lugar tendremos la plantilla general. Empezará siendo una simple plantilla (que colocaremos en la carpeta `templates/template.xhtml`) como la que sigue:

```
<!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:rich="http://richfaces.org/rich"
      xmlns:f="http://java.sun.com/jsf/core">
```

```
  <h:head>
      <title>jsfTasks - <ui:insert name="title">Application
Title</ui:insert></title>
      <meta http-equiv="content-type" content="text/xhtml; charset=UTF-8"
/>
  </h:head>

  <h:body>
    <ui:insert name="body">Default content</ui:insert>
  </h:body>
</html>
```

Vemos que lo único que contiene es la posibilidad de insertar el cuerpo que queramos, así como determinar parte del título de la página

El código de la pantalla de login (`login.xhtml`) será el siguiente:

```
      <!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:rich="http://richfaces.org/rich">

<body>
  <ui:composition template="/templates/template.xhtml">

  <ui:define name="title">Acceso</ui:define>

  <ui:define name="body">
    <rich:panel header="Acceso a la aplicación">
      <h:form prependId="false">
        <h:outputLabel
          value="Usuario:"
          for="nameInput" />
          <h:inputText
            value="hola" />
          <br />
          <h:outputLabel
            value="Contraseña: " />
          <h:inputSecret
            value="mundo" />
          <br />
          <h:commandButton
            action="errorPage?faces-redirect=true"
            value="Acceder" />
          <h:commandLink
            value="¿Olvidaste tu contraseña?"
            action="errorPage" />
        </h:form>
      </rich:panel>
    </ui:define>
  </ui:composition>
</body>
</html>
```

Por su parte, el código de la pantalla de error (`errorPage.xhtml`) tendrá la siguiente forma:

```
      <!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">

<body>
  <ui:composition template="/templates/template.xhtml">

    <ui:define name="title">Error</ui:define>

    <ui:define name="body">
      <h:form prependId="false">
        Se ha producido un error.
        <h:commandLink
          value="Volver"
          action="index?faces-redirect=true" />.
      </h:form>
     </ui:define>
   </ui:composition>
</body>
</html>
```

Modifica ahora el fichero web.xml para que la página de login se corresponda con la página de inicio

No te preocupes si no entiendes partes del código. Lo iremos viendo todo a lo largo de las sucesivas sesiones.

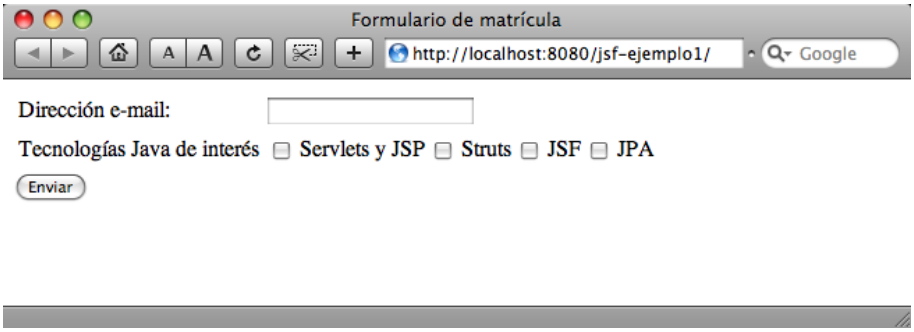
3. El MVC en JavaServer Faces

3.1. Modelo-Vista-Controlador

JSF utiliza el framework MVC (Modelo-Vista-Controlador) para gestionar las aplicaciones web. Vamos a verlo con cierto detalle.

La mejor forma de comprobar estas características de JSF es mediante un ejemplo concreto. Consideremos una versión muy simplificada de una aplicación web de matriculación de estudiantes a cursos. Tenemos una página en la que el estudiante debe escribir su correo electrónico y marcar los cursos de los que desea matricularse. Cuando se pulsa el botón para enviar los datos, la aplicación debe realizar una llamada a un método de negocio en el que se realiza la matrícula del estudiante.

La siguiente figura muestra el aspecto de esta página web.



Veamos cómo podemos implementar este sencillo ejemplo con JSF, separando las distintas responsabilidades de la aplicación según el modelo MVC.

3.1.1. Vista

La forma más común de definir la **vista** en JSF (2.0) es utilizando ficheros XHTML con etiquetas especiales que definen componentes JSF. Al igual que en JSP, estos componentes se convierten al final en código HTML (incluyendo JavaScript en las implementaciones más avanzadas de JSF) que se pasa al navegador para que lo muestre al usuario. El navegador es el responsable de gestionar la interacción del usuario.

Veamos el código JSF que genera el ejemplo visto anteriormente. El fichero llamado `selec-cursos.xhtml` define la vista de la página web. A continuación se muestra la parte de este fichero donde se define el árbol de componentes JSF.

Fichero **`selec-cursos.xhtml`**

```
<h:body>
  <h:form>
```

```
    <table>
       <tr>
         <td>Dirección e-mail:</td>
         <td>
           <h:inputText value="#{selecCursosBean.email}" />
         </td>
      </tr>
       <tr>
          <td>Tecnologías Java de interés</td>
          <td><h:selectManyCheckbox
             value="#{selecCursosBean.cursosId}">
             <f:selectItem itemValue="Struts" itemLabel="Struts" />
             <f:selectItem itemValue="JSF" itemLabel="JSF" />
             <f:selectItem itemValue="JSP" itemLabel="Servlets y JSP" />
             <f:selectItem itemValue="JPA" itemLabel="JPA" />
          </h:selectManyCheckbox></td>
       </tr>
    </table>
    <h:commandButton value="Enviar"
         action="#{selecCursosController.grabarDatosCursos}"/> />
  </h:form>
</h:body>
```

Los componentes JSF son un sencillo campo de texto (`h:inputText`) para el correo electrónico y una caja de selección de opción múltiple (`h:selectManyCheckbox`) con la que marcar los cursos seleccionados. Cada curso a seleccionar es a su vez un componente de tipo `f:selectItem`. Para lanzar la acción de matricular de la capa de negocio se utiliza el componente botón (`h:commandButton`). Todos los componentes se encuentran dentro de un `h:form` que se traducirá a un formulario HTML.

Los componentes tienen un conjunto de atributos con los que se especifican sus características. Por ejemplo, el componente `f:selectItem` utiliza el atributo `itemLabel` para definir el texto que aparece en la página y el atributo `itemValue` para indicar el valor que se enviará a la aplicación.

Un aspecto muy importante de JSF es la conexión de las vistas con la aplicación mediante los denominados *beans gestionados*. En nuestro ejemplo, la vista utiliza dos beans: `selecCursosBean` y `selecCursosController`. El primero se utiliza para guardar los datos introducidos por el usuario, y el segundo proporciona el método manejador al que se llamará cuando se pulse el botón de la página. El primer bean mantiene el modelo de la vista y el segundo su controlador..

La conexión entre las clases Java y las páginas JSF se realiza mediante el Lenguaje de Expresiones JSF (JSF EL), una versión avanzada del lenguaje de expresiones de JSP. Con este lenguaje, podemos definir conexiones (*bindings*) entre las propiedades de los beans y los valores de los componentes que se muestran o que introduce el usuario.

En el ejemplo anterior se define un *binding* entre el componente `h:selectManyCheckbox` y la propiedad `cursosId` del bean `selecCursosBean` mediante la expresión

```
<h:selectManyCheckbox
   value="#{selecCursosBean.cursosId}">
```

De esta forma, los valores de los datos seleccionados por el usuario se guardarán en esa

propiedad del bean y podremos utilizarlos en la acción `grabarDatosCursos` que se ejecuta cuando se pulsa el botón.

3.1.2. Modelo: beans gestionados

El modelo JSF se define mediante beans idénticos a los que se utilizan en JSP. Un bean es una clase con un conjunto de atributos (denominados *propiedades*) y métodos *getters* y *setters* que devuelven y actualizan sus valores. Las propiedades del bean se pueden leer y escribir desde las páginas JSF utilizando el lenguaje de expresiones EL.

Por ejemplo, en la anterior página `selec-cursos.xhtml` se utiliza el bean `selecCursosBean` para guardar los datos seleccionados por el usuario. La definición del bean es la siguiente:

Clase **`SelecCursosBean.java`**

```
public class SelecCursosBean {
  private String email;
  private String[] cursosId;

  public String getEmail() {
    return email;
  }

  public void setEmail(String email) {
    this.email = email;
  }

  public String[] getCursosId() {
    return cursosId;
  }

  public void setCursosId(String[] cursosId) {
    this.cursosId = cursosId;
  }
}
```

El bean define dos propiedades:

- `email`: un `String` que guardará el correo electrónico introducido por el usuario. Asociado a este atributo se definen los métodos `getEmail()` y `setEmail(String email)`.
- `cursosId`: un array de `Strings` que guardará la selección de cursos realizada por el usuario. También se definen en el bean los métodos *get* y *set* asociados a esta propiedad, que devuelven y toman como parámetro un array de cadenas.

Para escoger el nombre de la clase hemos seguido el convenio de utilizar el nombre de la página en la que se usa el bean y el sufijo `Bean`. De esta forma remarcamos que las instancias de esta clase van a ser beans gestionados que van a contener los datos mostrados y obtenidos en esa página.

En las expresiones JSF EL de la página en la que se usa el bean se puede acceder a sus propiedades utilizando el nombre de la propiedad. Si la expresión es sólo de lectura se

utilizará internamente el método *get* para recuperar el contenido del bean y mostrarlo en la página. Si la expresión es de lectura y escritura, se utilizará además el *set* para modificarlo con los datos introducidos por el usuario de la página.

Por ejemplo, en el siguiente fragmento de código se está mapeando la propiedad `cursoIds` del bean `selecCursosBean` con el valor del componente `h:selectManyCheckbox`. De esta forma, los valores de los cursos seleccionados por el usuario se guardarán en la propiedad `cursosIds` como un array de cadenas utilizando el método `setCursosId`. Después la aplicación podrá utilizar el método `getCursosId` para recuperar esos valores.

```
<h:selectManyCheckbox
   value="#{selecCursosBean.cursoIds}">
  <f:selectItem itemValue="JSP" itemLabel="Servlets y JSP" />
  <f:selectItem itemValue="Struts" itemLabel="Struts" />
  <f:selectItem itemValue="JSF" itemLabel="JSF" />
  <f:selectItem itemValue="JPA" itemLabel="JPA" />
</h:selectManyCheckbox>
```

¿Dónde se declaran los beans en una aplicación JSF? Hemos visto hasta ahora el uso de un bean en una página JSF y su definición como una clase Java. Nos falta explicar cómo se indica que el bean `selecCursosBean` es un objeto de la clase `selecCursosBean`. Históricamente, los beans de una aplicación se han declarado en su fichero de configuración `WEB-INF/faces-config.xml`.

En el siguiente fragmento de código podemos comprobar cómo se define un bean llamado `selecCursosBean` de la clase `jtech.jsf.presentacion.SelecCursosBean` con el ámbito de sesión. El ámbito determina cuándo se crea un bean nuevo y desde dónde se puede acceder a él. El ámbito de sesión indica que se debe crear un bean nuevo en cada nueva sesión HTTP y que va a ser accesible en todas las vistas JSF que compartan esa misma sesión.

Fichero **`faces-config.xml`**

```
...
<managed-bean>
   <managed-bean-name>selecCursosBean</managed-bean-name>
   <managed-bean-class>
      jtech.jsf.presentacion.SelecCursosBean
   </managed-bean-class>
   <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
...
```

Sin embargo, con la llegada de la versión 2 de JSF, podemos usar anotaciones para definir beans gestionados y como el ámbito de los mismos. Así, declararemos la clase `SelecCursosBean.java` como bean gestionado de la siguiente manera:

```
@ManagedBean
@SessionScope
public class SelecCursosBean {
   private String email;
   private String[] cursosId;
...
```

3.1.2.1. Ámbito de los beans gestionados

El ámbito de los beans determina su ciclo de vida: cuándo se crean y destruyen instancias del bean y cuándo se asocian a las páginas JSF. Es muy importante definir correctamente el ámbito de un bean, porque en el momento de su creación se realiza su incialización. JSF llama al método constructor de la clase, donde habremos colocado el código para inicializar sus valores, creando una instancia de la clase que pasará a ser gestionada por el framework.

En JSF se definen los siguientes ámbitos para los beans: petición, sesión, vista y aplicación. El ámbito de vista es un elemento nuevo de JSF 2.0. Los otros ámbitos son similares a los definidos con los JavaBeans de JSP.

- **Petición**: Se define con el valor `request` en la propiedad `managed-bean-scope` del `faces-config.xml` o con la anotación `@RequestScoped` en la clase. El bean se asocia a una petición HTTP. Cada nueva petición (cuando desde el navegador se abre una página por primera vez una página o se recarga) crea un nuevo bean y lo asocia con la página. Dada su corta vida, se recomienda usar este ámbito para el paso de mensajes (bien sea de error o de estatus), o para cualquier otro tipo de dato que no sea necesario propagar a lo largo de la aplicación
- **Sesión**: Se define con el valor `session` en el `faces-config.xml` o con la anotación `@SessionScoped` en la clase. Las sesiones se definen internamente con el API de Servlets. Una sesión está asociada con una visita desde una navegador. Cuando se visita la página por primera vez se incia la sesión. Cualquier página que se abra dentro del mismo navegador comparte la sesión. La sesión mantiene el estado de los elementos de nuestra aplicación a lo largo de las distintas peticiones. Se implementa utilizando cookies o reescritura de URLs, con el parámetro `jsessionid`. Una sesión no finaliza hasta que se invoca el método `invalidate` en el objeto `HttpSession`, o hasta que se produce un timeout.
- **Aplicación**: Se define con el valor `application` y con la anotación `@ApplicationScoped`. Los beans con este ámbito viven asociados a la aplicación. Definen *singletons* que se crean e inicializa sólo una vez, al comienzo de la aplicación. Se suelen utilizar para guardar características comunes compartidas y utilizadas por el resto de beans de la aplicación.
- **Vista** (JSF 2.0): Se define con el valor `view` en el `faces-config.xml` o con la anotación `@ViewScoped` en la clase. Un bean en este ámbito persistirá mientras se repinte la misma página (vista = página JSF), al navegar a otra página, el bean sale del ámbito. Es bastante útil para aplicaciones que usen Ajax en parte de sus páginas.
- **Custom** (`@CustomScoped`): Un ámbito al fin y al cabo no es más que un mapa que enlaza nombres y objetos. Lo que distingue un ámbito de otro es el tiempo de vida de ese mapa. Los tiempos de vida de los ámbitos estándar de JSF (sesión, aplicación, vista y petición) son gestionados por la implementación de JSF. En JSF 2.0 podemos crear ámbitos personalizados, que son mapas cuyo ciclo de vida gestionamos nosotros. Para incluirlo en ese mapa, usaremos la anotación `@CustomScoped("#{expr}")`, donde `#{expr}` indica el mapa. Nuestra aplicación será

la responsable de eliminar elementos de ese mapa.

- **Conversación** (`@ConversationScoped`) - provee de persistencia de datos hasta que se llega a un objetivo específico, sin necesidad de mantenerlo durante toda la sesión. Está ligado a una ventana o pestaña concreta del navegador. Así, una sesión puede mantener varias conversaciones en distintas páginas. Es una característica propia de CDI, no de JSF.

3.1.2.2. Inicialización de los beans

Existen dos formas de inicializar el estado de un bean. Una es hacerlo por código, incluyendo las sentencias de inicialización en su constructor. La otra es por configuración, utilizando el fichero `faces-config.xml`.

Supongamos que queremos incializar la propiedad `e-mail` del bean `selecCursosBean` al valor `Introduce tu e-mail`, para que aparezca este String cuando se carga la página por primera vez.

Para hacer la inicialización por código basta incluir la actualización de la propiedad en el constructor de la clase:

```
public class SelecCursosBean {
  private String email;
  private String[] cursosId;

  public SelecCursosBean() {
    email="Introduce tu e-mail";
  }
   ...
```

La otra forma de inicializar la propiedad es añadiendo al fichero `faces-config.xml` la etiqueta `managed-property` con el nombre de la propiedad que se quiere inicializar y el valor. Sería de esta forma:

```
<managed-bean>
   <managed-bean-name>selecCursosBean</managed-bean-name>
   <managed-bean-class>
     org.especialistajee.jsf.SelecCursosBean
   </managed-bean-class>
   <managed-bean-scope>request</managed-bean-scope>
   <managed-property>
      <property-name>email</property-name>
      <value>Introduce to e-mail</value>
   </managed-property>
</managed-bean>
```

Incluso existe una tercera forma con JSF 2.0: utilizando la anotación `@ManagedProperty` y el atributo `value`:

```
@ManagedBean
@RequestScoped
public class SelecCursosBean {
   @ManagedProperty(value="Introduce tu e-mail")
   private String email;
   private String[] cursosId;
   ...
```

Podemos poner como valor cualquier expresión EL válida. Por ejemplo, podríamos llamar a algún método de otro bean ya existente y guardar en la propiedad el resultado. Aquí una expresión muy sencilla:

```
<managed-bean>
   <managed-bean-name>selecCursosBean</managed-bean-name>
   <managed-bean-class>
     org.especialistajee.jsf.SelecCursosBean
   </managed-bean-class>
   <managed-bean-scope>request</managed-bean-scope>
   <managed-property>
      <property-name>email</property-name>
      <value>#{1+2+3+4}</value>
   </managed-property>
</managed-bean>
```

3.1.2.3. Relaciones entre beans

Es posible utilizar la inicialización de propiedades para crear relaciones entre distintos beans, almacenando un bean en una propiedad de otro. Esto es muy útil para implementar correctamente un modelo MVC. Veremos en el siguiente apartado que para seguir este enfoque necesitamos un bean haga el papel de *controller* y que defina las acciones a ejecutar en la aplicación. Estas acciones son métodos definidos en el bean. Los datos introducidos por el usuario se encuentran en otro bean (o incluso en más de un bean asociado a una página) ¿Cómo acceder desde el bean *controller* a los beans del modelo?.

Para solucionar este problema JSF provee tres soluciones.

Inyección de dependencias

En caso de disponer de un servidor de aplicaciones que tenga soporte para CDI, ésta es la opción recomendada. El objetivo de CDI, entre otros, unificar el modelo de componentes gestionados de JSF con el modelo de componentes de EJB, con lo que el desarrollo se simplifica considerablemente.

```
@Named @RequestScoped
public class EntradaBlogBean {
  @Inject private UsuarioBean usuario;

  public void setUsuario(UsuarioBean usuario){
    this.usuario = usuario;
  }

  public UsuarioBean getUsuario(){
    return usuario;
  }
}
```

Inyección de beans gestionados

Caso y declaración muy similares a la anterior. La única diferencia es que la inyección de dependencias de JSF no es tan potente como la de CDI. Es el modelo que utilizaremos en

los ejercicios, dado que la versión 7 de Apache Tomcat por defecto no incorpora soporte para CDI.

```
@ManagedBean @RequestScoped
public class EntradaBlogBean {
  @ManagedProperty(value="#{usuarioBean}")
  private UsuarioBean usuario;

  public void setUsuario(UsuarioBean usuario){
    this.usuario = usuario;
  }

  public UsuarioBean getUsuario(){
    return usuario;
  }
}
```

Configurar los beans gestionados con XML

JSF 2 mantiene la configuración previa de beans a través de un fichero XML. Es más farragosa, pero permite la configuración en tiempo de despliegue. Este fichero XML puede estar en:

- WEB-INF/faces-config.xml. Sitio tradicional por defecto.
- Cualquier fichero que finalice en .faces-config.cdml dentro del directorio META-INF de un jar. Muy útil para elaborar componentes reusables y distribuirlos en un jar
- Ficheros listados en el parámetro de inicialización `javax.faces.CONFIG_FILES` en el `web.xml`. Esto resulta muy útil para separar navegación de configuración de beans, etc

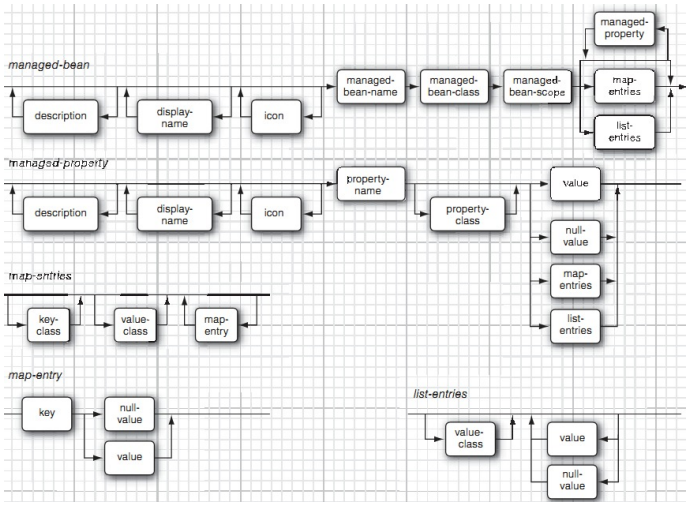
```
<web-app>
  <context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
<param-value>WEB-INF/navigation.xml,WEB-INF/managedbeans.xml</param-value>
  </context-param>
  ...
</web-app>
```

En El fichero XML, los beans se definen de la siguiente manera:

```
<faces-config>
  <managed-bean>
    <managed-bean-name>usuario</managed-bean-name>
    <managed-bean-class>
      org.especialistajee.beans.UsuarioBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <!-- (request, view, session, application, none) -->
    <managed-bean eager="true"> (opcional)
    <managed-property>
      <property-name>nombre</property-name>
```

```
      <value>Alejandro</value> <-- Inicializamos valores concretos
    </managed-property>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>entradaBean</managed-bean-name>
    <managed-bean-class>
      org.especialistajee.beans.EntradaBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
      <property-name>usuario</property-name>
      <value>#{usuarioBean}</value> <-- Inyectamos bean mediante
expresiones EL
    </managed-property>
    <managed-property>
      <property-name>titulo</property-name>
      <null-value /> <-- Valores nulos
    </managed-property>
    <managed-property>
      <property-name>autr</property-name>
      <value>#{usuarioBean}</value> <-- Inyectamos bean mediante
expresiones EL
    </managed-property>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>listBean</managed-bean-name>
    <managed-bean-class>java.util.ArrayList</managed-bean-class>
    <managed-bean-scope>none</managed-bean-scope>
    <list-entries>
      <value>Domingo Gallardo</value>
      <value>Otto Colomina</value>
      <value>Fran García</value>
      <value>Alejandro Such</value>
    </list-entries>
  </managed-bean>

  <managed-bean>
    <managed-bean-name>mapBean</managed-bean-name>
    <managed-bean-class>java.util.HashMap</managed-bean-class>
    <managed-bean-scope>none</managed-bean-scope>
    <map-entries>
      <map-entry>
        <key>JPA</key>
        <value>Domingo Gallardo</value>
      </map-entry>
      <map-entry>
        <key>Spring</key>
        <value>Otto Colomina</value>
      </map-entry>
      <map-entry>
        <key>Grails</key>
        <value>Fran García</value>
      </map-entry>
      <map-entry>
        <key>JSF</key>
        <value>Alejandro Such</value>
      </map-entry>
    </map-entries>
  </managed-bean>
</faces-config>
```

A modo de resumen, así declararemos un bean en el fichero xml:



En cualquiera de los tres casos vistos, hay que tener en cuenta que el ámbito de la propiedad no puede ser inferior al del bean contenedor.

Cuando nuestro bean tiene el ámbito...	...su propiedades pueden ser beans de los ámbitos
none	none
application	none, application
session	none, application, session
view	none, application, session, view
request	none, application, session, view, request

Anotaciones para controlar el ciclo de vida de los beans

Las anotaciones `@PostConstruct` y `@PreDestroy` sirven para especificar métodos que se invocan automáticamente nada más construir un bean, o justo antes de que éste salga del ámbito:

```
public class MiBean {
   @PostConstruct
   public void initialize() {
      // Código de inicialización
   }
   @PreDestroy
   public void shutdown() {
      // Código de finalización
   }
   // resto de métodos del bean
}
```

3.1.3. Navegación

Antes de ver el controlador, es interesante comprender mejor la navegación en JSF. La navegación se refiere al flujo de páginas a lo largo de nuestra aplicación. Veremos cómo navegar de una página a otra de distintas maneras.

3.1.3.1. Navegación estática

La navegación estática se refiere a aquella situación en que hacemos click en un botón o enlace, y el destino de esa acción va a ser siempre el mismo. Ésta se hace dándole a un determinado enlace una acción. Por ejemplo:

```
<h:commandButton label="Login" action="welcome"/>
```

En la navegación estática, y si no se provee un mapeo como veremos más adelante, la acción se transformará en un identificador de vista de la siguiente manera:

- Si el nombre no tiene una extensión, se le asigna la misma extensión que la vista actual.
- Si el nombre no empieza por /, se pone como prefijo el mismo que la vista actual.

Por ejemplo, la acción `welcome` en la vista `/index.xhtml` nos lleva al identificador de vista `/welcome.xhtml`, y en la vista `/user/index.xhtml` nos llevaría a `/user/welcome.xhtml`

3.1.3.2. Navegación dinámica

Ésta es muy común en muchas aplicaciones web, donde el flujo no depende del botón que se pulse, sino de los datos introducidos. Por ejemplo: hacer login lleva a dos páginas distintas en función de si el par usuario/password introducidos son correctos (la página de login erróneo, o la página de inicio de la parte privada).

Para implementar esta navegación dimámica, el botón de login debería apuntara un método:

```
<h:commandButton label="Login"
action="#{loginController.verificarUsuario}"/>
```

Aquí, se llamará al método verificarUsuario() del bean loginController (sea de la clase que sea), que puede tener cualquier tipo de retorno que pueda ser convertible a String.

```
String verificarUsuario() {
   if (...)
      return "success";
   else
      return "failure";
}
```

Este retorno "success"/"failure" será el que determine la siguiente vista.

Este mapeo cadena-vista se realiza en un fichero de configuración, permitiéndonos separar la presentación de la lógica de negocio. Porque, ¿para qué tengo que decir que vaya a la página /error.xhtml? ¿Y si luego cambio mi estructura?. Éstas salidas lógicas nos dan esta flexibilidad. La configuración se realiza mediante reglas de navegación en el fichero faces-config.xml. Un ejemplo de reglas de navegación sería:

```
<navigation-rule>
   <from-view-id>/index.xhtml</from-view-id>
   <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/welcome.xhtml</to-view-id>
   </navigation-case>
     <navigation-case>
      <from-outcome>failure</from-outcome>
      <to-view-id>/index.xhtml</to-view-id>
   </navigation-case>
</navigation-rule>
```

Esta regla determina que la salida "success" nos llevará de la vista /index.xhtml a la vista /welcome.xhtml. Si es "failure", llevará de nuevo al /index.xhtml

Si especificamos una regla sin incluir un from-view-id, ésta se aplicará a todas las páginas de nuestra aplicación.

También podemos añadir wildcards en el elemento from-view-id para que se aplique a ciertas vistas nada más. Sólo se permite un único *, que debe estar al final de la cadena

```
<navigation-rule>
   <from-view-id>/secure/*</from-view-id>
   <navigation-case>
      . . .
   </navigation-case>
</navigation-rule>
```

Otros casos de navegación dinámica

Según acción

Otro caso de navegación que podemos incluir es `from-action`, que combinado con el tag `from-outcome` permite tener cadenas iguales que desemboquen en diferentes destinos

```
<navigation-case>
   <from-action>#{loginController.cerrarSesion}</from-action>
   <from-outcome>success</from-outcome>
   <to-view-id>/index.xhtml</to-view-id>
</navigation-case>
<navigation-case>
   <from-action>#{entradaController.anyadirEntrada}</from-action>
   <from-outcome>success</from-outcome>
   <to-view-id>/verEntrada.xhtml</to-view-id>
</navigation-case>
```

Casos de navegación condicionales

Estos casos de navegación son exclusivos de JSF2, y nos permiten hacer comparaciones simples haciendo uso del lenguaje de expresiones para determinar la siguiente vista.

```
<navigation-rule>
  <from-view-id>login.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <if>#{user.powerUser}</if>
      <to-view-id>/index_power.xhtlm</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <if>#{user.vipUser}</if>
      <to-view-id>/index_vip.xhtlm</to-view-id>
    </navigation-case>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/index_user.xhtlm</to-view-id>
    </navigation-case>
</navigation-rule>
```

Obviamente, sería equivalente al siguiente código java:

```
if(user.isPowerUser()){
  return "powerUser";
} else if(user.isVipUser()){
  return "vipUser";
}

return "user"
```

Sin embargo, el empleo de reglas condicionales en este caso nos permite dotar al controlador de login de una única responsabilidad: validar al usuario.

Dynamic Target View IDs

El elemento to-view-id puede ser una expresión EL, que se evaluará en tiempo de ejecución.

```
<navigation-rule>
  <from-view-id>/main.xhtml</from-view-id>
  <navigation-case>
    <to-view-id>#{quizBean.nextViewID}</to-view-id>
  </navigation-case>
</navigation-rule>
```

Esta regla es exclusiva de JSF2

3.1.4. Controlador

Llegamos al último pilar del patrón MVC aplicado a JSF. Hemos visto cómo en JSF se define la vista de una aplicación y cómo se obtienen y se guardan los datos del modelo. Nos falta el controlador. ¿Cómo se define el código que se debe ejecutar cuándo el usuario realiza alguna acción sobre algún componente?

Debemos diferenciar dos tipos de acciones, las *acciones del componente* y las *acciones de*

la aplicación. En el primer tipo de acciones es el propio componente el que contiene el código (HTML o JavaScript) que le permite reaccionar a la interacción del usuario. Es el caso, por ejemplo, de un menú que se despliega o un calendario que se abre. En este caso no hay ninguna petición al controlador de la aplicación para obtener datos o modificar algún elemento, sino que toda la interacción la maneja el propio componente. Con RichFaces y JSF 2.0 es posible utilizar eventos JavaScript para configurar este comportamiento.

Las acciones de la aplicación son las que determinan las funcionalidades de negocio de la aplicación. Se trata de código que queremos que se ejecute en el servidor cuando el usuario pulsa un determinado botón o pincha en un determinado enlace. Este código realizará llamadas a la capa de negocio de la aplicación y determinará la siguiente vista a mostrar o modificará la vista actual.

Como hemos visto en la navegación dinámica, las acciones se definen en beans gestionados de la página JSF. Son métodos del bean que se ligan al elemento `action` del componente que vaya a lanzar esa acción.

Por ejemplo, en la página `selec-curso` se define llama a la acción `grabarDatosCursos` del bean `selecCursosController` asociándola a un `<h:commandButton>`:

```
<h:commandButton value="Enviar"
   action="#{selecCursosController.grabarDatosCursos}"/>
```

El método `grabarDatosCursos` se ejecuta, realizando la lógica de negocio, y devuelve una cadena que determina en el fichero `faces-config.xml` la siguiente vista a mostrar.

3.1.4.1. Llamadas a la capa de negocio

En el controlador se guarda la relación con el bean en el que se recogen los datos de la forma que hemos visto antes y se define el método que realiza la llamada a la capa de negocio (el método `grabarDatosCursos`):

Fichero **`jtech.jsf.controlador.SelecCursosController.java`**

```
@ManagedBean
@SessionScoped
public class SelecCursosController {
  @ManagedProperty(value="#{selecCursosBean}")
  private SelecCursosBean datosCursos;

  public SelecCursosBean getDatosCursos() {
    return datosCursos;
  }

  public void setDatosCursos(SelecCursosBean datosCursos) {
    this.datosCursos = datosCursos;
  }

  public String grabarDatosCursos() {
    EstudianteBO estudianteBO = new EstudianteBO();
    String email = datosCursos.getEmail();
    String[] cursosId = datosCursos.getCursosId();
    estudianteBO.grabarAsignaturas(email, cursosId);
```

```
    return "OK";
  }
}
```

Vemos en el método `grabarDatosCursos()` la forma típica de proceder del controlador. Se obtiene el objeto de negocio con el método de negocio al que se quiere llamar y se realiza la llamada, pasándole los parámetros introducidos por el usuario. En nuestro caso, realizamos una llamada a un método `grabarAsignaturas` que realiza la matrícula del estudiante a esas asignaturas.

El email y los cursos seleccionados han sido introducidos por el usuario y, como hemos visto, se encuentran en el bean `selecCursosBean`.

3.1.4.2. Determinando la nueva vista de la aplicación

Además de lanzar la lógica de negocio requerida por la selección del usuario, el bean controlador debe definir también qué sucede con la interfaz una vez realizada la acción.

Recordemos que la definición de la acción en la página JSF es:

```
<h:commandButton value="Enviar"
   action="#{selecCursosController.grabarDatosCursos}"/>
```

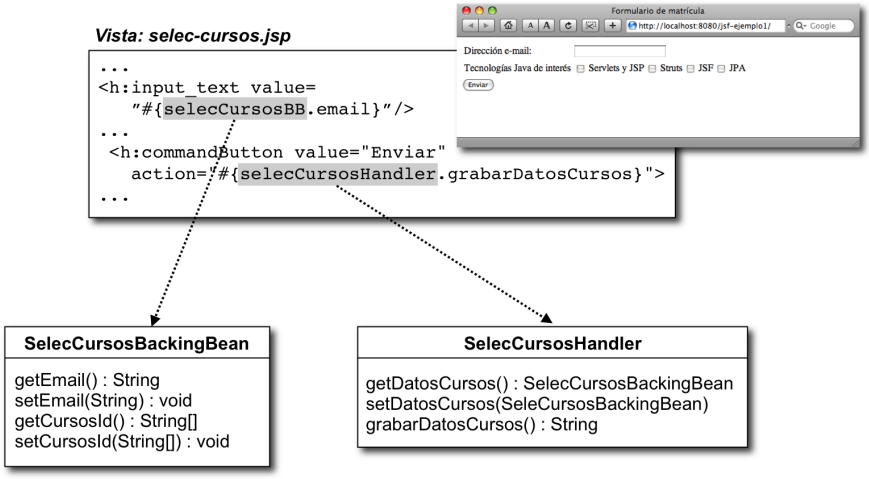
JSF obliga a que todas las acciones devuelvan una cadena. Esa cadena es el valor que termina guardándose en el atributo `action` y será evaluado en las reglas de navegación.

3.1.5. Resumen de los elementos de JSF

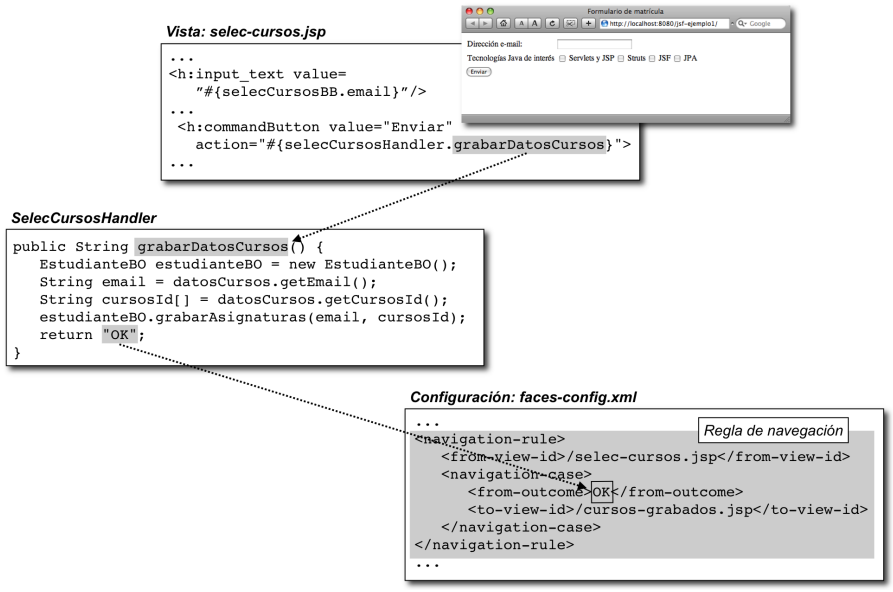
Veamos un resumen rápido de cómo se relacionan los distintos elementos de JSF.

En primer lugar, la **vista** se define mediante páginas con componentes JSF que utilizan *beans gestionados* para almacenar los datos. Los beans se declaran en el fichero de configuración `faces-config.xml`. La siguiente figura muestra las relaciones entre ambos en nuestro ejemplo. Para acceder a los datos, JSF utiliza un *lenguaje de expresiones* (JSF EL) similar al de JSP. El lenguaje de expresiones se puede utilizar también en el fichero `faces-config.xml` para inicializar los valores de las propiedades de los beans. En este caso, se utiliza para poder acceder a un bean desde otro.

En segundo lugar, el **modelo** se define mediante los beans. Son beans Java normales con propiedades y métodos *getters* y *setters*.



Por último, el **controlador** se define mediante métodos de los beans ligados a acciones de la vista. La acción a ejecutar se define en el código del método y la vista resultante depende de la cadena devuelta y del fichero de configuración `faces-config.xml`.



3.2. Expresiones EL

Las expresiones JSF EL son muy similares a las vistas en JSP. Son expresiones evaluables utilizadas en los atributos de las etiquetas JSF, normalmente el atributo `value`. Su sintaxis es `#{...}`. Por ejemplo, la siguiente expresión se utiliza en el atributo `value` de un `<h:outputText>` para definir un valor que se mostrará en la página HTML:

```
<h:outputText value="El resultado de 1+2+3 es #{1+2+3}"
```

La diferencia fundamental con JSP es que las expresiones JSF se incluyen tal cual en los componentes JSF. Cuando JSF obtiene el árbol de componentes asociado a la petición, las expresiones EL no se evalúan, sino que se incluyen en los componentes. De hecho, JSF convierte el texto de la expresión EL en un objeto de tipo `javax.el.ValueExpression` que se asocia a la propiedad correspondiente del componente. En el caso anterior, la expresión `#{1+2+3}` se convertiría en un objeto de tipo `ValueExpression` y se asociaría al atributo `value` del `outputText`.

Los métodos del API JSF que se utilizan para definir y obtener la expresión EL asociada a un atributo de un componente se definen precisamente en la clase `UIComponent`. Esta es una clase abstracta a partir de la que se construyen todos los componentes específicos. Son los siguientes métodos:

```
ValueExpression getValueExpression(String nombrePropiedad)
void setValueExpression(String nombrePropiedad, ValueExpression
```

```
expresionEL)
```

La evaluación de las expresiones se realiza en la fase *Apply request values* cuando JSF llama al método `decode` del componente.

El uso más frecuente de las expresiones EL es el *binding* de una propiedad de un bean a una propiedad del componente. Por ejemplo:

```
<h:outputText
   value="El total del pedido es: #{pedido.importe}"
   style="#{pedido.cssStyle}"
```

En esta expresión se está ligando la propiedad `importe` del bean `pedido` con la propiedad `value` del `outputText`. Además, se está definiendo el estilo CSS del texto de salida de forma dinámica, ligando la propiedad `style` del componente con la propiedad `cssStyle` del bean.

Cuando ligamos una propiedad de un bean a un componente, la expresión EL puede utilizarse para obtener el valor del bean y asignarlo al componente o, al revés, para obtener el valor del componente y asignarlo al bean. En el primer caso se dice que la expresión tiene una semántica *getValue* y en el segundo caso una semántica *setValue*.

Ejemplos de expresiones JSF EL correctas:

```
#{foo.bar}
#{foo[bar]}
#{foo["bar"]}
#{foo[3]}
#{foo[3].bar}
#{foo.bar[3]}
#{customer.status == 'VIP'}
#{(page1.city.farenheitTemp - 32) * 5 / 9}
```

En el caso de las expresiones con semántica *setValue*, la sintaxis está restringida a expresiones del tipo:

```
#{expr-a.value-b}
#{expr-a[value-b]]
#{value-b}
```

Siendo `expr-a` una expresión EL que se evalúa a un objeto de tipo `Map`, `List` o un JavaBean y `value-b` un identificador.

En las expresiones EL es posible utilizar un conjunto de identificadores que denotan ciertos objetos implícitos que podemos utilizar:

- `requestScope`, `sessionScope`, `applicationScope`: permite acceder a las variables definidas en el ámbito de la petición, de la sesión y de la aplicación Estas variables se pueden actualizar desde código en los beans utilizando la clase `FacesContext`.
- `param`: para acceder a los valores de los parámetros de la petición.
- `paramValues`: para acceder a los arrays de valores de los parámetros de la petición.
- `header`: para acceder a los valores de las cabeceras de la petición.
- `headerValues`:para acceder a los arrays de valores de los parámetros de la petición.

- `cookie`: para acceder a los valores almacenados en las *cookies* en forma de objetos `javax.servlet.http.Cookie`
- `initParam`: para acceder a los valores de inicialización de la aplicación.
- `facesContext`: para acceder al objeto `javax.faces.context.FacesContext` asociado a la aplicación actual.
- `view`: para acceder al objeto `javax.faces.component.UIViewRoot` asociado a la vista actual.

Veamos algunos ejemplos de utilización de estas variables.

```
#{view.children[0].children[0].valid}
```

Aquí se accede a la propiedad `valid` del primer hijo, del primer hijo de la raíz del árbol de componentes.

```
FacesContext.getCurrentInstance().getExternalContext()
   .getSessionMap().put("variable Name", value);
```

Este código se debe ejecutar en el bean (en un evento o una acción) para actualizar una determinada variable de la sesión JSF.

3.3. Componentes estándar de JSF

En JSF se definen un número de componentes estándar que implementan las interfaces definidas en el punto anterior. Cada clase está asociada normalmente a una etiqueta JSP y se renderiza en código HTML.

Hay que notar que en JSF los componentes se definen en base a su función, no a su aspecto. El aspecto se modifica mediante el render asociado al componente. Así, por ejemplo, un campo de entrada de texto y un campo de entrada de contraseñas se representan por el mismo tipo de componente JSF pero tienen distintos Renders asociados.

Normalmente, existe una relación uno a uno entre componentes JSF y etiquetas.

Referencias:

- JavaDoc del API de JSF 2.0: http://java.sun.com/j2ee/javaserverfaces/1.2/docs/api
- Etiquetas de JSF: http://java.sun.com/j2ee/javaserverfaces/1.2/docs/tlddocs

3.3.1. Un ejemplo: el componente <h:dataTable>

El componente `<h:dataTable>` permite generar una tabla HTML a partir de una lista o un array de objetos Java.

La forma más sencilla de utilizarlo es la siguiente:

```
<h:dataTable value="#{selecCursosBB.cursos}" var="curso">
   <h:column>
      <h:outputText value="#{curso.nombre}" />
```

```
   </h:column>
   <h:column>
      <h:outputText value="#{curso.profesor}"/>
   </h:column>
   <h:column>
      <h:outputText value="#{curso.creditos}"/>
   </h:column>
</h:dataTable>
```

La propiedad `cursos` contiene una lista de cursos que se muestran en la tabla. La variable `curso` definida en el atributo `var` toma el valor de cada uno de los cursos de la lista y se utiliza para construir las filas de la tabla. La etiqueta `<h:column>` define cada una de las columnas de la tabla, y en ella se utiliza la variable `curso` para acceder a las distintas propiedades que queremos mostrar en la tabla.

Supongamos que la lista `cursos` ha sido inicializada así:

```
cursos.add(new Curso("JSP","Miguel Ángel Lozano",2));
cursos.add(new Curso("JSF", "Domingo Gallardo", 1));
cursos.add(new Curso("Struts", "Otto Colomina", 1));
```

La tabla resultante será:

```
   JSP     Miguel Ángel Lozano  2
   JSF     Domingo Gallardo     1
   Struts  Otto Colomina        1
```

Para definir una cabecera en la tabla hay que utilizar la etiqueta `<f:facet name="header">` en la columna, y generar el contenido de la cabecera con un `<h:outputText/>`:

```
<h:dataTable value="#{selecCursosBB.cursos}" var="curso">
   <h:column>
      <f:facet name="header">
         <h:outputText value="Curso"/>
      </f:facet>
      <h:outputText value="#{curso.nombre}" />
   </h:column>
   <h:column>
      <f:facet name="header">
         <h:outputText value="Profesor"/>
      </f:facet>
      <h:outputText value="#{curso.profesor}"/>
   </h:column>
   <h:column>
      <f:facet name="header">
         <h:outputText value="Créditos"/>
      </f:facet>
      <h:outputText value="#{curso.creditos}"/>
   </h:column>
</h:dataTable>
```

La tabla resultante será:

```
   Curso   Profesor             Créditos
   JSP     Miguel Ángel Lozano  2
   JSF     Domingo Gallardo     1
   Struts  Otto Colomina        1
```

En la etiqueta también se definen atributos para generar clases CSS que permiten definir el aspecto de las tablas, lo que da mucha versatilidad a la presentación. También es posible definir distintos tipos de elementos dentro de la tabla; no sólo texto, sino también otros componentes como botones, campos de texto o menús. De esta forma es posible definir formularios complejos.

3.3.2. Otro ejemplo: el componente ui:repeat

Una de las novedades de JSF 2 es el uso del tag `ui:repeat`, para ser usado en lugar de `h:dataTable`. El funcionamiento es similar en el sentido de que ambos iteran sobre un conjunto de datos. La diferencia radica en que `ui:repeat` no genera una estructura de tabla, sino que tenemos que definirla nosotros:

```
<table>
   <ui:repeat value="#{tableData.names}" var="name">
   <tr>
      <td>#{name.last},</td>
      <td>#{name.first}</td>
   </tr>
   </ui:repeat>
</table>
```

Pese a que puede ser un poco más tedioso ya que tenemos que declarar nosotros las etiquetas para darle un aspecto tabular, este tag nos permite que mostremos la información en la forma que nosotros queramos con DIVs, listas o el aspecto que más nos interese.

Además, el tag `ui:repeat` expone algunos atributos que pueden ser muy interesantes si quisiéramos recorrer un subconjunto de la colección:

- `offset` es el índice por el que empieza la iteración (valor por defecto: 0)
- `step` es la diferencia entre sucesivos valores de índice (valor por defecto: 1)
- `size` es el número de iteraciones (valor por defecto: (tamaño de la colección - `offset`) / `step`)

Así, si quisiéramos mostrar los elementos 10, 12, 14, 16, 18 de una colección, usaríamos:

```
<ui:repeat ... offset="10" step="2" size="5">
```

El atributo `varStatus` determina una variable con información del estado de la iteración. La variable declarada tendrá las siguientes propiedades:

- De tipo `Boolean`: `even, odd, first, last`, muy útiles para determinar estilos.
- De tipo `Integer`: `index, begin, step, end`, que dan el índice de la iteración actual, el offset inicial, step, size y offset final. Fijáos que `begin = offset` y `end = offset + step × size`, siendo `offset` y `size` los valores de los atributos del tag `ui:repeat`

La propiedad `index` puede usarse para numerar filas:

```
<table>
```

```
   <ui:repeat value="#{tableData.names}" var="name" varStatus="status">
   <tr>
      <td>#{status.index + 1}</td>
      <td>#{name.last},</td>
      <td>#{name.first}</td>
   </tr>
   </ui:repeat>
</table>
```

3.3.3. Componentes HTML <h:>

Veamos una rápida descripción de todas las posibles etiquetas con el prefijo `<h:>`

Etiqueta	Descripción
`<h:column>`	Se utiliza dentro de una etiqueta `<h:dataTable>` para representar una columna de datos tabulares. Podemos añadirle una etiqueta `<f:facet name="header">` o `<f:facet name="footer">`.

Ejemplo:

```
<h:dataTable value="#{reportController.currentReports}" var="report">
  <h:column rendered="#{reportController.showDate}">
    <f:facet name="header">
      <h:outputText value="Date" />
    </f:facet>
    <h:outputText value="#{report.date}" />
  </h:column>
  ...
</h:dataTable>
```

Etiqueta	Descripción
`<h:commandButton>`	Representa un comando que se renderiza como un botón HTML de tipo entrada. Cuando el usuario cliquea el botón, se envía el formulario al que pertenece y se lanza un evento `ActionEvent`.

Ejemplo:

```
<h:form>
  <h:commandButton value="Save" action="#{formController.save}" />
</h:form>
```

Etiqueta	Descripción
`<h:commandLink>`	Representa un comando que se renderiza como un enlace. Cuando el usuario pincha en el enlace, se ejecuta un código javascript que envía el formulario al que pertenece y se lanza un evento `ActionEvent`.

Ejemplo:

```
<h:form>
  <h:commandLink action="#{formController.save}">
    <hLoutputText value="Save" />
  </h:commandLink>
```

```
</h:form>
```

Etiqueta	Descripción
<h:dataTable>	Se renderiza en un elemento HTML <table>. Los elementos hijo <h:column< son los responsables de renderizar las columnas de la tabla. El atributo value debe ser un array de objetos y se define una variable que hace de iterador sobre ese array. Se puede indicar el primer objeto a mostrar y el número de filas con los atributos first="first" y rows="rows". Los componentes de la tabla pueden declarse con la faceta header y footer.

Ejemplo:

```
<h:dataTable value="#{reportController.currentReports}" var="report">
  <f:facet name="header">
    <h:outputText value="Expense Reports" />
  </f:facet>
  <h:column rendered="#{reportController.showDate}">
    <f:facet name="header">
      <h:outputText value="Date" />
    </f:facet>
    <h:outputText value="#{report.date}" />
  </h:column>
  ...
</h:dataTable>
```

Etiqueta	Descripción
<h:form>	Se renderiza como un elemento <form> con un atributo de acción definido por una URL que identifica la vista contenida en el formulario. Cuando se envía el formulario, sólo se procesan los componentes hijos del formulario enviado.

Ejemplo:

```
<h:form>
  <h:panelGrid columns="2">
    <h:outputText value="First name:" />
    <h:inputText value="#{user.firstName}" />
    <h:outputText value="Last name:" />
    <h:inputText value="#{user.lastName}" />
  </h:panelGrid>
</h:form>
```

Etiqueta	Descripción
<h:graphicImage>	Se renderiza como un elemento <img> con un atributo src que toma como valor el valor del atributo value de la etiqueta.

Ejemplo:

```
<h:graphicImage value="/images/folder-open.gif" />
```

Etiqueta	Descripción
<h:inputHidden>	Se renderiza como un elemento <input> con un atributo type definido como hidden.

Ejemplo:

```
<h:form>
  <h:inputHidden value="#{user.type}" />
</h:form>
```

Etiqueta	Descripción
<h:inputSecret>	Se renderiza como un elemento <input> con un atributo type definido como password.

Ejemplo:

```
<h:form>
  <h:inputSecret value="#{user.password}" />
</h:form>
```

Etiqueta	Descripción
<h:inputText>	Se renderiza como un elemento <input> con un atributo type definido como text.

Ejemplo:

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

Etiqueta	Descripción
<h:inputTextarea>	Se renderiza como un elemento <textarea>.

Ejemplo:

```
<h:form>
  <h:inputTextarea value="#{user.bio}" />
</h:form>
```

Etiqueta	Descripción
<h:message>	Este elemento obtiene el primer mensaje encolado para el componente identificado por el atributo for.

Ejemplo:

```
<h:form>
  <h:inputText id="firstName" value="#{user.firstName}" />
  <h:message for="firstName" errorStyle="color: red" />
</h:form>
```

Etiqueta	Descripción
<h:messages>	Este elemento obtiene todos los mensajes encolados.

Ejemplo:

```
<h:messages/>
<h:form>
  <h:inputText id="firstName" value="#{user.firstName}" />
  <h:message for="firstName" errorStyle="color: red" />
</h:form>
```

Etiqueta	Descripción
<h:outputFormat>	Define un mensaje parametrizado que será rellenado por los elementos definidos en parámetros `<f:param>`

Ejemplo:

```
<f:loadBundle basename="messages" var="msgs" />
<h:outputFormat value="#{msgs.sunRiseAndSetText}">
  <f:param value="#{city.sunRiseTime}" />
  <f:param value="#{city.sunSetTime}" />
</h:outputFormat>
```

Etiqueta	Descripción
<h:outputLabel>	Define un elemento HTML `<label>`.

Ejemplo:

```
<h:inputText id="firstName" value="#{user.firstName}" />
<h:outputLabel for="firstName" />
```

Etiqueta	Descripción
<h:outputLink>	Se renderiza como un elemento `<a>` con un atributo `href` definido como el valor del atributo `value`.

Ejemplo:

```
<h:outputLink value="../../logout.jsp" />
```

Etiqueta	Descripción
<h:outputText>	Se renderiza como texto.

Ejemplo:

```
<h:outputText value="#{user.name}" />
```

Etiqueta	Descripción
<h:panelGrid>	Se renderiza como una tabla de HTML, con el número de columnas definido por el atributo `columns`. Los componentes del

	panel pueden tener las facetas `header` y `footer`.

Ejemplo:

```
<h:form>
  <h:panelGrid columns="2">
    <h:outputText value="First name:" />
    <h:inputText value="#{user.firstName}" />
    <h:outputText value="Last name:" />
    <h:inputText value="#{user.lastName}" />
  </h:panelGrid>
</h:form>
```

Etiqueta	Descripción
`<h:panelGroup>`	El componente actúa como un contenedor de otros componentes en situaciones en las que sólo se permite que exista un componente, por ejemplo cuando un grupo de componentes se usa como una faceta dentro de un `panelGroup`. Se renderizará como un elemento `<span>`. Si le ponemos el atributo `layout="block"`, se renderizará como un `>div<`

Ejemplo:

```
<h:form>
  <h:panelGrid columns="2">
    <f:facet name="header">
      <h:panelGroup>
        <h:outputText value="Sales stats for " />
        </h:outputText value="#{sales.region}" style="font-weight: bold" />
      </h:panelGroup>
    </f:facet>
    <h:outputText value="January" />
    <h:inputText value="#{sales.jan}" />
    <h:outputText value="February" />
    <h:inputText value="#{sales.feb}" />
    ...
  </h:panelGrid>
</h:form>
```

Etiqueta	Descripción
`<h:selectBooleanCheckbox>`	Se renderiza como un elemento `<input>` con un atributo `type` definido como `checkbox`.

Ejemplo:

```
<h:form>
  <h:selectBooleanCheckbox value="#{user.vip}" />
</h:form>
```

Etiqueta	Descripción
`<h:selectManyCheckbox>`	Se renderiza como un elemento HTML `<table>` con un elemento `input` por cada uno de sus hijos, componentes de tipo `<f:selectItem>` y `<f:selectItems>`.

Ejemplo:

```
<h:form>
  <h:selectManyCheckbox value="#{user.projects}">
    <f:selectItems value="#{allProjects}" />
  </h:selectManyCheckbox>
</h:form>
```

Etiqueta	Descripción
<h:selectManyListbox>	Se renderiza como un elemento <select>. Las opciones se representan por los componentes hijos de tipo <f:selectItem> y <f:selectItems>.

Ejemplo:

```
<h:form>
  <h:selectManyListbox value="#{user.projects}">
    <f:selectItems value="#{allProjects}" />
  </h:selectManyListbox>
</h:form>
```

Etiqueta	Descripción
<h:selectManyMenu>	Se renderiza como un elemento <select>. Las opciones se representan por los componentes hijos de tipo <f:selectItem> y <f:selectItems>.

Ejemplo:

```
<h:form>
  <h:selectManyMenu value="#{user.projects}">
    <f:selectItems value="#{allProjects}" />
  </h:selectManyMenu>
</h:form>
```

Etiqueta	Descripción
<h:selectOneListbox>	Se renderiza como un elemento <select>. Las opciones se representan por los componentes hijos de tipo <f:selectItem> y <f:selectItems>.

Ejemplo:

```
<h:form>
  <h:selectOneListbox value="#{user.country}">
    <f:selectItems value="#{allCountries}" />
  </h:selectOneListbox>
</h:form>
```

Etiqueta	Descripción
<h:selectOneMenu>	Se renderiza como un elemento <select>. Las opciones se representan por los componentes hijos de tipo <f:selectItem> y <f:selectItems>.

Ejemplo:

```
<h:form>
  <h:selectOneMenu value="#{user.country}">
    <f:selectItems value="#{allCountries}" />
  </h:selectOneMenu>
</h:form>
```

Etiqueta	Descripción
<h:selectOneRadio>	Se renderiza como un elemento <input> con un atributo type definido como radio. Las opciones se representan por los componentes hijos de tipo <f:selectItem> y <f:selectItems>.

Ejemplo:

```
<h:form>
  <h:selectOneRadio value="#{user.country}">
    <f:selectItems value="#{allCountries}" />
  </h:selectOneRadio>
</h:form>
```

3.3.4. Etiquetas core <f:>

Las otras etiquetas del núcleo de JSF son las etiquetas *core custom actions* con el prefijo <f:>. Definen acciones asociadas a los componentes o las páginas JSF. Se procesan en el servidor, una vez creado el árbol de componentes y modifican alguna característica del mismo. Por ejemplo, utilizando estas etiquetas es posible añadir elementos hijos, conversores o validadores a un componente.

En el ejemplo anterior hemos utilizado la etiqueta <f:selectItem> para añadir elementos hijos al componente <h:selectManyCheckbox>. Cada hijo define una etiqueta y un valor. Las etiquetas se muestran en pantalla y el valor es la cadena que se guarda en el array cursosId del bean gestionado por el componente selecCursosBean.

```
<h:selectManyCheckbox
   value="#{selecCursosBean.cursoIds}">
   <f:selectItem itemValue="JSP" itemLabel="Servlets y JSP" />
   <f:selectItem itemValue="Struts" itemLabel="Struts" />
   <f:selectItem itemValue="JSF" itemLabel="JSF" />
   <f:selectItem itemValue="JPA" itemLabel="JPA" />
</h:selectManyCheckbox>
```

Otra etiqueta muy útil es <f:setPropertyActionListener>. Junto con las etiquetas <h:commandLink> y <h:commandButton> permite definir alguna propiedad del bean que ejecutar una acción antes de que la acción se lance. De esta forma podemos simular un paso de parámetros a la acción.

Por ejemplo (tomado del blog de BalusC), si queremos definir las propiedades propertyName1 y propertyName2 en el bean antes de que se llame a la acción action podemos hacerlo de cualquiera de estas formas::

```
<h:form>
```

```
   <h:commandLink value="Click here" action="#{myBean.action}">
      <f:setPropertyActionListener target="#{myBean.propertyName1}"
                                   value="propertyValue1" />
      <f:setPropertyActionListener target="#{myBean.propertyName2}"
                                   value="propertyValue2" />
   </h:commandLink>

   <h:commandButton value="Press here" action="#{myBean.action}">
      <f:setPropertyActionListener target="#{myBean.propertyName1}"
                                   value="propertyValue1" />
      <f:setPropertyActionListener target="#{myBean.propertyName2}"
                                   value="propertyValue2" />
   </h:commandButton>
</h:form>
```

En el bean debemos definir las propiedades con al menos sus setters:

```
public class MyBean {

   private String propertyName1;
   private String propertyName2;

   // Actions

   public void action() {
      System.out.println("propertyName1: " + propertyName1);
      System.out.println("propertyName2: " + propertyName2);
   }

   // Setters

   public void setPropertyName1(String propertyName1) {
      this.propertyName1 = propertyName1;
   }

   public void setPropertyName2(String propertyName2) {
      this.propertyName2 = propertyName2;
   }
}
```

A continuación vemos una rápida descripción de otras etiquetas de la librería *Core custom actions* de JSF. Al igual que en las etiquetas HTML se incluyen ejemplos de su utilización.

Etiqueta	Descripción
`<f:actionListener>`	Crea una instancia de la clase definida en el atributo `type` y la añade al componente padre de tipo `action` para manejar el evento relacionado con el disparo de la acción.

Ejemplo:

```
<h:form>
  <h:commandButton value="Save">
    <f:actionListener type="com.mycompany.SaveListener" />
  </h:commandButton>
</h:form>
```

Etiqueta	Descripción
`<f:attribute>`	Define un atributo genérico para el componente padre.

Ejemplo:

```
<h:form>
  <h:inputText id="from" value="#{filter.from}" />
  <h:inputText value="#{filter.to}">
    <f:validator validatorId="com.mycompany.laterThanValidator" />
    <f:attribute name="compareToComp" value="from" />
  </h:inputText>
</h:form>
```

Etiqueta	Descripción
<f:convertDateTime>	Crea una instancia de un conversor de tipo `DateTime` con los parámetros proporcionados y lo registra en el componente padre.

Ejemplo:

```
<h:form>
  <h:inputText value="#{user.birthDate}">
    <f:convertDateTime dateStyle="short" />
  </h:inputText>
</h:form>
```

Etiqueta	Descripción
<f:convertNumber>	Crea una instancia de un conversor de tipo `Number` y lo registra en el componente padre.

Ejemplo:

```
<h:form>
  <h:inputText value="#{user.salary}">
    <f:convertNumber integerOnly="true" />
  </h:inputText>
</h:form>
```

Etiqueta	Descripción
<f:converter>	Crea una instancia de un conversor definido por el usuario y registrado con un identificador determinado.

Ejemplo:

```
<h:form>
  <h:inputText value="#{user.ssn}">
    <f:converter converterId="ssnConverter" />
  </h:inputText>
</h:form>
```

Etiqueta	Descripción
<f:facet>	Añade un componente con una faceta determinada al componente padre. Para que más de un componente pueda compartir la misma faceta, se deben agrupar con un elemento `<h:panelGroup>`.

Ejemplo:

```
h:dataTable value="#{reportController.reports}" var="report">
  <f:facet name="header">
    <h:outputText value="Reports" />
  </f:facet>
  ...
</h:dataTable>
```

Etiqueta	Descripción
<f:loadBundle>	Carga un fichero de recursos y lo hace disponible a través de una variable. El path del fichero debe estar disponible en el classpath de la aplicación web (esto es, en el directorio `WEB-INF/classses`).

Ejemplo:

```
<f:loadBundle basename="messages" var="msgs" />
<h:outputText value="#{msgs.title}" />
```

Etiqueta	Descripción
<f:param>	Define un parámetro de una expresión de texto.

Ejemplo:

```
<f:loadBundle basename="messages" var="msgs" />
<h:outputFormat value="#{msgs.sunRiseAndSetText}">
  <f:param value="#{city.sunRiseTime}" />
  <f:param value="#{city.sunSetTime}" />
</h:outputFormat>
```

Etiqueta	Descripción
<f:selectItem>	Crea una instancia de un ítem y se lo asigna al componente padre.

Ejemplo:

```
<h:form>
  <h:selectManyCheckbox value="#{user.projects}">
    <f:selectItem itemValue="JSF" itemValue="1" />
    <f:selectItem itemValue="JSP" itemValue="2" />
    <f:selectItem itemValue="Servlets" itemValue="3" />
  </h:selectManyCheckbox>
</h:form>
```

Etiqueta	Descripción
<f:selectItems>	Crea múltiples instancias de ítems y los asigna al componente padre.

Ejemplo:

```
<h:form>
  <h:selectManyCheckbox value="#{user.projects}">
    <f:selectItems value="#{allProjects}" />
```

```
  </h:selectManyCheckbox>
</h:form>
```

Etiqueta	Descripción
<f:subView>	Crea un grupo de componentes dentro de una vista.

Ejemplo:

```
<f:view>
  <f:subview id="header">
    <jsp:include page="header.jsp" />
  </f:subview>
  ...
  <f:subview id="footer">
    <jsp:include page="footer.jsp" />
  </f:subview>
</f:view>
```

Etiqueta	Descripción
<f:validateDoubleRange>	Crea una instancia de validador de rango doble y lo asigna al componente padre.

Ejemplo:

```
<h:inputText value="#{product.price}">
  <f:convertNumber type="currency" />
  <f:validateDoubleRange minimum="0.0" />
</h:inputText>
```

Etiqueta	Descripción
<f:validateLength>	Crea una instancia de validador de longitud de texto y lo asigna al componente padre.

Ejemplo:

```
<h:inputText value="#{user.zipCode}">
  <f:validateLength minimum="5" maximum="5" />
</h:inputText>
```

Etiqueta	Descripción
<f:validateLongRange>	Crea una instancia de validador de rango long y lo asigna al componente padre.

Ejemplo:

```
<h:inputText value="#{employee.salary}">
  <f:convertNumber type="currency" />
  <f:validateLongRange minimum="50000" maximum="150000" />
</h:inputText>
```

Etiqueta	Descripción

<f:validator>	Crea un validador definido por el usuario.

Ejemplo:

```
<h:form>
  <h:inputText value="#{user.ssn}">
    <f:validator validatorId="ssnValidator" />
  </h:inputText>
</h:form>
```

Etiqueta	Descripción
<f:valueChangeListener>	Crea una instancia de la clase definida por el atributo `type` y la asigna al componente padre para ser llamada cuando sucede un evento de cambio de valor.

Ejemplo:

```
<h:form>
  <h:selectBooleanCheckbox value="Details" immediate="true">
    <f:valueChangeListener type="com.mycompany.DescrLevelListener" />
  </h:selectBooleanCheckbox>
</h:form>
```

Etiqueta	Descripción
<f:verbatim>	Permite renderizar texto.

Ejemplo:

```
<f:subview id="header">
  <f:verbatim>
    <html>
      <head>
        <title>Welcome to my site!</title>
      </head>
  </f:verbatim>
</f:subview>
```

Etiqueta	Descripción
<f:view>	Crea una vista JSF.

Ejemplo:

```
<f:view locale="#{user.locale}">
  ...
</f:view>
```

Etiqueta	Descripción
<f:phaseListener>	Añade un phase listener a la vista padre. Esto nos permite realizar llamadas antes y después de la fase que nosotros queramos dentro del ciclo de vida de JSF

Ejemplo:

```
<h:commandButton action="#{jsfBean.submit}" value="Submit">
  <f:phaseListener binding="#{jsfBean.phaseListenerImpl}"
    type="org.especialistajee.jsf.PhaseListenerImpl"/>
</h:commandButton>
```

Etiqueta	Descripción
<f:event>	listener de eventos

Ejemplo:

```
<h:outputText id="beforeRenderTest1" >
    <f:event type="javax.faces.event.beforeRender"
        action="#{eventTagBean.beforeEncode}" />
</h:outputText>
```

Etiqueta	Descripción
<f:validateRequired>	El valor es obligatorio

Ejemplo:

```
<h:inputSecret id="password" value="#{user.password}">
    <f:validateRequired />
</h:inputSecret>
```

Etiqueta	Descripción
<f:validateRegex>	Valida un valor contra una expresión regular

Ejemplo:

```
<h:inputSecret id="password" value="#{user.password}">
  <f:validateRegex
pattern="((?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%]).{6,20})" />
</h:inputSecret>
```

Etiqueta	Descripción
<f:validateBean>	Hace uso del API de validación de beans (JSR 303) para realizar la validación.

Ejemplo:

```
<h:inputText value="#{sampleBean.userName}">
  <f:validateBean disabled="true" />
</h:inputText>
```

Etiqueta	Descripción
<f:viewParam>	Define un parámetro en la vista que puede inicializarse a partir de cualquier parámetro de la petición.

Ejemplo:

```
<f:metadata>
    <f:viewParam name="id" value="#{bean.id}" />
</f:metadata>
```

Etiqueta	Descripción
<f:metadata>	Agrupa viewParams.

Ejemplo:

```
<f:metadata>
    <f:viewParam name="id" value="#{bean.id}" />
</f:metadata>
```

Etiqueta	Descripción
<f:ajax>	Dota de comportamiento AJAX a los componentes.

Ejemplo:

```
<h:inputSecret id="passInput" value="#{loginController.password}">
  <f:ajax event="keyup" render="passError"/>
</h:inputSecret>
```

4. Ejercicios sesión 2 - MVC

4.1. Login de usuario (1 punto)

Vamos a añadir una pequeña regla de navegación a nuestra aplicación, que consistirá en el login del usuario.

Crearemos un controlador al que llamaremos `es.ua.jtech.jsf.AccessController`. Éste deberá tener un método llamado `doLogin`, que comprobará que tanto el login como el password sean la palabra 'admin'. En caso de que no sea así, vuelve a la página de registro y muestra en ella un mensaje de error.

Pista: En la solución que se os dará, el mensaje de error aparecerá en la cabecera de esta manera:

```
<h:panelGroup
     layout="block"
     rendered="#{accessController.loginError}"
     style="color:#B94A48; background-color: #F2DEDE; border: 1px solid
#B94A48; padding: 5px">
    <h:outputText value="#{accessController.errorMsg}" />
</h:panelGroup>
```

4.2. Guardando el usuario recién logueado (1 punto)

Cuando hayamos realizado el login, deberemos almacenar la información del usuario en algún lado. Para ello, nos crearemos un *managed bean*, que tendrá el siguiente esqueleto:

```
package es.ua.jtech.jsf.model;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.faces.bean.ManagedBean;

@ManagedBean(name="user")
//DETERMINAR ÁMBITO
public class UserBean implements Serializable {
   private String name;
   private List<TaskBean> tasks = null;

   public UserBean(){
     tasks = new ArrayList<TaskBean>();
   }

   public void setTasks(List<TaskBean> tasks) {
     this.tasks = tasks;
   }

   public void addTask(TaskBean t){
      ...
   }

   public void removeTask(int index){
```

```
      ...
   }

    //GETTERS Y SETTERS

}
```

Por su parte, el objeto TaskBean tendrá la forma:

```
package es.ua.jtech.jsf.model;

import java.io.Serializable;

public class TaskBean implements Serializable {
   int id=-1;
   String title;
   String description;

   //GETTERS & SETTERS
}
```

Como aún no tenemos la lógica de la parte de las tareas realizada, una vez hagamos login se nos redirigirá a una vista donde tengamos un mensaje que diga "Bienvenido *admin*", y un `commandLink` para poder hacer logout

Deberemos crear una regla de navegación en el fichero `faces-config.xml` que nos lleve de la página de login a esta que acabamos de crear.

4.3. Logout del usuario

En el ejercicio anterior, hemos hecho el login del usuario y lo hemos llevado a una página donde la única posibilidad es hacer logout. Así, ahora lo que tendremos que hacer será un método `doLogout` en el `AccessController`. Éste se encargará de invalidar la sesión y redirigirnos a la página de login nuevamente.

Como los logouts suelen poder hacerse en distintos puntos de una aplicación, la regla de navegación que insertemos aquí deberá hacer uso de *wildcards* en la etiqueta `from-view-id`

5. El ciclo de vida de JSF

5.1. La arquitectura JSF

En esta sesión vamos a estudiar en profundidad cómo se gestionan las peticiones a JavaServer Faces, incluyendo detalles de la implementación de esta arquitectura. Igual que en las sesiones pasadas, utilizaremos un ejemplo concreto para ilustrar todos los aspectos. Será una sencilla aplicación web que llamaremos *Calculadora* que permite al usuario realizar operaciones matemáticas.

Veremos los conceptos del ciclo de vida de una petición JSF, qué es una petición, cómo se validan los datos que el usuario introduce en el componente, cómo obtiene el componente los datos del modelo y cómo se procesan y definen los eventos asociados.

Tal y como veíamos en la sesión anterior, y definiéndolo de una forma muy simple, JSF es un framework orientado a recoger datos del usuario, pasarlos a la capa del modelo de la aplicación, realizar las acciones correspondientes en la aplicación y *pintar* los datos resultantes. Todo ello en un entorno web, con peticiones HTTP y páginas HTML.

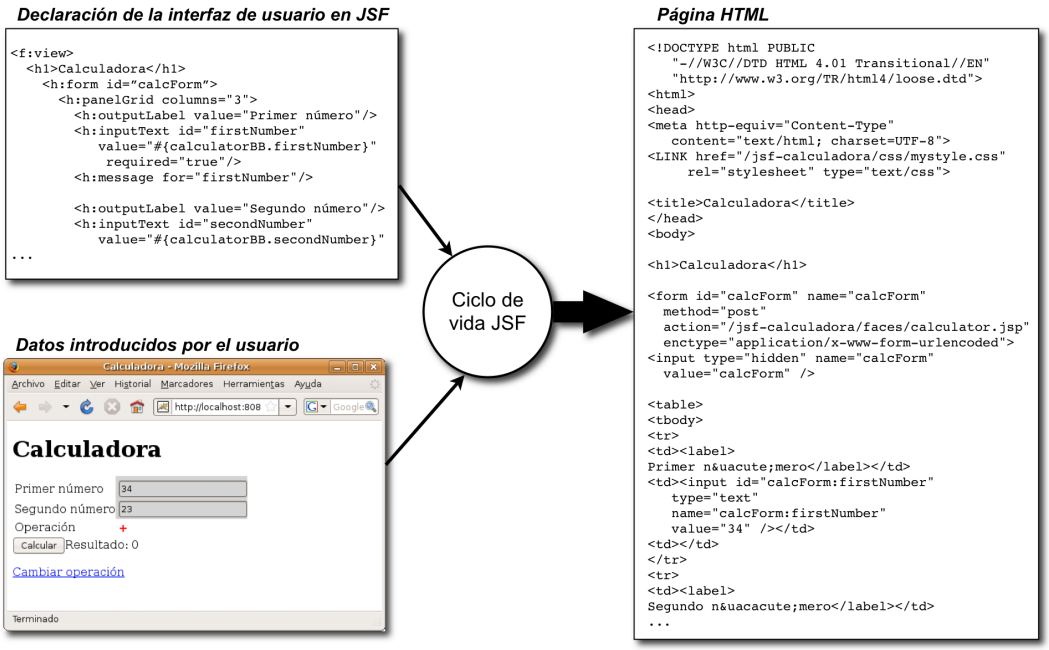
Los datos se introducen y se muestran en forma de texto, y se almacenan en un formato dependiente de la aplicación. Por ejemplo, una fecha se puede representar con un formato `dd-mm-aaaa` mientras que su representación interna puede ser un objeto de la clase `java.util.Date` . Para realizar esta conversión entre el texto y el formato interno, se asocian al componente *validadores* y *conversores* .

Como vimos en la sesión pasada, esta separación entre la parte visual del componente (código HTML en la página web), el modelo de datos (*managed beans*) y las acciones (código Java que procesa el modelo y controla la vista) es un esquema tradicional en todos los framework de gestión de interfaces de usuario que se denomina *patrón Modelo/Vista/Controlador (MVC)* .

La aportación fundamental de la tecnología JSF es la adaptación del patrón MVC al entorno web. Para ello, el código final en el que se define un componente es código HTML y los eventos disparados por el usuario se guardan en la petición HTTP. Un *servlet* de la clase `javax.faces.webapp.FacesServlet` es el motor de cualquier aplicación JSF. Este servlet procesa la petición, gestiona todos los componentes relacionados y termina generando el código HTML en el que se traducen estos componentes.

Recordemos, como vimos en la sesión pasada, que el funcionamiento básico de JSF cuando recibe una petición JSF consiste en obtener la vista JSF, procesarla con los datos introducidos por el usuario y que llegan en la petición y generar una página HTML como resultado. Este proceso (petición, procesamiento y generación de página HTML) es lo que se denomina el **ciclo de vida JSF** . Veremos con detalle los pasos que realiza la

arquitectura JSF dentro de este ciclo de procesamiento. La siguiente figura muestra un ejemplo concreto de la aplicación Calculadora que utilizaremos en esta sesión.



5.2. Ciclo de vida

Cuando se carga la aplicación web en el servidor se inicializa el framework JSF. Se lee el fichero de configuración `faces-config.xml` y se crean los beans gestionados definidos con el ámbito `application` , realizando las sentencias de incialización necesarias. Después el motor de JSF está listo para recibir peticiones y para lanzar el ciclo de vida de JSF con cada una.

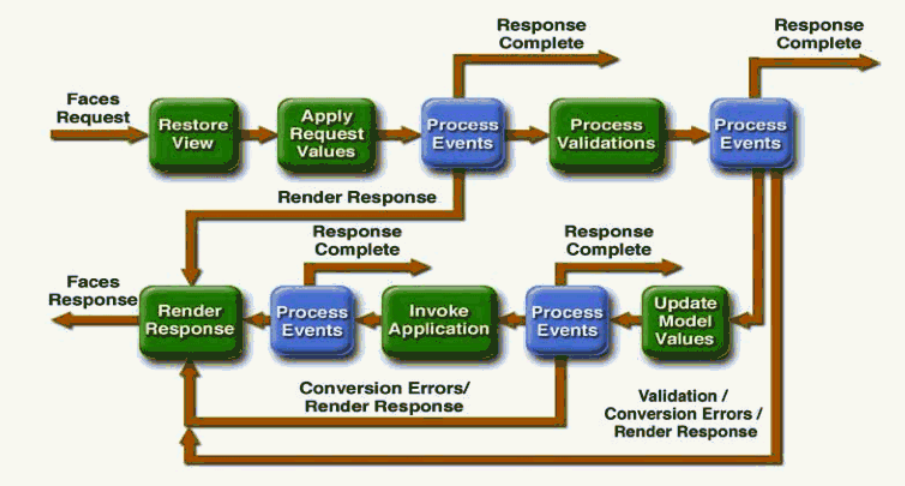
Lo que en JSF se denomina ciclo de vida no es más que una secuencia de fases por las que pasa una petición JSF desde que se recibe en el servidor hasta que se genera la página HTML resultante. El servlet que implementa el framework (`javax.faces.webapp.FacesServlet`) recibe la petición y realiza todo el ciclo, creando y utilizando los objetos Java que representan los componentes JSF y los beans gestionados. La relación entre estos objetos y la generación de código HTML a partir del árbol de componentes constituyen la base del funcionamiento del framework.

Las fases del ciclo de vida son las siguientes:

1. **Restaurar la vista** (*restore view*). En este paso se obtiene el árbol de componentes correspondiente a la vista JSF de la petición. Si se ha generado antes se recupera, y si es la primera vez que el usuario visita la página, se genera a partir de la descripción JSF.
2. **Aplicar los valores de la petición** (*apply request values*). Una vez obtenido el árbol

de componentes, se procesan todos los valores asociados a los mismos. Se convierten todos los datos de la petición a tipos de datos Java y, para aquellos que tienen la propiedad `inmediate` a cierta, se validan, adelantándose a la siguiente fase.

3. **Procesar las validaciones** (*process validations*). Se validan todos los datos. Si existe algún error, se encola un mensaje de error y se termina el ciclo de vida, saltando al último paso (renderizar respuesta).
4. **Actualizar los valores del modelo** (*update model values*). Cuando se llega a esta fase, todos los valores se han procesado y se han validado. Se actualizan entonces las propiedades de los beans gestionados asociados a los componentes.
5. **Invocar a la aplicación** (*invoke application)* . Cuando se llega a esta fase, todas las propiedades de los beans asociados a componentes de entrada (*input*) se han actualizado. Se llama en este momento a la acción seleccionada por el usuario.
6. ***Renderizar* la respuesta** (*render response*).



Ciclo de vida de una petición JSF

Al final de cada una de las fases, se comprueba si hay algún evento que debe ser procesado en esa fase en concreto y se llama a su manejador. También se llaman a los manejadores de los eventos que deben ser procesados en cualquier fase. Los manejadores, a su vez, pueden saltar a la última fase del ciclo para renderizar el árbol de componentes llamando al método `renderResponse()` del `FacesContext` . También pueden renderizar el componente asociado al evento y llamar al método `responseComplete()` del `FacesContext` para terminar el ciclo de vida.

JSF emite un evento `PhaseListener` al comienzo y al final de cada fase del ciclo de vida de la petición. Para capturar el evento, debemos definir una clase que implemente la interfaz `PhaseListener` y sus métodos `beforePhase` y `afterPhase` .

En el ejemplo que vamos a ver más adelante (`calculator`) se podría hacer de la siguiente forma:

```
package calculator.controller;
```

```
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;
import javax.faces.event.PhaseListener;

public class CalculatorPhaseListener implements PhaseListener {

  public void beforePhase(PhaseEvent pe) {
    FacesContext context = FacesContext.getCurrentInstance();
    if (pe.getPhaseId() == PhaseId.RESTORE_VIEW)
      context.addMessage(
        null,
        new FacesMessage("Procesando una nueva peticion!")
      );
      context.addMessage(
        null,
        new FacesMessage("antes de - " + pe.getPhaseId().toString())
      );
  }

  public void afterPhase(PhaseEvent pe) {
    FacesContext context = FacesContext.getCurrentInstance();
    context.addMessage(
      null,
      new FacesMessage("despues de - " + pe.getPhaseId().toString())
    );

    if (pe.getPhaseId() == PhaseId.RENDER_RESPONSE)
      context.addMessage(
        null,
        new FacesMessage("Peticion terminada!")
      );
  }

  public PhaseId getPhaseId() {
    return PhaseId.ANY_PHASE;
  }
}
```

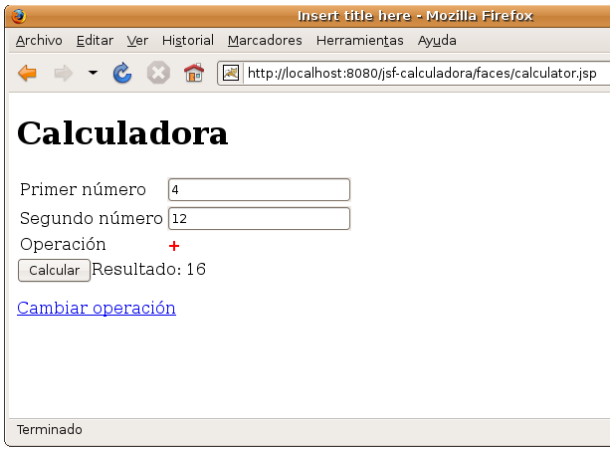
Para que el framework llame a este manejador hay que añadir en el fichero `faces-config.xml` la siguiente declaración:

```
<lifecycle>
   <phase-listener>
      calculator.controller.CalculatorPhaseListener
   </phase-listener>
</lifecycle>
```

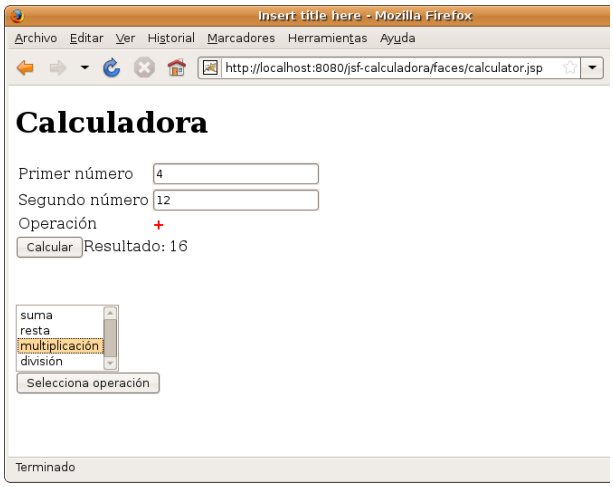
Se pueden ver los mensajes con la etiqueta `h:messages` dentro de cualquier vista (`<f:view>`)

5.3. Un programa de ejemplo

Veamos en primer lugar un sencillo programa ejemplo en el que vamos a presentar algunas de las características presentadas. Se trata de un programa que implementa una simple calculadora de números enteros. Su apariencia es la siguiente:



Vemos que la interfaz de usuario se construye en una única página, en la que se definen dos campos de texto para introducir los números a operar y un botón para realizar la operación matemática. Por defecto, se utiliza la operación suma, pero es posible cambiar esta operación pinchando en la opción *Cambiar operación* . Entonces, en la misma página, se muestra una lista de operaciones de las que el usuario puede seleccionar una, pulsando el botón *Selecciona operación* :



Una vez seleccionada la nueva operación, se vuelve a una configuración similar a la primera figura, pero con la nueva operación.

Vamos a utilizar esta aplicación para explicar con un poco de más detalle el ciclo de vida de las peticiones y la validación de los datos. Comencemos estudiando su código fuente, analizando los elementos que funcionan como vista, modelo y controlador.

- **Vista** : Toda la vista de la aplicación se implementa con un único fichero JSF, el fichero `calculator.xhtml` . En él se definen varios componentes, organizados en dos elementos de tipo `h:form` . El primero contiene la calculadora propiamente dicha: los campos para los números, la operación, el botón para calcular el resultado y el

resultado propiamente dicho. Al final se añade un enlace con una acción para activar el cambio de operación. El segundo elemento es la lista de operaciones y el botón para confirmar el cambio de la operación activa. Este componente y el enlace para activarlo se ocultan y se muestran, dependiendo del estado de la interfaz. Veremos cómo se consigue este efecto.

- **Modelo** . Se utiliza un único bean, llamado `calculatorBean` , en el que se han definido las propiedades `firstNumber` , `secondNumber` , `operation` y `result` . Además, se utiliza la clase `CalculatorBO` que implementa la "lógica de negocio" y que es con la que que realizamos finalmente la operación.
- **Controlador** . El controlador es un objeto de la clase `CalculatorController` que tiene la responsabilidad de realizar la operación entre los dos números y actualizar el bean con el resultado y también de guardar y modificar las propiedades que determinan la visibilidad de los componentes. El controlador debe tener acceso al bean `calculatorBean` que define el modelo. Para eso se define en el controlador la propiedad `numbers` que se inicializa con el bean `calculatorBean` en el fichero de configuración `faces-config.xml` .

Veamos con detalle cómo se implementan estos elementos.

5.3.1. Vista

5.3.1.1. Código fuente

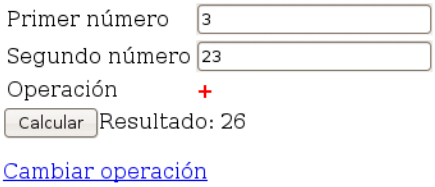
Comenzamos por la **vista** definida en el fichero `calculator.xhtml` .

Fichero `calculator.xhtml` :

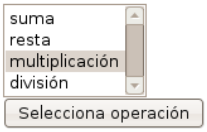
```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core">
<h:head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<LINK href="<%=request.getContextPath()%>/css/mystyle.css"
  rel="stylesheet" type="text/css">
<title>Calculadora</title>
</h:head>
<h:body>
  <h1>Calculadora</h1>
  <h:form>
    <h:panelGrid columns="3">
      <h:outputLabel value="Primer número"/>
      <h:inputText id="firstNumber"
          value="#{calculatorBean.firstNumber}"
          required="true"/>
      <h:message for="firstNumber"/>

      <h:outputLabel value="Segundo número"/>
      <h:inputText id="secondNumber"
          value="#{calculatorBean.secondNumber}"
          required="true"/>
      <h:message for="secondNumber" />

      <h:outputLabel value="Operación"/>
```

```
      <h:outputLabel value="#{calculatorBean.operation}"
                     styleClass="strong"/>
      <h:outputLabel value=""/>
    </h:panelGrid>

    <h:commandButton value="Calcular"
        action="#{calculatorController.doOperation}"/>
    <h:outputText value="Resultado: #{calculatorBean.result}"/><br/>

    <p></p>

    <h:commandLink
rendered="#{calculatorController.newOperationCommandRendered}"
        action="#{calculatorController.doNewOperation}"
        value="Cambiar operación"/>
    </h:form><br/>

    <p></p>

    <h:form
rendered="#{calculatorController.selectOperationFormRendered}">
      <h:selectOneListbox value="#{calculatorBean.operation}">
        <f:selectItem itemValue="+" itemLabel="suma"/>
        <f:selectItem itemValue="-" itemLabel="resta"/>
        <f:selectItem itemValue="*" itemLabel="multiplicación"/>
        <f:selectItem itemValue="/" itemLabel="división"/>
      </h:selectOneListbox><br/>
      <h:commandButton action="#{calculatorController.doSelectOperation}"
          value="Selecciona operación"/>
    </h:form>
</h:body>
</html>
```

Primero se define el componente que contiene la calculadora propiamente dicha. Se trata de un `h:form` que contiene un `h:panelGrid` con 3 columnas. En la primera columnas se colocan los `h:outputLabel` que describen el elemento de la calculadora que hay a su derecha. En la segunda columna se colocan los datos propiamente dichos, conectados con las distintas propiedades del bean `calculatorBean` : `firstNumber` , `secondNumber` y `operation` .



El segundo bloque de componentes de la página es un `h:form` que contiene una caja de selección, un `h:selectOnListbox` , con todas las posibles operaciones y un `h:commandButton` asociado a la acción para confirmar el cambio de operación.



5.3.1.2. Renderizado de componentes

Vamos ahora a uno de los puntos importantes del ejemplo. ¿Cómo se hace aparecer y desaparecer los componentes en la página? Podemos ver en este componente, y en el *commandLink Cambiar operación* , la opción `rendered` , que indica si el componente va a ser renderizado y volcado en el HTML que se envía al navegador.

```
...
<h:commandLink
rendered="#{calculatorController.newOperationCommandRendered}"
    action="#{calculatorController.doNewOperation}"
 ...
<h:form rendered="#{calculatorController.selectOperationFormRendered}"
   <h:selectOneListbox value="#{calculatorBean.operation}">
         <f:selectItem itemValue="+" itemLabel="suma"/>
...
```

Si fijáramos en el atributo `rendered` el valor `true` o `false` haríamos que siempre se mostraran o se escondieran los componentes. Esto no es muy útil. Esta opción (como muchas otras de los componentes JSF) se vuelve interesante de verdad cuando hacemos lo que aparece en el ejemplo. Ligamos el atributo a una propiedad del bean de forma que podemos modificar el estado del componente en función de los valores de los objetos del modelo y de las opciones seleccionadas por el usuario. En este caso ligamos al atributo `redered` las propiedades `newOperationCommandRendered` y `selectOperationFormRendered` del bean que hace de controlador. De esta forma podemos hacer que aparezca o desaparezcan los componentes poniendo a `true` o `false` esas propiedades. Como la fase de render es la última de todas las fases, el estado de los componentes dependerá de las modificaciones que las acciones y eventos hayan realizado en las propiedades del bean.

5.3.1.3. Hoja de estilos

Un detalle también interesante es la forma de añadir CSS a la página. Incluimos la hoja de estilos con la directiva:

```
<LINK href="<%=request.getContextPath()%>/css/mystyle.css"
  rel="stylesheet" type="text/css">
```

Y después indicamos la clase CSS del `outputlabel` con el atributo `styleClass` :

```
<h:outputLabel value="#{calculatorBean.operation}" styleClass="strong"/>
```

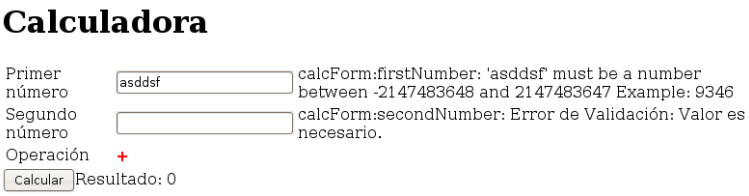
El contenido del fichero con la hoja de estilos es sencillo:

Fichero **`WebContent/css/mystyle.css`**

```
.strong {
  font-weight:bold;
  color: red;
}
```

5.3.1.4. Errores de validación

Como se puede comprobar en el código JSF, se han definido en la tercera columna del panel de la calculadora los mensajes de error JSF asociados a los componentes y que pueden generarse en la conversión o validación de los números introducidos por el usuario. La siguiente figura muestra los mensajes de error generados por un valor que es imposible de convertir al tipo de datos de la propiedad.



5.3.2. Modelo

El código Java que da soporte a estos componentes se implementa en las clases `calculator.model.Calculator` y `calculator.controller.CalculatorController`. El primero define la *capa de negocio* de la aplicación con los posibles casos de uso y su implementación en código Java.

Fichero `es.ua.jtech.jsf.CalculatorBO` :

```
package es.ua.jtech.jsf;

public class CalculatorBO {

   public int add(int a, int b) {
      return a + b;
   }

   public int substract(int a, int b) {
      return a - b;
   }

   public int multiply(int a, int b) {
      return a * b;
   }

   public int divide(int a, int b) {
      return a / b;
   }
}
```

```
package es.ua.jtech.jsf;

public class CalculatorBean {
  private int firstNumber = 0;
  private int secondNumber = 0;
  private String operation = "+";
  private int result = 0;

  public void setFirstNumber(int firstNumber) {
    this.firstNumber = firstNumber;
```

```
  }

  public void setResult(int result) {
    this.result = result;
  }

  public int getFirstNumber() {
    return firstNumber;
  }

  public void setSecondNumber(int secondNumber) {
    this.secondNumber = secondNumber;
  }

  public int getSecondNumber() {
    return secondNumber;
  }

  public void setOperation(String operation) {
    this.operation = operation;
  }

  public String getOperation() {
    return operation;
  }

  public int getResult() {
    return result;
  }
}
```

5.3.3. Controlador

La segunda clase, `calculator.controller.CalculatorController` define el *bean* gestionado y el método de acción que realiza la operación matemática seleccionada por el usuario.

Fichero `es.ua.jtech.jsf.CalculatorController` :

```
package es.ua.jtech.jsf;

public class CalculatorController {

  private CalculatorBean numbers;
  private CalculatorBO calculator = new CalculatorBO();
  private boolean selectOperationFormRendered=false;
  private boolean newOperationCommandRendered=true;

  public boolean isSelectOperationFormRendered() {
    return selectOperationFormRendered;
  }

  public void setSelectOperationFormRendered(boolean
selectOperationFormRendered) {
    this.selectOperationFormRendered = selectOperationFormRendered;
  }

  public boolean isNewOperationCommandRendered() {
    return newOperationCommandRendered;
  }

  public void setNewOperationCommandRendered(boolean
```

```
newOperationCommandRendered) {
    this.newOperationCommandRendered = newOperationCommandRendered;
  }

  public CalculatorBean getNumbers() {
    return numbers;
  }

  public void setNumbers(CalculatorBean numbers) {
    this.numbers = numbers;
  }

  public String doNewOperation() {
    selectOperationFormRendered=true;
    newOperationCommandRendered=false;
    return null;
  }

  public String doSelectOperation() {
    selectOperationFormRendered=false;
    newOperationCommandRendered=true;
    doOperation();
    return null;
  }

  public String doOperation() {
    String operation = numbers.getOperation();
    int firstNumber = numbers.getFirstNumber();
    int secondNumber = numbers.getSecondNumber();
    int result = 0;
    String resultStr = "OK";

    if (operation.equals("+"))
      result = calculator.add(firstNumber, secondNumber);
    else if (operation.equals("-"))
      result = calculator.substract(firstNumber, secondNumber);
    else if (operation.equals("*"))
      result = calculator.multiply(firstNumber, secondNumber);
    else if (operation.equals("/"))
      result = calculator.divide(firstNumber, secondNumber);
    else
      resultStr="not-OK";

    numbers.setResult(result);
    return resultStr;
  }
}
```

Por último, el fichero de configuración `faces-config.xml` relaciona el nombre lógico del *bean* `calculatorController` con la clase `calculator.Controller.CalculatorController` y le asigna un alcance de sesión. También define el bean `calculatorBean` con alcance sesión e inicializa la propiedad `numbers` de `calculatorController` con este bean recién creado. De esta forma es posible acceder al bean que representa el modelo desde el controlador, cuando haya que realizar una acción.

También se definen en este fichero las reglas de navegación entre las vistas JSF. Tras la página `/calculator.xhtml` se muestra (si el resultado de la acción es " `OK` ") la página `/result.xhtml` . Y tras realizar una acción en la página `/result.xhtml` se muestra (si el resultado de la acción es la cadena `OK` la página `/calculator.xhtml`).

Fichero **`faces-config.xml`** :

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

<faces-config
    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_1_2.xsd"
    version="1.2">
  <managed-bean>
    <managed-bean-name>calculatorBean</managed-bean-name>
<managed-bean-class>es.ua.jtech.jsf.CalculatorBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
  <managed-bean>
    <managed-bean-name>calculatorController</managed-bean-name>
<managed-bean-class>es.ua.jtech.jsf.CalculatorController</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
      <property-name>numbers</property-name>
      <property-class>es.ua.jtech.jsf.CalculatorBean</property-class>
      <value>#{calculatorBean}</value>
    </managed-property>
  </managed-bean>
</faces-config>
```

5.4. Conversión de formatos entre la interfaz y los beans

Una de las ayudas que nos proporciona JSF es la conversión de formatos entre los componentes y los beans. Los datos de la interfaz suelen ser cadenas de texto. Los tipos de los datos de los beans dependen del modelo. JSF debe realizar esta conversión de formatos en la fase *Apply Request Value* para convertir los datos del usuario a datos del modelo y en la fase *Render Response* para hacer la conversión inversa. En concreto, el método JSF que realiza esta conversión es `decode()` , que está definido en todos los objetos componentes.

Por ejemplo, en el caso de la aplicación Calculadora, los dos números introducidos por el usuario se guardan en las propiedades `firstNumber` y `secondNumber` . Y el número resultante se guarda en la propiedad `result` . Todas las propiedades son de tipo `int` . Cuando el usuario introduce los números se deben convertir a este tipo. Y al revés; cuando se genera la página se debe transformar del tipo de datos `int` a la cadena que se introduce en la página HTML.

Si no se especifica nada en el componente, JSF utilizar el conversor por defecto de texto al tipo de datos del bean. Es posible también escoger el conversor que queremos utilizar en el componente, incluyendo en el componente la etiqueta `f:converter` y un identificador del conversor. Por ejemplo, para indicar que queremos aplicar un conversor de fecha con un formato corto a un componente de salida de texto, debemos especificar lo siguiente:

```
<h:outputText value="Fecha de salida: #{bean.fechaSalida}">
   <f:convertDateTime dateStyle="short"/>
</h:outputText>
```

Los posibles formatos de fecha son los siguientes, en el locale Inglés:

Tipo	Formato
`default`	Sep 9, 2003 5:41:15 PM
`short`	9/9/03 5:41 PM
`medium`	Sep 9, 2003 5:41:15 PM
`long`	September 9, 2003 5:41:15 PM PST
`full`	Tuesday, September 9, 2003 5:41:15 PM PST

La conversión de la fecha depende del `locale` activo para la aplicación. El locale, y el fichero de recursos asociado, se configura en el fichero `faces-config` :

```
<application>
<locale-config>
  <default-locale>es</default-locale>
  <supported-locale>en</supported-locale>
</locale-config>
<message-bundle>
  es.ua.jtech.MessageResources
</message-bundle>
</application>
```

Si durante la conversión algo va mal y se produce un error (debido, por ejemplo, a que el usuario no ha introducido correctamente el valor), se marca el valor como no válido y se añade un mensaje de error a la lista mantenida en el contexto de la sesión JSF, implementado por un objeto de la clase `FacesContext` . Esta es la misma lista que será utilizada también por los validadores. Los mensajes de error pueden mostrarse con las etiquetas `h:messages` (todos los mensajes) y `h:message for:` *identificador* . Por ejemplo, en el siguiente fragmento de página JSF definimos un componente con el identificador `firstNumber` y escribimos a continuación el mensaje de error que se pueda generar en él:

```
<h:outputLabel value="Primer número"/>
<h:inputText id="firstNumber"
         value="#{calculatorBean.firstNumber}"
 required="true"/>
<h:message for="firstNumber"/>
```

5.4.1. Custom converters

JSF nos permite crearnos conversores específicos para cubrir necesidades más específicas, como por ejemplo DNIs/pasaportes, números de cuenta bancaria y, en general, cualquier objeto que necesitemos.

Un conversor es una clase que convierte de String a objeto y viceversa. Debe implementar la interfaz Converter, que proporciona los métodos:

- *Object getAsObject(FacesContext context, UIComponent component, String*

newValue): Convierte un String en un objeto del tipo deseado. Lanza una ConverterException si la conversión no se puede llevar a cabo.

- *String getAsString(FacesContext context, UIComponent component, Object value)*: Convierte un objeto en String para que pueda mostrarse en la pantalla del usuario.

```
package es.ua.jtech.jsf.converter;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;
import javax.faces.convert.FacesConverter;

import es.ua.jtech.jsf.beans.DniBean;

@FacesConverter("conversorDni")
public class DniConverter implements Converter {

  public Object getAsObject(FacesContext context, UIComponent component,
String value)
          throws ConverterException {
    boolean situacionDeError = false;
    DniBean dni = new DniBean();
    dni.setNumero(value.substring(0, 8));
    dni.setLetra(value.substring(8, 9));

    if (situacionDeError) {
      FacesMessage message = new FacesMessage(FacesMessage.SEVERITY_ERROR,
           "Se ha producido un error en la conversión",
           "Detalle del error");
      throw new ConverterException(message);
    }

    return dni;
  }

  public String getAsString(FacesContext context, UIComponent component,
Object value)
        throws ConverterException {
    DniBean dni = (DniBean) value;
    return dni.getNumero() + dni.getLetra();
  }
}
```

Como hemos visto, definimos nuestro conversor con la anotación *@FacesConverter*, a la que se le ha asignado un ID (en este caso, conversorDni). Así, para validar el dni del usuario, habrá que usar el conversor como hemos visto anteriormente:

```
<h:inputText value="#{usuario.dni}">
   <f:converter converterId="conversorDni"/>
</h:inputText>
```

o bien

```
<h:inputText value="#{usuario.dni}" converter="conversorDni"/>
```

Yendo un poco más allá, podemos anotar nuestro conversor de la siguiente manera, para que se aplique siempre que usemos un objeto del tipo Dni.class

```
@FacesConverter(forClass=Dni.class)
```

```
...
```

De esta manera, y simplemente usando *<h:inputText value="#{usuario.dni}" />*, la implementación de JSF buscará los conversores definidos para esta clase.

En algunas ocasiones, puede que sea necesario enviar algún parámetro adicional a nuestro conversor. Para ello, usamos el tag *f:attribute*.

```
<h:outputText value="#{usuario.dni}">
   <f:converter converterId="es.ua.jtech.Dni"/>
   <f:attribute name="separador" value="-"/>
</h:outputText>
```

Así, en nuestro conversor recogeremos el atributo de la siguiente manera:

```
String separator = (String) component.getAttributes().get("separator");
```

5.5. Validación

Una vez que se ha convertido con éxito el valor, los validadores se encargan de asegurar que los datos de nuestra aplicación tienen los valores esperados, como por ejemplo:

- Una fecha tiene el formato dd/MM/yyyy
- Un float se encuentra entre los valores 1.0 y 100.0

La implementación JSF provee un mecanismo que nos permite realizar una serie de validaciones sobre un componente, simplemente añadiendo un tag dentro del mismo:

Tag	Validator	Atributos	Descripción
`f:validateDoubleRa`	`DoubleRangeValidat`	`minimum, maximum`	Un valor `double`, con un rango opcional
`f:validateLongRang`	`LongRangeValidator`	`minimum, maximum`	Un valor `long`, con un rango opcional
`f:validateLength`	`LengthValidator`	`minimum, maximum`	Un `String`, con un mínimo y un máximo de caracteres
`f:validateRequired`	`RequiredValidator`		Valida la presencia de un valor
`f:validateRegex`	`RegexValidator`	`pattern`	Valida un `String` contra una expresión regular
`f:validateBean`	`BeanValidator`	`validation-Groups`	Especifica grupos de validación para los validadores

Aviso:

Los espacios en blanco por defecto cuentan como valores válidos en un `required`. Si no nos

interesa que esto ocurrra, en JSF 2.0 podemos modificar este comportamiento estableciendo, en el fichero `web.xml`, el parámetro de contexto `javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL` a `true`

Al igual que con los conversores, si algún valor no cumple la validación se marca como no válido y se añade un mensaje de error al `FacesContext` . Por ejemplo, el siguiente código comprobaría que el número introducido es mayor que cero

```
<h:outputLabel value="Primer número"/>
<h:inputText id="firstNumber"
   value="#{calculatorBean.firstNumber}"
   required="true">
      <f:validateLongRange minimum="0"/>
</h:inputText>
<h:message for="firstNumber"/>
```

Desde JSF 1.2 podemos definir un mensaje personalizado para un componente, estableciendo valores a los atributos `requiredMessage` y/o `validatorMessage`:

```
<h:inputText id="card" value="#{usuario.dni}" required="true"
      requiredMessage="El DNI es obligatorio"
      validatorMessage="El DNI no es válido">
   <f:validateLength minimum="9"/>
</h:inputText>
```

Además, podemos sobreescribir lso mensajes de error de los validadores estándar en un fichero de propiedades:

Resource ID	Texto por defecto
`javax.faces.component.UIInput.REQUIRE`	{0}: Validation Error: Value is required
`javax.faces.validator.DoubleRangeVali` `javax.faces.validator.LongRangeValida`	{2}: ValidationError: Specified attribute is not between the expeced values of {0} and {1}
`javax.faces.validator.DoubleRangeVali` `javax.faces.validator.LongRangeValida`	{1}: Validation Error: Value is greater than allowable maximum of {0}
`javax.faces.validator.DoubleRangeVali` `javax.faces.validator.LongRangeValida`	{1}: Validation Error: Value is less than allowable minimum of {0}
`javax.faces.validator.DoubleRangeVali` `javax.faces.validator.LongRangeValida`	{1}: Validation Error: Value is not of the correct type
`javax.faces.validator.LengthValidator`	{1}: ValidationError: Value is greater than allowable maximum of {0}
`javax.faces.validator.LengthValidator`	{1}: ValidationError: Value is less than allowable maximum of {0}
`javax.faces.valiadtor.BeanValidator.M`	{0}

Aviso:

Por defecto, los validadores se ejecutarán siempre a no ser que indiquemos lo contrario. Es decir,

que si estoy en una pantalla de registro, le doy al botón CANCELAR y he introducido algún valor no válido, no podré salir de ahí hasta que lo solvente. Como esto se trata de un comportamiento no deseado, ya que queremos salir de ahí sin importarnos la validación, podemos saltárnosla estableciendo a true el valor `immediate` de nuestros `commandButton` o `commandLink`.

5.5.1. JSR 303

JSF 2.0 se integra con el Bean Validation Framework (JSR303), un framework que especifica *validation constraints*. Estos validadores son anotaciones ligadas a atributos de una clase java o a sus getters:

```
public class PagoBean {
   @Size(min=13) private String card;
   @Future public Date getDate() { ... }
   ...
}
```

Las anotaciones que nos ofrece JSR303 son:

Anotación	Atributos	Descripción
`@Null, @NotNull`	Ninguno	Comprueba que un valor sea nulo o no lo sea
`@Min, @Max`	El límite como `long`	Comprueba que un valor es, como máximo o como mínimo, el valor límite descrito. El tipo debe ser `int, long, short, byte`, o a de sus *wrappers* (`BigInteger, BigDecimal, String`)
`@DecimalMin, @DecimalMax`	El límite como `String`	Igual que la anterior, puede aplicarse a un `String`
`@Digits`	`integer, fraction`	Comprueba que un valor tiene, como máximo, el número dado de dígitos enteros o fraccionales. Se aplica a `int, long, short, byte`, o a de sus *wrappers* (`BigInteger, BigDecimal, String`)
`@AssertTrue, @AssertFalse`	Ninguno	Comprueba que un booleano es verdadero o false
`@Past, @Future`	Ninguno	Comprueba que una fecha esté en el pasado o en el futuro
`@Size`	`min, max`	Comprueba que el tamaño de una cadena, array, colección o mapa está en los límites

		definidos
`@Pattern`	`regexp, flags`	Una expresión regular, y sus flags opcionales de compilación

El uso de JSR303 tiene una clara ventaja sobre la validación a nivel de página: supongamos que realizamos una actualización importante en un bean, que es usado en muchas páginas. De esta manera no necesitamos cambiar las reglas de validación más que en la clase para que se aplique a todas las páginas por igual, y así no se nos escapará por error ninguna de ellas.

Para sobreescribir los mensajes de error, hay que crear un fichero ValidationMessages.properties en la raíz del paquete, e introducir allí los mensajes:

```
javax.validation.constraints.Min.message=El valor debe ser como mínimo
{value}
```

Además, para dar un valor específico para un caso concreto, podemos referenciar a la clave del mensaje en la anotación:

```
@Size(min=9, max=9, message="{es.ua.jtech.longitudDni}")
private String dni = "";
```

E introducir ese mensaje en nuestro ValidationMessages.properties

```
es.ua.jtech.longitudDni = El DNI debe tener 9 caracteres
```

Aviso:

Si usamos un servidor de aplicaciones compatible con Java EE 6, tendremos automáticamente acceso a la implementación de JSR 303. En otro caso, deberemos incluir el jar de Hibernate Calidator en el directorio `WEB-INF/lib`

5.5.2. Custom validators

Para implementar nuestro propio validador, tendremos que crear una clase que implemente la interfaz `javax.faces.validator.Validator`, que nos ofrece el método `public void validate(FacesContext context, UIComponent component, Object value)`

Es posible además programar validadores adicionales a los ya existentes en el framework. Estos nuevos validadores definidos deberían ser, en lo posible, reusables para más de un formulario y más de una aplicación. Por ejemplo, podríamos construir un validador que comprobara si una cadena es un código correcto de tarjeta Visa (la implementación de JSF de Apache *MyFaces* lo hace).

Como ejemplo de implementación de nuevos validadores, vamos a definir un validador que sólo permita introducir números pares en nuestra calculadora. No es un ejemplo realista de validación, pero nos sirve para explicar el funcionamiento del framework.

Los pasos para definir un validador propio son los siguientes:

1. Crear una clase que implemente la interfaz `javax.faces.validator.Validator` e implementar en esa clase el método `validate` .
2. Anotarlo con la interfaz `@FacesValidator("validator_Id")`.
3. Usar la etiqueta `<f:validator validatorId="validator_Id" />` en las vistas JSF.

He aquí el código que habría que añadir a la aplicación `calculator` , paso a paso.

1. **Implementamos la interfaz `Validator` .**

   El primer paso es definir una clase que implemente la interfaz `Validator.` y el método `validate` . Para ello creamos el fichero `calculator.validator.PairNumberValidator` :

```
package calculator.validator;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.FacesValidator;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

@FacesValidator("calculator.isPair")
public class PairNumberValidaotr implements Validator{
  public void validate(FacesContext context, UIComponent component, Object
value)
                    throws ValidatorException {
    int number = ((Integer)value).intValue();

    if(number%2 != 0){
      FacesMessage message = new FacesMessage(FacesMessage.SEVERITY_ERROR,
             "No es un número par", "No es un número par");
    }
  }
}
```

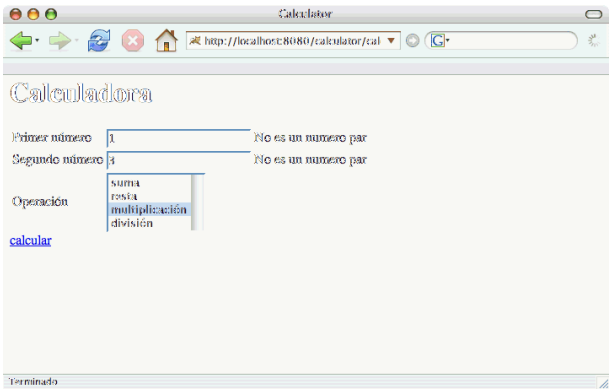
   El método `validate()` recibe el objeto `value` , que en este caso será la conversión a `String` del valor que ha introducido el usuario.

2. Al anotarlo con la interfaz `@FacesValidator`, le hemos asignaod el ID `calculator.isPair`.
3. **Usamos la etiqueta `f:validator` en los ficheros XHTML** .

   Añadimos el siguiente código en los dos `inputText` del fichero `calculator.xhtml` , asociando el validador definido a los dos componentes de entrada.

```
<h:inputText id="firstNumber" value="#{calcBean.firstNumber}"
required="true">
    <f:validator validatorId="calculator.isPair"/>
</h:inputText>
```

El resultado se puede comprobar en la siguiente página HTML generada cuando se introducen números impares en la calculadora.



Página que muestra el mensaje generado por el validador propio.

5.6. Contexto asociado a la petición

Hemos dicho que cada petición procesada por JSF está asociada con un árbol de componentes (llamado también una "vista") que se define en la vista JSF con el mismo nombre que la petición. Cuando se realiza una petición de una vista por primera vez se crea el árbol de componentes asociado. Las peticiones siguientes que se hagan sobre la misma vista recuperán el árbol de componentes ya creado y asociado a la petición anteriormente creada.

Veamos un ejemplo con la aplicación anterior `calculator` . Cuando desde el navegador solicitamos la URI `http://localhost:8080/jsf-calculadora/faces/calculator.xhtml` se accede a la página `calculator.xhtml` .

En el fichero `web.xml` se configura el mapeo de peticiones para que las peticiones que contienen `/faces/` sean procesadas por el servlet `javax.faces.webapp.FacesServlet` .

```
<!-- Faces Servlet -->
   <servlet>
      <servlet-name>Faces Servlet</servlet-name>
      <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
      <load-on-startup> 1 </load-on-startup>
   </servlet>

    <!-- Faces Servlet Mapping -->
   <servlet-mapping>
      <servlet-name>Faces Servlet</servlet-name>
      <url-pattern>/faces/*</url-pattern>
   </servlet-mapping>
```

El servlet analiza la URI de la petición, y decodifica el nombre de la vista a mostrar (`calculator.xhtml`). El identificador de la sesión sirve para recuperar la vista asociada a una petición previa de ese mismo recurso y esa misma sesión, en el caso de que ya se hubiera solicitado esa vista previamente. En este caso, sin embargo, se trata de la primera petición que se realiza en la sesión sobre esta vista. JSF construye entonces el árbol de componentes definido por el fichero `calculator.xhtml` y se guarda en el `FacesContext`

asociado a la petición actual.

También es posible generar una petición desde el cliente cuando se pulsa en algún botón generado por una etiqueta `<h:commandButton>` o se pincha en un enlace resultante de una etiqueta `<h:commandLink` . En nuestra aplicación esto sucede, por ejemplo, cuando pulsamos en el enlace "calcular" de la página principal de la aplicación. En este caso la vista asociada a la petición que JSF recupera es la propia vista desde la que se realiza la petición, ya que es la que corresponde al formulario que debe ser procesado. En la petición se envían los nuevos valores que el usuario ha modificado y la acción solicitada por el usuario ("calcular"). JSF realiza entonces el procesamiento de la petición que veremos más adelante.

Cada petición tiene asociado un contexto, en forma de una instancia de `FacesContext` . Este contexto se usa para almacenar los distintos objetos que necesarios para procesar la petición hasta generar el *render* de la interfaz que se está construyendo. En concreto, gestiona los siguientes aspectos de la petición recibida:

- la cola de mensajes
- el árbol de componentes
- objetos de configuración de la aplicación
- métodos de control del flujo del ciclo de vida

Algunos de los métodos definidos en el objeto `FacesContext` se listan en la siguiente tabla.

Método	Descripción
`addMessage()`	Añade un mensaje a la cola de mensajes de la petición.
`getExternalContext()`	Obtiene el contexto externo (normalmente el contexto del servlet `FacesServlet`) en el que se está procesando la petición.
`getMessages()`	Obtiene un `Iterator` sobre los mensajes que han sido encolados.
`getRenderKit()`	Obtiene la instancia de `RenderKit` especificada para el `UIViewRoot` , si existe.
`getViewRoot()`	Obtiene el `UIViewRoot` asociado a la petición.
`renderResponse()`	Señala a la implementación de JSF que, tan pronto como la fase actual del procesamiento de la petición se haya completado, se debe pasar el control a la fase *Render Response* pasando por alto todas las fases que no se hayan ejecutado todavía.

La cola de mensajes de una petición mantiene un conjunto de mensajes de error que se pueden producir en las distintas fases del ciclo de vida de la misma. El método

addMessage se usa para añadir un nuevo mensaje de error en la cola. Es posible usar el método getMessages() sobre el FacesContext para obtener una colección de todos los mensajes de error asociados a una petición. También es posible mostrar todos los mensajes del FacesContext en el propio componente que se está construyendo mediante la etiqueta <h:messages/> .

El método getViewRoot devuelve el componente UIViewRoot asociado a la petición. Este componente es un tipo especial de componente que representa la raíz del árbol.

Vamos a ver un ejemplo de programación con los distintos elementos del FacesContext usando nuestra aplicación calculator .

En primer lugar, para poder trabajar con el FacesContext hay que obtener la instancia asociada a la petición actual. Para ello basta con llamar al método estático getCurrentInstance() de la clase FacesContext :

```
import javax.faces.context.FacesContext;
...
FacesContext context = FacesContext.getCurrentInstance();
...
```

Podemos hacer esta llamada en el método validate() de un Validator , en el método decode() de un Renderer , en un manejador de una acción o en cualquier otro punto en el que escribamos código que extiende el framework. Una vez obtenido el FacesContext asociado a la petición actual es posible acceder a sus elementos.

Vamos a modificar el método validate que hemos implementado anteriormente para acceder al FacesContext y a uno de sus elementos más importantes: el árbol de componetes. El siguiente código consigue esto.

```
package calculator.validator;
...
import javax.faces.component.UIComponentBase;
import javax.faces.component.UIViewRoot;
import javax.faces.context.FacesContext;

public class PairNumberValidator implements Validator {
  public void validate(FacesContext arg0,
      UIComponent component, Object value)
      throws ValidatorException {
    FacesContext context = FacesContext.getCurrentInstance();
    UIViewRoot viewRoot = context.getViewRoot();
    String ids = getComponentIds(viewRoot);
    FacesMessage message = new FacesMessage("Componentes: "+ ids);
    context.addMessage(null,message);
    ...
  }


  // Obtiene los identificadores y tipos de un componente y de sus hijos.
  // Se llama a si misma de forma recursiva
  private String getComponentIds(UIComponentBase component) {
    String ids = "";
    ids += component.getFamily() + " (" + component.getId() + ") ";
    Iterator it = component.getFacetsAndChildren();
    while (it.hasNext()) {
```

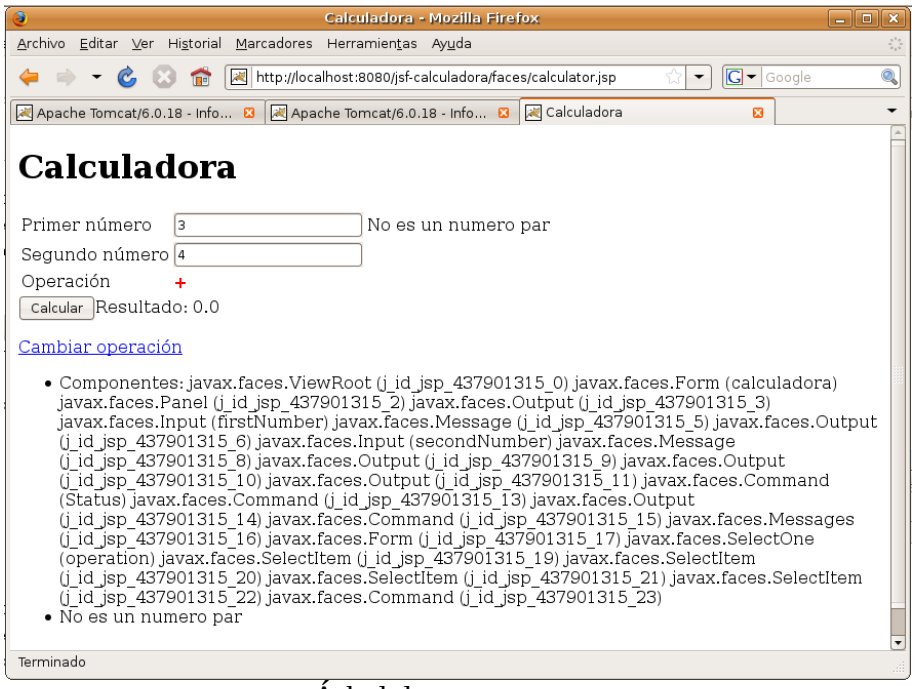
```
      UIComponentBase childComponent = (UIComponentBase) it.next();
      ids += getComponentIds(childComponent);
    }
    return ids;
  }
}
```

Una vez obtenido el `FacesContext` , lo usamos para conseguir el `UIViewRoot` de la petición, el componente raíz del árbol de componentes asociado a la petición JSF. Una vez obtenido, llamamos al método `getComponentIds()` , un método que está implementado más adelante que recorre recursivamente el árbol de componentes y devuelve una cadena con todos los tipos de componente y su identificador.

En el método `getComponentIds()` se llama a `getFacetsAndChildren` , un método del componente que devuelve un iterador con los hijos inmediatos y los `Facets` asociados.

Una vez obtenida la cadena con los tipos e identificadores de los componentes, se añade en la cola de mensajes del contexto de la petición con el método `addMessage()` . Estos mensajes podemos mostrarlos con la etiqueta `<h:messages/>` colocada en la parte inferior de la página.

La siguiente imagen muestra lo que aparece en pantalla. Hay que notar que esta pantalla sólo aparece cuando introducimos un error en la validación del número.



Árbol de componentes

5.7. Árbol de componentes

Recordemos JSF funciona en tres fases: primero genera un árbol de componentes (objetos Java) a partir de la vista JSF asociada a la petición, después activa el ciclo de vida de los componentes en el que se evalúan las expresiones EL y se procesan los eventos generados, y por último se renderizan los componentes resultantes.

En JSF los componentes se organizan en *vistas* . Cuando el framework recibe una petición, se construye una vista con los componentes relacionados con la petición. Una vista es un árbol de componentes, cuya raíz debe ser una instancia de `UIViewRoot` , una clase que no tiene rendering y que sólo sirve como raíz del árbol de componentes. Los componentes en el árbol pueden ser anónimos o pueden tener un identificador de componente proporcionado por el usuario del framework. Los componentes en el árbol pueden ser localizados en base a estos identificadores de componentes, que deben ser únicos en el espacio de nombres definido por los componentes hijos del antecesor más cercano que sea un `NamingContainer` .

¿Qué características tienen los componentes JSF (instancias que forman el árbol de componentes)? En este apartado vamos a repasar algunas de las más importantes.

En primer lugar, todos los componentes JSF extienden la clase abstracta `javax.faces.component.UIComponentBase` , que proporciona la implementación por defecto de los métodos soportados por los componentes JSF.

Cada componente JSF contiene:

- Una lista de componentes hijos.
- Una tabla hash de atributos.
- Uno o más validadores.
- Uno o más manejadores de eventos.
- Un identificador para un renderer opcional.

Todos los componentes JSF son potencialmente contenedores de otros componentes. De esta forma es posible construir componentes compuestos, una característica compartida por la mayoría de frameworks de interfaces de usuario como Swing o Smalltalk.

Los componentes JSF mantienen una lista de atributos almacenados en una tabla hash e indexados por el nombre del atributo. Como valor del atributo es posible insertar cualquier objeto, como una dirección URL, un entero o una imagen.

Todos los componentes realizan tres tareas fundamentales:

- Validar los valores del componente.
- Manejar los eventos del componente.
- Renderizar el componente, normalmente generando código HTML.

Los componentes JSF pueden tener uno o más validadores que validan la entrada. Estos validadores, habitualmente creados por la implementación JSF o por el usuario, se almancenan por componentes en un array list.

La gestión de eventos del componente puede manejarse directamente por un componente o se puede delegar en un manejador de eventos. Se pueden registrar uno o más manejadores de eventos para un componente en la fase *Apply Request Values* del ciclo de vida del componente. Los manejadores son registrados por el renderer del componente o por el componente mismo.

Los componentes JSF pueden renderizarse ellos mismos o delegar el renderizado a un renderer. El método booleano `UIComponent.rendersSelf()` dice a la implementación JSF si un componente se renderiza a si mismo o no. Si no, la implementación JSF obtiene una referencia al renderer del componente con la llamada `UIComponent.getRendererType()` y después llama al renderer para producir el código HTML del componente.

5.8. Ligando componentes a beans gestionados

Mediante el atributo `binding` de una etiqueta es posible ligar un componente con una propiedad de un bean. Después, en cualquier llamada a código de la aplicación, es posible acceder a esa propiedad y consultar o modificar el componente. Por ejemplo, en el siguiente código ligamos el generado por la etiqueta `<h:inputText>` con la propiedad `inputText` del bean `todoController` :

```
<h:panelGroup>
   <h:inputText binding="#{todoController.inputText}"
                size="30"/><br/>
   <h:commandLink value="Añadir projecto"
                actionListener="#{todoController.addNewProject}"
                immediate="true"/>
</h:panelGroup>
...
<h:selectOneMenu id="project" required="true"
  value="#{todo.project}">
  <f:selectItems value="#{todoController.projects}" />
</h:selectOneMenu>
```

Después, en el código del método `addNewProject` del bean podemos acceder al valor introducido por el usuario en el `inputText` , utilizando el objeto `UIInput` que habíamos ligado con la etiqueta `binding` y que JSF ha guardado en la propiedad del bean:

```
public class TodoController {
  ...
  private UIInput inputText;
  ...

  public UIInput getInputText() {
    return inputText;
  }

  public void setInputText(UIInput inputText) {
    this.inputText = inputText;
  }

  public void addNewProject(ActionEvent event) {
    String newProject = (String)inputText.getSubmittedValue();
    inputText.setSubmittedValue(null);
```

```
    projects.add(newProject);
  }
  ...
}
```

Cuando ligamos un componente a una propiedad de un bean, debemos declarar la propiedad de la misma clase que el componente. La siguiente tabla muestra las clases Java de los componentes de cada una de las etiquetas básicas JSF que se transforman en código HTML (de ahí viene el prefijo `h:`):

Etiqueta	Clase Java
<h:column>	**UIColumn**
<h:commandButton>	**UICommand**
<h:commandLink>	**UICommand**
<h:dataTable>	**UIData**
<h:form>	**UIForm**
<h:graphicImage>	**UIGraphic**
<h:inputHidden>	**UIInput**
<h:inputSecret>	**UIInput**
<h:inputText>	**UIInput**
<h:inputTextarea>	**UIInput**
<h:message>	**UIMessage**
<h:messages>	**UIMessages**
<h:outputFormat>	**UIOutput**
<h:outputLabel>	**UIOutput**
<h:outputLink>	**UIOutput**
<h:outputText>	**UIOutput**
<h:panelGrid>	**UIPanel**
<h:panelGroup>	**UIPanel**
<h:selectBooleanCheckbox>	**UISelectBoolean**
<h:selectManyCheckbox>	**UISelectMany**
<h:selectManyListbox>	**UISelectMany**
<h:selectManyMenu>	**UISelectMany**
<h:selectOneListbox>	**UISelectOne**

<h:selectOneMenu>	**UISelectOne**
<h:selectOneRadio>	**UISelectOne**

En el programa ejemplo de la sesión de ejercicios se utilizan algunos de estos componentes.

En todas estas etiquetas se pueden añadir elementos propios del HTML que se trasladarán tal cual cuando se realice el renderizado. Por ejemplo, en todos los elementos es posible asignar un código Javascript a eventos gestionados por el HTML, como `onclick` , `ondblclick` , `onmouseover` , etc. En principio, el código que se introduce en esos atributos no tiene ninguna implicación en el ciclo de vida JSF. Sin embargo, algunas implementaciones de renders HTML incorporan algunas funcionalidades interesantes. Por ejemplo, aunque no es estándar, en la mayoría de implementaciones de JSF, si el Javascript del atributo `onclick` de un `<h:command>` devuelve `false` no se lanza la petición asociada al comando. Así hemos implementado en el ejemplo la típica ventana de diálogo para confirmar un borrado:

```
<h:commandLink value="delete" action="#{todoController.delete}"
   onclick="if (!confirm('¿Seguro que quieres borrar #{todo.title}?'))
return false">
   ...
</h:commandLink>
```

5.9. Gestión de eventos

La gestión de eventos hace referencia a la necesidad de nuestras aplicaciones a responder a acciones del usuario, como pueden ser la selección de ítems de un menú o hacer clic en un botón.

JSF soporta cuatro tipos distintos de eventos:

- Value change events.
- Action events
- Phase events
- System events (a partir de JSF 2.0)

5.9.1. Value Change Events

Los lanzan los "editable value holders" (h:inputText, h:selectOneRadio, h:selectManyMenu,...) cuando cambia el valor del componente.

Es útil porque en muchos casos, los valores / contenidos de un componente dependen de los valores de otro. Un ejemplo clásico son las combinaciones de menús país/provincia. O, como en el siguiente ejemplo, uno que establece el idioma de la aplicación en función del elemento seleccionado

```
<h:selectOneMenu
    value="#{form.country}"
```

```
    onchange="submit()"
    valueChangeListener="#{form.countryChanged}">
  <f:selectItems value="#{form.countries}"
    var="loc"
    itemLabel="#{loc.displayCountry}"
    itemValue="#{loc.country}"/>
</h:selectOneMenu>
```

Aquí, forzamos que el formulario se envíe una vez se ha seleccionado un país, y se lance el método countryChanged del Bean Gestionado Form. Éste cambiará el Locale de la aplicación en función del país

```
public void countryChanged(ValueChangeEvent event) {
  for (Locale loc : countries)
    if (loc.getCountry().equals(event.getNewValue()))
      FacesContext.getCurrentInstance().getViewRoot().setLocale(loc);
}
```

Cabe destacar los métodos del objeto `javax.faces.event.ValueChangeEvent`:

- `UIComponent getComponent()` - Devuelve el components que disparó el evento
- `Object getNewValue()` - Devuelve el nuevo valor del components, una vez ha sido convertido y validado
- `Object getOldValue()` - Devuelve el valor previo del componente

5.9.2. Action events

Los action events son los que lanzan botones y enlaces. Se disparan durante la "Invoke Application phase", cerca del final del ciclo de vida de la aplicación. Se añaden de la siguiente manera:

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

Es importante diferenciar entre actionListener y action. Una acción implica cierta lógica de negocio y participa en la navegación, mientras los actionListeners no participan en la navegación. Normalmente trabajan junto con las acciones cuando una acción necesita realizar ciertas acciones sobre la interfaz de usuario.

JSF siempre invoca a los actionListeners antes que a las acciones.

5.9.3. Los tags f:actionListener y f:valueChangeListener

Éstos tags son análogos a los atributos que acabamos de ver. Por ejemplo, en el caso del menú visto anteriormente, también podríamos presentarlo de la siguiente manera:

```
<h:selectOneMenu value="#{form.country}" onchange="submit()">
  <f:valueChangeListener type="org.expertojee.CountryListener"/>
  <f:selectItems value="#{form.countryNames}"/>
</h:selectOneMenu>
```

Los tags tienen una ventaja sobre los atributos, y es que permiten asociar varios listeners

al mismo componente.

Vemos que hay una diferencia importante: mientras que en el atributo asociábamos un método, en el tag estamos vinculando una clase Java. Ésta debe implementar la interfaz `ValueChangeListener`:

```
public class CountryListener implements ValueChangeListener {
  public void processValueChange(ValueChangeEvent event) {
    FacesContext context = FacesContext.getCurrentInstance();
    if ("ES".equals(event.getNewValue()))
      context.getViewRoot().setLocale(Locale.ES);
    else
      context.getViewRoot().setLocale(Locale.EN);
    }
}
```

Para el caso de los action listeners, deben implementar la interfaz `ActionListener` y se presenta de forma idéntica al caso anterior:

```
<h:commandButton image="logo-experto.jpg" action="#{ation.navigate}">
  <f:actionListener type="org.expertojee.ClickListener"/>
</h:commandButton>
```

5.9.4. Pasando información desde la interfaz al componente: el tag f:setPropertyActionListener

Hasta la especificación 1.2 de JSF podíamos pasar datos de la interfaz a un componente, mediante los tags f:param y f:attribute. Sin embargo, teníamos que "excavar" dentro del componente para obtener esta información.

Con el tag `f:setPropertyActionListener`, conseguimos setear una propiedad en nuestro bean gestionado. Un ejemplo sencillo sería un menú de cambio de idioma:

```
<h:commandLink immediate="true" action="#{localeChanger.changeLocale}">
  <f:setPropertyActionListener target="#{localeChanger.languageCode}"
value="es"/>
  <h:graphicImage library="images" name="es_flag.gif" style="border:
0px"/>
</h:commandLink>

<h:commandLink immediate="true" action="#{localeChanger.changeLocale}">
  <f:setPropertyActionListener target="#{localeChanger.languageCode}"
value="en"/>
  <h:graphicImage library="images" name="en_flag.gif" style="border:
0px"/>
</h:commandLink>
```

En el código anterior, le decimos a JSF que establezca la propiedad languageCode del bean localeChanger a los valores *es* o *en*.

```
public class LocaleChanger {
  private String languageCode;

  public String changeLocale() {
    FacesContext context = FacesContext.getCurrentInstance();
    context.getViewRoot().setLocale(new Locale(languageCode)); return
null;
```

```
  }

  public void setLanguageCode(String newValue) {
    languageCode = newValue;
  }
}
```

5.9.5. Phase Events

Las implementaciones de JSF lanzan eventos antes y después de cada una de las fases del ciclo de vida. Estos eventos son interceptados por los phase listeneners.

Al contrario que los vistos anteriormente, los phase listeners tienen que asociarse a la raíz de la vista, mediante el tag `f:phaseListener`

```
<f:phaseListener type="es.ua.jtech.PhaseTracker"/>
```

Además, podemos declarar phase listeners globales en el fichero `faces-config.xml`.

```
<faces-config>
  <lifecycle>
    <phase-listener>es.ua.jtech.PhaseTracker</phase-listener>
  </lifecycle>
</faces-config>
```

Nuestros phase listeners deberán implementar la interfaz `javax.faces.event.PhaseListener`, que define los siguientes tres métodos:

- `PhaseId getPhaseId()`. Dice a la implementación de JSF en qué fase enviar los eventos al listener. Estas fases pueden ser:
  - `PhaseId.ANY_PHASE`
  - `PhaseId.APPLY_REQUEST_VALUES`
  - `PhaseId.INVOKE_APPLICATION`
  - `PhaseId.PROCESS_VALIDATIONS`
  - `PhaseId.RENDER_RESPONSE`
  - `PhaseId.RESTORE_VIEW`
  - `PhaseId.UPDATE_MODEL_VALUES`
- `void afterPhase(PhaseEvent)`
- `void beforePhase (PhaseEvent)`

Imaginémonos un listener cuyo `getPhaseId()` devolviese `PhaseId.APPLY_REQUEST_VALUES`. En ese caso, los métodos `beforePhase()` y `afterPhase()` se llamarían una vez por cada ejecución del ciclo de vida. Sin embargo, con `PhaseId.ANY_PHASE`, los métodos `beforePhase()` y `afterPhase()` se ejecutarán seis veces por cada ejecución del ciclo de vida.

De manera alternativa, podemos envolver una vista JSF en un tag `f:view` con atributos `beforePhase` y/o `afterPhase`. Estos atributos deben apuntar a métodos del tipo `void listener(javax.faces.event.PhaseEvent)`:

```
<f:view beforePhase="#{backingBean.beforeListener}">
```

```
  ...
</f:view>
```

Los phase listeners constituyen un mecanismo muy útil para el debugging de nuestras aplicaciones.

5.9.6. System Events

A partir de la especificación 2.0 de JSF, se introduce un sistema de notificaciones en el cual tanto la propia implementación como los distintos componentes pueden notificar a distintos listeners acerca de una serie de eventos potencialmente interesantes.

Clase del evento	Descripción	Origen
`PostConstructApplicationE` `PreDestroyApplicationEve`	Inmediatamente después del inicio de la aplicación; inmediatamente antes del apagado de la aplicación	`Application`
`PostAddToViewEvent;` `PreRemoveFromViewEvent`	Después de que un componente haya sido aladido al árbol de la vista; justo antes de que vaya a ser eliminado	`UIComponent`
`PostRestoreStateEvent`	Después de que el estado de un componente haya sido restaurado	`UIComponent`
`PreValidateEvent;` `PostValidateEvent`	Antes y después de que un componente haya sido validado	`UIComponent`
`PreRenderViewEvent`	Antes de que la vista raíz vaya a renderizarse	`UIViewRoot`
`PreRenderComponentEvent`	Antes de que vaya a renderizarse un componente	`UIComponent`
`PostConstructViewMapEven` `PreDestroyViewMapEvent`	Después de que el componente raíz ha construído el mapa de ámbito vista; cuando el mapa de la vista se limpia	`UIViewRoot`
`PostConstructCustomScopeE` `PreDestroyCustomScopeEve`	Tras la construcción de un ámbito de tipo *custom*; justo antes de su destrucción	`ScopeContext`
`ExceptionQueuedEvent`	Después de haber encolado una excepción	`ExceptionQueuedEventContext`

Hay cuatro maneras por las cuales una clase puede recibir `system events`:

La primera de ellas es mediante el tag `f:event`. Ésta constituye la manera más adecuada

para realizar un *listening* de eventos a nivel de componente o vista

```
<h:inputText value="#{...}">
  <f:event name="postValidate" listener="#{bean.method}"/>
</h:inputText>
```

El método debe tener la forma `public void listener(ComponentSystemEvent) throws AbortProcessingException`.

La segunda manera es utilizando una anotación para una clase del tipo `UIComponent` o `Renderer`:

```
@ListenerFor(systemEventClass=PreRenderViewEvent.class)
```

Este mecanismo es muy útil para el desarrollo de componentes, aunque esta materia queda fuera del objetivo de este curso.

En tercer lugar, podemos declararlos en el fichero de configuración `faces-config.xml`.Este mecanismo es últil para instalar un listener para los eventos de la aplicación

```
<application>
  <system-event-listener>
<system-event-listener-class>listenerClass</system-event-listener-class>
    <system-event-class>eventClass</system-event-class>
  </system-event-listener>
</application>
```

Por último, podemos llamar al método `subscribeToEvent` de las clases `UIComponent` o `Application`. Este método está destinado a desarrolladores de *frameworks*, quedando también fuera del objeto del curso.

5.9.7. Usando el tag f:event para validaciones múltiples

JSF no provee de ningún mecanismo para la validación de un grupo de componentes. Por ejemplo, si usamos tres campos distintos para la introducción de una fecha, la única manera de comprobar su validez es, por ejemplo, mediante un evento `PostValidateEvent`.

```
<h:panelGrid id="date" columns="2">
  <f:event type="postValidate" listener="#{bb.validateDate}"/>
  Día: <h:inputText id="day" value="#{bb.day}" size="2" required="true"/>
  Mes: <h:inputText id="month" value="#{bb.month}" size="2"
required="true"/>
  Año: <h:inputText id="year" value="#{bb.year}" size="4"
required="true"/>
</h:panelGrid>
<h:message for="date" styleClass="errorMessage"/>
```

En nuestro listener: obtendremos los valores introducidos por el usuario y verificaremos que se corresponden con una fecha válida. Sino, añadiremos un mensaje de error al componente e invocaremos al método `renderResponse`:

```
public void validateDate(ComponentSystemEvent event) {
```

```
  UIComponent source = event.getComponent();
  UIInput dayInput = (UIInput) source.findComponent("day");
  UIInput monthInput = (UIInput) source.findComponent("month");
  UIInput yearInput = (UIInput) source.findComponent("year");

  int d = ((Integer) dayInput.getLocalValue()).intValue();
  int m = ((Integer) monthInput.getLocalValue()).intValue();
  int y = ((Integer) yearInput.getLocalValue()).intValue();

  if (!isValidDate(d, m, y)) {
    FacesMessage message = es.ua.jtech.util.Messages.getMessage(
      "es.ua.jtech.messages",
      "invalidDate",
      null
    );
    message.setSeverity(FacesMessage.SEVERITY_ERROR);
    FacesContext context = FacesContext.getCurrentInstance();
    context.addMessage(source.getClientId(), message);
    context.renderResponse();
  }
}
```

Otros ejemplos de validaciones de este tipo y que nos encontramos prácticamente a diario podrían ser, por ejemplo, la verificación de la segunda contraseña introducida en el caso de un registro o, verificar que una provincia seleccionada se corresponde con el país seleccionado previamente.

5.9.8. Tomando decisiones antes de renderizar la vista

A veces, tenemos la necesidad de ser notificados antes de renderizar una vista, por ejemplo, para cargar datos, realizar cambios sobre algún componente o incluso navegar a otra vista. Por ejemplo: si nos queremos asegurar de que un usuario está loqueado antes de mostrar una página, podemos envolver la vista en un tag `f:view` y añadir un *listener*:

```
<f:view>
  <f:event type="preRenderView" listener="#{user.checkLogin}"/>
  <h:head>
    <title>...</title>
  </h:head>
  <h:body>
    ...
  </h:body>
</f:view>
```

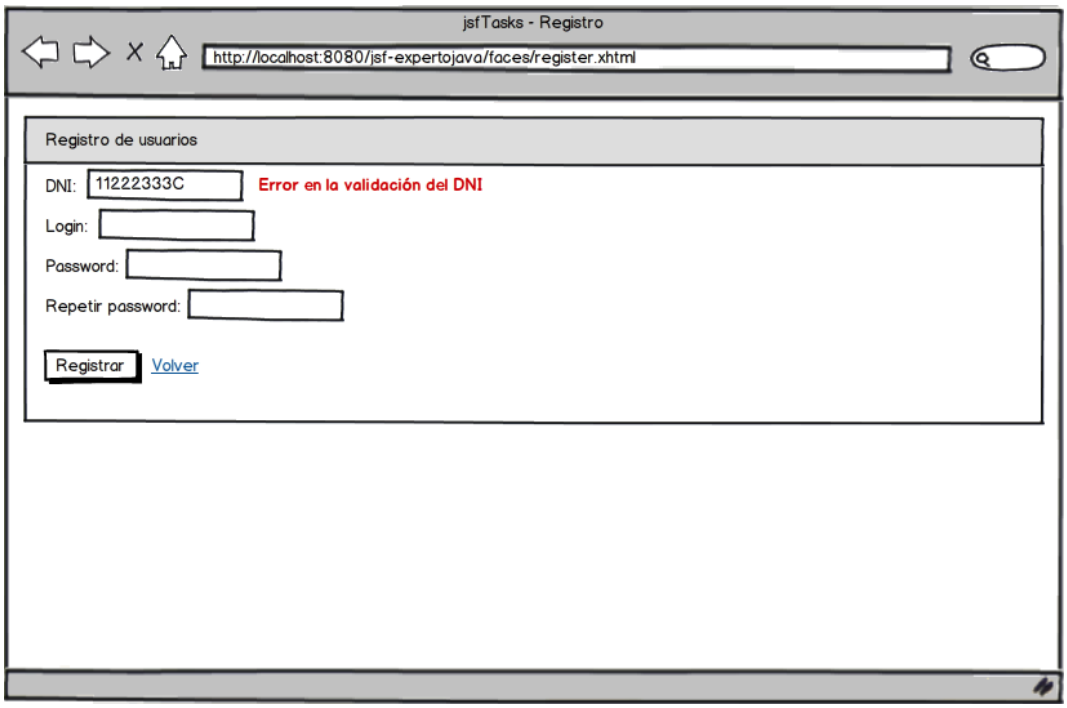
En el listener, verificaremos si el usuario está loqueado. En caso contrario, le redirigiremos a la página de login:

```
public void checkLogin(ComponentSystemEvent event) {
  if (!loggedIn) {
    FacesContext context = FacesContext.getCurrentInstance();
    ConfigurableNavigationHandler handler =
(ConfigurableNavigationHandler)context.
      getApplication().
      getNavigationHandler();

    handler.performNavigation("login");
  }
}
```

6. Ejercicios sesión 3 - Funcionamiento y arquitectura de JSF

6.1. Conversores y validadores

En esta sesión, vamos a aplicar los conocimientos adquiridos en la creación de una página de registro de usuario. Ésta tendrá un aspecto similar al del siguiente mockup:



La vista deberá llamarse `register.xhtml`, y utilizar la misma plantilla que la página de login (0.25 puntos).

Crearemos una clase `es.ua.jtech.jsf.RegisterController`, que tendrá los siguientes atributos (0.25 puntos):

- DniBean dni.
- String login.
- String pass.
- String pass2

Para la clase DniBean, deberemos crear sus conversores y validadores (0.5 puntos)

Los campos del formulario deberán tener las siguientes restricciones, que deberás controlar aplicando las anotaciones de JSR303 en la medida que sea posible (0.5 puntos)

- Todos los campos son obligatorios

- login: longitud mínima, 4 caracteres; longitud máxima: 12 caracteres
- pass: longitud mínima, 6 caracteres; longitud máxima: 12 caracteres

Además, introduciremos dos eventos *postValidate* en nuestra vista (0.75 puntos):

- El primero de ellos se encargará de verificar que los passwords introducidos coinciden
- El segundo, se encargará de verificar que no existe ningún usuario registrado con ese login. Como no tenemos base de datos, con verificar que el login no es *admin* será suficiente

El botón de registro llamará a un método `doRegister` del controlador, que nos "registrará" al usuario y lo dará de alta en sesión, llevándolo a su página de tareas (0.25 puntos).

Por último, nuestra aplicación deberá ser capaz de mostrar cada mensaje de error al lado del campo que lo haya ocasionado (0.5 puntos). Para resaltar un poco más el error, le daremos el siguiente estilo: `color:#B94A48; font-weight: bolder`

7. Internacionalización. Menajes Flash. RichFaces: una librería de componentes profesional

7.1. Mensajes e internacionalización

Cuando creamos una aplicación web, es una buena idea colocar todos los mensajes en una mima localización central. Este proceso hace que sea más facil, entre otras cosas, internacionalizar la aplicación a otros idiomas.

Estos mensajes se guardarán en un fichero `properties`, guardando las cadenas de una forma ya conocida por todos a estas alturas del curso:

```
login_nombre=Nombre de usuario:
login_password=Contraseña:
```

Este fichero se puede guardar donde sea, pero es muy importante que tenga la extensión `.properties` (ej: `src/es/ua/jtech/i18n/messages.properties`)

Para indicar el fichero de recursos que vamos a emplear, podemos hacerlo de dos maneras. La más fácil es indicarlo en el fichero `faces-config.xml`, dentro del directorio `WEB-INF`, de la siguiente manera:

```
<?xml version="1.0"?>
<faces-config 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"
version="2.0">
  <application>
    <resource-bundle>
      <base-name>es.ua.jtech.i18n.messages</base-name>
      <var>msgs</var>
    </resource-bundle>
  </application>
  </faces-config>
```

En lugar de usar una declaración global del fichero de recursos, podemos añadir el elemento `f:loadBundle` a cada página JSF que necesite acceso a los recursos. Ejemplo:

```
<f:loadBundle basename="es.ua.jtech.messages" var="msgs"/>
```

En cualquiera de los casos, los mensajes son accesibles a través de un mapa, con el nombre `msgs`.

A partir de este momento, podremos usar expresiones como `#{msgs.login_nombre}` para acceder a las cadenas de los mensajes.

7.1.1. Mensajes con partes variables

Muchas veces, los mensajes tienen partes variables que no se pueden rellenar a la hora de

declarrar el fichero de recursos. Imaginemos por ejemplo un marcador de puntuación con la frase *Tienes **n** puntos*, donde *n* es un valor devuelto por un bean. En este caso, en nuestro fichero de recursos haremos un String con un *placeholder*:

```
tuPuntuacion=Tienes {0} puntos.
```

Los *placeholders* se numeran partiendo del cero. En nuestra página JSF, usaremos el tag `h:outputFormat`, introduciendo como hijos tantos `f:param` como placeholders haya:

```
<h:outputFormat value="#{msgs.tuPuntuacion}">
   <f:param value="#{marcadorBean.puntos}"/>
</h:outputFormat>
```

El tag `h:outputFormat` usa la clase MessageFormat de la librería estándar para formatear el mensaje.

7.1.2. Añadiendo un segundo idioma

Cuando localizamos un fichero de recursos, debemos añadir un sufijo de *locales* al nombre del fichero: un guión bajo seguido del código de dos letras ISO-639 en minúsculas. Por ejemplo, las cadenas en inglés estarían en el fichero `messages_en.properties` y las cadenas en alemán en el fichero `messages_de.properties`.

Como parte del soporte de internacionalización en Java, el fichero de recursos que coincida con el locale actual se cargará automáticamente. El fichero por defecto sin sufijo de locale se usará como alternativa si no hay un locale específico para un idioma dado

7.1.3. Cambiando de idioma

Ahora que ya sabemos que podemos tener un fichero de recursos por idioma y cómo introducir variables en determinados mensajes, tenemos que decidir cómo establecer el locale de nuestra aplicación. Tenemos tres opciones:

1. Podemos dejar al navegador que escoja el locale. Estableceremos el locale por defecto y los idiomas soportados en el fichero `WEB-INF/faces-config.xml`:

```
<faces-config>
  <application>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>de</supported-locale>
    </locale-config>
  </application>
</faces-config>
```

   Cuando el navegador se conecta a nuestra aplicación, normalemente incluye un valor `Accept-Language` en las cabeceras HTTP. La implementación de JSF lee este header y averigua la que mejor casa de entre la lista de idiomas soportados. Esta característica se puede probar fácilmente si cambias el idioma predeterminado en tu navegador web.

2. También podemos establecer el idioma de manera programada. Esto se consigue llamando al método `setLocale` del objeto`UIViewRoot`.

```
UIViewRoot viewRoot = FacesContext.getCurrentInstance().getViewRoot();
viewRoot.setLocale(new Locale("de"));
```

3. Por último, podemos establecer el idioma de una página determinada si usamos el elemento `f:view` con un atributo `locale`. Por ejemplo:

```
<f:view locale="de">
```

   Podemos convertirlo en algo dinámico si el atributo `locale` se vincula a una variable:

```
<f:view locale="#{langController.lang}">
```

   Esta última opción es muy útil en aquellas aplicaciones en que dejamos al usuario elegir un idioma determinado.

Aviso:

Si optamos por este último método, tendremos que meter toda nuestra página dentro de `f:view`

7.2. Mensajes Flash

Desde JSF 2.0, se ha incluido un objeto *flash*, que puede declararse en una petición y ser usado en la siguiente. Este concepto se ha tomado prestado del *framework* Ruby on Rails. Un uso común que se le da al objeto flash es para el paso de mensajes. Por ejemplo: un controlador puede poner un mensaje en el flash:

```
ExternalContext.getFlash().put("message", "Campo actualizado
correctamente");
```

El método `ExternalContext.getFlash()` devuelve un objeto de la clase `Flash` que implementa un `Map<String, Object>`

En nuestra página JSF, podemos referenciar el objeto flash mediante la variable `flash`. Por ejemplo, podemos mostrar el mensaje de la siguiente manera:

```
#{flash.message}
```

Una vez el mensaje se ha renderizado y la vista ha sido enviada al cliente, la cadena se elimina automáticamente del flash.

Si queremos mantener el valor del flash para más de una petición, podemos hacerlo invocándolo de la siguiente manera:

```
#{flash.keep.message}
```

De esta manera, el valor de mensaje se añadirá nuevamente al flash, pudiendo ser utilizado en la próxima petición.

7.3. Introducción a RichFaces

RichFaces surgió como un framework desarrollado por la compañía Exadel orientado a introducir Ajax en JSF. De hecho, su primer nombre fue Ajax2jsf y ahora algunas etiquetas tienen el prefijo `a4j`. Exadel lanzó el framework en marzo de 2006. A finales de año, el framework se dividió en 2 partes llamados *Rich Faces*, que define los componentes propiamente dichos, y *Ajax4jsf*, que define las etiquetas específicas que dan soporte a Ajax. La primera parte se hizo comercial y la segunda se convirtió en un proyecto opensource en Java.net.

En marzo de 2007, JBoss (ahora parte de Red Hat) y Exadel firmaron un acuerdo por el que ambos frameworks serían distribuidos y utilizados por JBoss y serían gratuitos y opensource. En septiembre de 2007, JBoss y Exadel decidieron unirlos en un único producto llamado RichFaces. Esta decisión ha facilitado el mantenimiento de ambos productos y el lanzamiento de nuevas versiones.

En la actualidad, RichFaces es una parte fundamental del framework Seam propuesto por JBoss. Este framework incluye un *stack* de tecnólogías Java EE formado por JSF, JPA, EJB 3.0 y BPM (Business Pocess Management) además de plug-ins para Eclipse para

facilitar su desarrollo.

Entrando en algo de detalle sobre sus características técnicas, podemos decir que trabaja sobre JSF, ampliando su ciclo de vida y sus funcionalidades de conversión y de validación mediante la introducción de Ajax. Los componentes RichFaces llevan incluido soporte Ajax y un look-and-feel altamente customizable que puede ser fácilmente integrado en la aplicación JSF. Las características más importantes son:

- Integración de Ajax en el ciclo de vida JSF. Mientras que otros frameworks sólo utilizan las funcionalidades de JSF de acceso a los beans gestionados, RichFaces utiliza la gestión de eventos de JSF para integrar Ajax. En concreto, hace posible la invocación de validadores y conversores en el lado del cliente cuando se utiliza Ajax en los manejadores de eventos de acción y eventos de cambio de valor.
- Posibilidad de añadir capacidades de Ajax a las aplicaciones JSF existentes. El framework proporciona dos librerías de componentes: Core Ajax y UI. La librería de Core Ajax permite utilizar las funcionalidades de Ajax en las páginas existentes, de forma que no hay necesidad de escribir nada de código Javascript ni de reemplazar los componentes existentes con los nuevos componentes Ajax. RichFaces permite soporte de Ajax a nivel de página, en lugar del tradicional soporte a nivel de componente y ello hace posible definir los eventos en la página. Un evento invoca una petición Ajax al servidor, se ejecuta esta petición y se cambian ciertos valores del modelo y es posible indicar qué partes de la página (y del árbol de componentes) deben ser renderizados de nuevo.
- Posiblidad de crear vistas complejas creadas a partir de un amplio conjunto de componentes listos para ser utilizados. Se extiende el núcleo de componentes JSF con una amplia librería de componentes con capacidades Ajax que incluyen también soporte para definir skins. Además, los componentes JSF están diseñados para poder ser utilizados con componentes de terceros en la misma página.
- Soporte para la creación de componentes con Ajax incorporado. RichFaces pone a disposición de los desarrolladores el *Component Development Kit* (CDK) utilizado para la creación de los componentes de la librería. El CDK incluye una herramienta de generación de código y una utilidad de templates con una sintaxis similar a JSP.
- Los recursos se empaquetan junto con las clases Java de la aplicación. Además de sus funcionalidades básicas, las facilidades de Ajax de RichFaces proporcionan la posibilidad de gestionar tres tipos de recursos: imágenes, código Javascript y hojas de estilo CSS. El framework de recursos hace posible empaquetar estos recursos en ficheros Jar junto con el código de los componentes de la aplicación.
- Facilidades para generar recursos en caliente. El framework de recursos puede generar imágenes, sonidos, hojas de cálculo Excel, etc. en caliente, a partir de datos proporcionados por el usuario.
- **Skins**. RichFaces proporciona la posibilidad de configurar los componentes en base a skins que agrupan características visuales como fuentes o esquemas de color. Es posible también acceder a los parámetros del skin a partir del código Java y de la configuración de la página. RichFaces proporciona un conjunto de skins predefinidos, pero es posible ampliarlos con skins creados por nosotros.

7.4. Skins

Una de las funcionalidades de RichFaces son los skins. Es posible modificar el aspecto de todos los componentes cambiando una única constante del framework. En concreto, el skin a utilizar en la aplicación se define en el fichero `web.xml` con el parámetro `org.richfaces.SKIN`:

```
<context-param>
   <param-name>org.richfaces.SKIN</param-name>
   <param-value>wine</param-value>
</context-param>
```

Los posibles valores de este parámetro son:

```
DEFAULT
plain
emeraldTown
blueSky
wine
japanCherry
ruby
classic
deepMarine
```

Los skins definen el color y los fuentes tipográficos de los componentes aplicando valores a constantes que se aplican con hojas de estilo a las páginas HTML resultantes. Estas constantes están bastante bien pensadas, y dan coherencia al aspecto de todos los componentes. Por ejemplo, la constante `headerGradientColor` define el color de gradiente de todas las cabeceras y `generalTextColor` define el color por defecto del texto de todos los componentes.

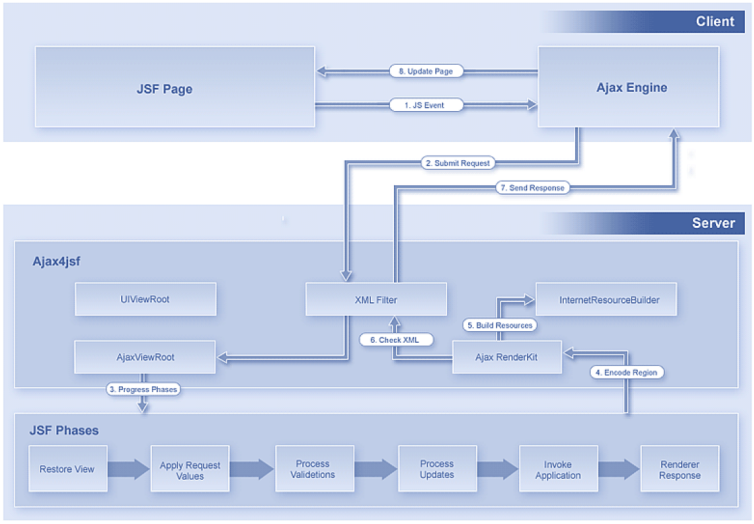
Para más información sobre las características de los skins, cómo utilizarlos y cómo definir nuevos, se puede consultar el apartado sobre skins de la guía de desarrollo (versión 3.0).

7.5. Peticiones Ajax y ciclo de vida

Una de las características fundamentales de RichFaces es la forma de tratar los eventos JavaScript y de incorporarlos en el ciclo de vida JSF.

Con el soporte Ajax de JSF es posible lanzar peticiones Ajax al ciclo de vida JSF desde cualquier componente, no sólo desde los componentes `<command>`. A diferencia de una petición JSF, una petición Ajax hace que el navegador redibuje todos los componentes de la página actual, sino sólo los que nos interesan.

La figura siguiente muestra el ciclo de proceso de una petición Ajax:



1. La página o el componente genera un evento JavaScript que es capturado por el motor de Ajax y por el manejador que definimos en la página.
2. Se envía la petición al servidor, con todos los atributos existentes en la página, como una petición JSF normal.
3. En el servidor se lanzan todas las fases del ciclo de vida JSF (conversión, validación, actualización del modelo y lanzamiento de acciones y el redibujado de la respuesta), pero la respuesta no se devuelve al navegador, sino al motor de Ajax.
4. El motor de Ajax actualiza sólo aquellas partes de la página del navegador que le hemos indicando, modificándolas con las partes obtenidas en la respuesta JSF.

7.5.1. Componentes Ajax

Existen bastantes componentes RichFaces con distintos comportamientos. Los más usados para enviar peticiones Ajax son `<a4j:commandButton>` y `<a4j:commandLink>`, que envían una petición cuando el usuario pulsa un botón (similares a los `h:command`) y `<a4j:support>` que se incluye en otro componente JSF y permite enviar una petición Ajax al servidor cuando sucede un determinado evento Javascript en el componente.

El siguiente código muestra un ejemplo de uso de `a4j:commandButton`:

```
<a4j:commandButton value="Enviar" action="#{bean.doAction}"/>
```

Vemos que es idéntico al componente `h:commandButton`. El atributo `value` define el texto que aparece en el botón. Y el atributo `action` apunta al método `doAction` del bean hace de controlador. La diferencia con el `h:commandButton` es la comentada anteriormente. En una petición estándar JSF la página resultante se devuelve al navegador y éste la vuelve a pintar por completo. En una petición Ajax, el servidor devuelve la página resultante al motor Ajax, y es éste el que decide lo que hay que pintar en la página.

En el apartado siguiente explicaremos cómo decidir qué partes de la página se deben pintar de nuevo con la petición Ajax.

Otro componente Ajax muy usado es `<a4j:support>`. Se puede definir dentro de cualquier componente JSF y permite capturar los eventos Javascript HTML soportados por ese componente y generar una petición Ajax. Por ejemplo:

```
<h:inputText value="#{user.name}">
    <a4j:support event="onkeyup" action="#{bean.doAction}"/>
</h:inputText>
```

En este caso se captura el evento Javascript `onkeyup` del `<h:inputText>` y se envía la petición Ajax al servidor para ejecutar el método `doAction` del bean.

Los eventos soportados por `<a4j:support>` son los posibles eventos javascript del componente padre:

- **`onblur`**: evento generado cuando el elemento pierde el foco.
- **`onchange`**: evento generado cuando el elemento pierdo el foco y se ha modificado su valor.
- **`onclick`**: evento generado cuando se hace un click con el ratón sobre el elemento.
- **`ondblclick`**: evento generado cuando se hace un doble click con el ratón sobre el elemento.
- **`onfocus`**: evento generado cuando el elemento recibe el foco.
- **`onkeydown`**: evento generado cuando se pulsa una tecla sobre el elemento.
- **`onkeypress`**: evento generado cuando se pulsa y se suelta una tecla sobre el elemento.
- **`onkeyup`**: evento generado cuando se suelta una tecla sobre el elemento.
- **`onmousedown`**: evento generado cuando se pulsa el ratón sobre el elemento.
- **`onmousemove`**: evento generado cuando se mueve el ratón dentro del elemento.
- **`onmouseout`**: evento generado cuando se saca el ratón fuera del elemento.
- **`onmouseover`**: evento generado cuando se entra con el ratón dentro del elemento.
- **`onmouseup`**: evento generado cuando el botón del ratón se suelta dentro del elemento.
- **`onselect`**: evento generado cuando se selecciona texto dentro del elemento.

Además de los dos componentes vistos anteriormente, RichFaces proporciona otros componentes que generan eventos Ajax. En la guía de desarrollo de RichFaces se encuentra una explicación detallada de su uso.

- **`<a4j:ajaxListener>`**: se utiliza dentro de otro componente Ajax para declarar manejadores de eventos de JSF igual que `<f:actionListener>` o `<f:valueChangeListener>`.
- **`<a4j:keepAlive>`**: permite mantener el estado de los beans entre distintas peticiones.
- **`<a4j:actionparam>`**: combina la funcionalidad de `<f:param>` y `<f:actionListener>`.
- **`<a4j:form>`**: es muy similar al mismo componente de la librería HTML de JSF, proporcionando además las funcionalidades de generación de enlaces y la posibilidad de petición Ajax por defecto.
- **`<a4j:htmlCommandLink>`**: es muy similar al mismo componente JSF HTML, pero resuelve algunos de sus problemas cuando se usa con Ajax.
- **`<a4j:jsFunction>`**: permite obtener datos del en el servidor en forma de objetos

JSON y lanzar una función Javascript con ellos.

- **`<a4j:include>`**: se usa para incluir código HTML en las páginas.
- **`<a4j:loadBundle>`**: se usa para cargar un conjunto de recursos.
- **`<a4j:log>`**: abre una ventana con la información del evento Javascript producido en el cliente.
- **`<a4j:mediaOutput>`**: permite generar imágenes, vídeo, sonidos y otros recursos en caliente (*on-the-fly*).
- **`<a4j:outputPanel>`**: similar al componente JSF HTML, pero incluyendo facilidades para su uso con Ajax como la posibilidad de insertar elementos no presentes en el árbol de componentes o la posibilidad de guardar el estado de elementos.
- **`<a4j:poll>`**: permite enviar eventos Ajax periódicos al servidor.
- **`<a4j:push>`**: permite enviar peticiones Ajax periódicas al servidor, para simular un comportamiento *push*.
- **`<a4j:region>`**: define el área que es redibujada después de la petición Ajax.

7.5.2. Redibujado de componentes

Por ejemplo, uno de los atributos más importantes es `reRender`, con el que se puede conseguir que JSF sólo renderice una parte de la página actual, la indicada por el valor del atributo. En el siguiente caso, se indica que sólo se debe volver a pintar el bloque marcado con el identificador `miBloque`, correspondiente a un `h:panelGrid`:

```
...
<a4j:commandButton value="Actualizar" reRender="miBloque"/>
...
<h:panelGrid id="miBloque">
...
</h:panelGrid>
...
```

Vemos que en el ejemplo no se define ningún evento de acción, sino que únicamente se envía la petición JSF para que se ejecute el ciclo de vida con la vista actual y se renderice únicamente el `panelGrid` identificado por `miBloque`. Si queremos actualizar más de un componente bastaría con denominarlo con el identificador `miBloque`, o ponerle otra identificador y añadirlo al atributo `reRender`:

```
<a4j:commandButton value="Actualizar" reRender="miBloque1, miBloque2"/>
```

Hay que tener cuidado cuando se utiliza el atributo reRender para redibujar un componente que ya tiene el atributo `rendered`. En el caso en que esa condición devuelva falso, no es posible hacer el redibujado, ya que JSF no incluye el componente en la página resultante y el módulo de Ajax no puede localizarlo para redibujar el DOM. Una forma de solucionar el problema es incluir el componente en un `<a4j:outputPanel layout="none">`. El atributo `ajaxRendered` del `outputPanel` hace que el área de la página sea redibujada, incluso si no se apunta a ella explícitamente, cuando se produzca cualquier petición Ajax. Por ejemplo, el siguiente código redibuja los mensajes de error:

```
...
<a4j:outputPanel ajaxRendered="true">
```

```
   <h:messages />
</a4j:outputPanel>
...
```

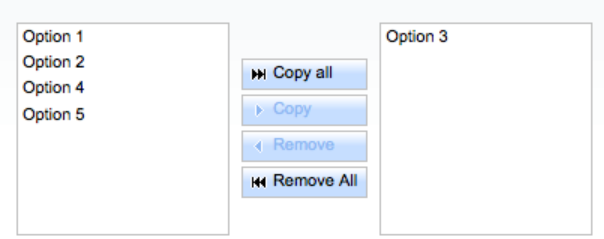
Es posible desactivar este comportamiento con el atributo `limitToList`

```
...
<h:form>
   <h:inputText value="#{person.name}">
      <a4j:support event="onkeyup" reRender="test" limitToList="true"/>
   </h:inputText>
   <h:outputText value="#{person.name}" id="test"/>
</h:form>
...
```

7.6. Algunos ejemplos de componentes

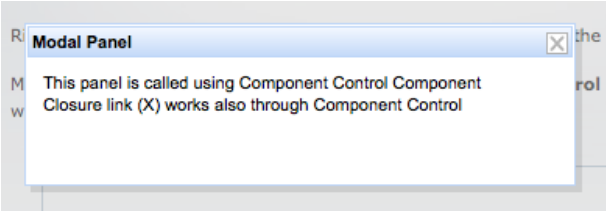
Ejemplos sacados de la demostración RichFaces Showcase. La puedes consultar en http://www.jtech.ua.es/richfaces-showcase

7.6.1. Pick List




```
<rich:pickList>
   <f:selectItem itemLabel="Option 1" itemValue="1"/>
   <f:selectItem itemLabel="Option 2" itemValue="2"/>
   <f:selectItem itemLabel="Option 3" itemValue="3"/>
   <f:selectItem itemLabel="Option 4" itemValue="4"/>
   <f:selectItem itemLabel="Option 5" itemValue="5"/>
</rich:pickList>
```

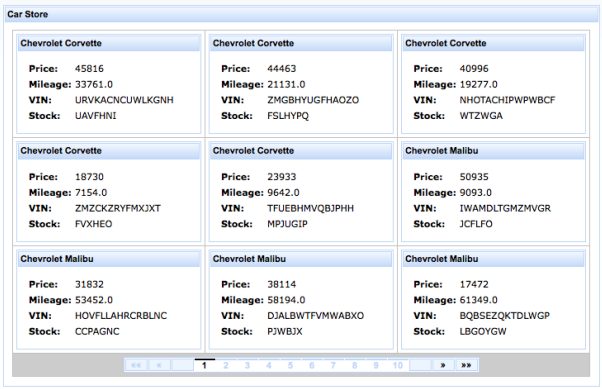
7.6.2. Panel modal




```
<rich:modalPanel id="panel" width="350" height="100">
   <f:facet name="header">
      <h:panelGroup>
         <h:outputText value="Modal Panel"></h:outputText>
```

```
      </h:panelGroup>
   </f:facet>
   <f:facet name="controls">
      <h:panelGroup>
         <h:graphicImage value="/images/modal/close.png"
            styleClass="hidelink" id="hidelink" />
         <rich:componentControl for="panel" attachTo="hidelink"
            operation="hide" event="onclick" />
      </h:panelGroup>
   </f:facet>
   <h:outputText
      value="This panel is called using Component Control Component"/>
   <br />
   <h:outputText
      value="Closure link (X) works also through Component Control"/>
</rich:modalPanel>
<h:outputLink value="#" id="link">
   Show Modal Panel
   <rich:componentControl for="panel" attachTo="link"
                          operation="show" event="onclick" />
</h:outputLink>
```

7.6.3. Data Grid



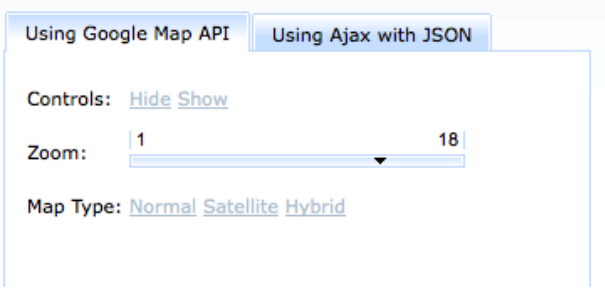

```
<rich:panel>
   <f:facet name="header">
      <h:outputText value="Car Store"></h:outputText>
   </f:facet>
   <h:form>
      <rich:dataGrid value="#{dataTableScrollerBean.allCars}"
         var="car" columns="3" elements="9" width="600px">
         <rich:panel bodyClass="pbody">
            <f:facet name="header">
               <h:outputText value="#{car.make} #{car.model}">
               </h:outputText>
            </f:facet>
            <h:panelGrid columns="2">
               <h:outputText value="Price:" styleClass="label"/>
               <h:outputText value="#{car.price}" />
               <h:outputText value="Mileage:" styleClass="label"/>
               <h:outputText value="#{car.mileage}" />
               <h:outputText value="VIN:" styleClass="label"/>
               <h:outputText value="#{car.vin}" />
               <h:outputText value="Stock:" styleClass="label"/>
               <h:outputText value="#{car.stock}" />
            </h:panelGrid>
```

```
         </rich:panel>
         <f:facet name="footer">
            <rich:datascroller></rich:datascroller>
         </f:facet>
      </rich:dataGrid>
   </h:form>
</rich:panel>
```

7.6.4. Google map


```
<rich:gmap
   gmapVar="map" zoom="16"  style="width:400px;height:400px"
   gmapKey="ABQIAAAA5SzcCLDOLK2VPhx3P-poFxQDT1fstRCWND9TPh4
            hnvi3n3eSLhQH-hQAsES9VPnDb0M9QRvXK83_Lw"
   lat="38.38463"
   lng="-0.51287"/>
```

7.6.5. Tab panel




```
<rich:tabPanel switchType="ajax" width="350" height="400">
   <rich:tab label="Using Google Map API">
      <h:panelGrid columns="2" columnClasses="optionList">
```

```
         <h:outputText value="Controls:" />
         <h:panelGroup>
            <a href="javascript: void 0" onclick=
        map.hideControls();>Hide</a>
            <a href="javascript: void 0" onclick=
        map.showControls();>Show</a>
            <br/>
         </h:panelGroup>

         <h:outputText value="Zoom:" />
         <rich:inputNumberSlider id="zoom" showInput="false"
            minValue="1" maxValue="18" value="#{gmBean.zoom}"
            onchange="map.setZoom(this.value)" />

         <h:outputText value="Map Type:" />
         <h:panelGroup>
            <a href="javascript: void 0"
               onclick=
        map.setMapType(G_NORMAL_MAP);>Normal</a>
            <a href="javascript: void 0"
               onclick=
        map.setMapType(G_SATELLITE_MAP);>Satellite</a>
            <a href="javascript: void 0"
               onclick=
        map.setMapType(G_HYBRID_MAP);>Hybrid</a>
            <br/>
         </h:panelGroup>

      </h:panelGrid>
   </rich:tab>

   <rich:tab label="Using Ajax with JSON">
      <rich:dataGrid var="place" value="#{gmBean.point}" columns="2">
         <h:graphicImage onclick="showPlace('#{place.id}')"
            style="cursor:pointer" value="resource://#{place.pic}" />
      </rich:dataGrid>
   </rich:tab>
</rich:tabPanel>
```

7.7. Referencias

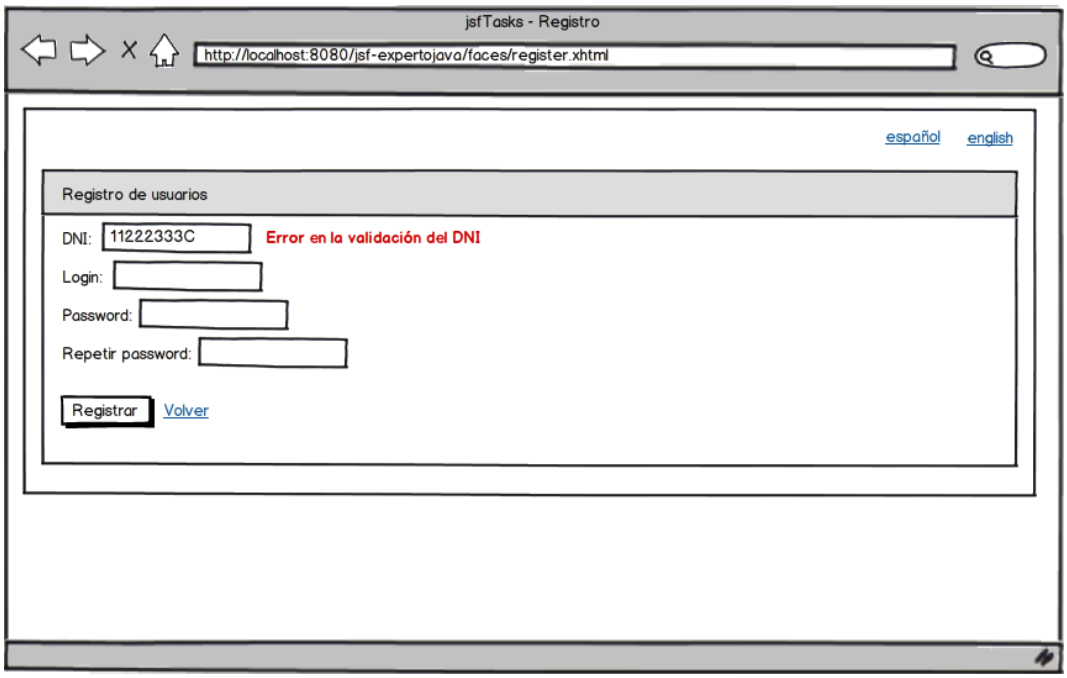
- Página principal de RichFaces en JBoss
- Documentación de RichFaces
- RichFaces, documento sobre RichFaces en el Wiki de JBoss, con bastantes enlaces interesantes.
- Road map de versiones de RichFaces
- Servidor Hudson de RichFaces

8. Ejercicios sesión 4 - Proyecto en RichFaces

El objetivo de esta sesión es introducir en nunuestro proyecyo JSF tradicionaluna serie de componentes de RichFaces. Además, introduciremos opciones para internacionalizar nuestra aplicación.

8.1. Internacionalización de la aplicación (1 punto)

El objetivo consiste en modificar nuestra plantilla inicial para dar la opción de mostrar la aplicación en dos idiomas: Español e Inglés. El idioma soportado por defecto será el Español:



Es posible que en algún momento necesites obtener alguna cadena desde el código Java. Para ello, puedes servirte de la siguiente clase:

```
package es.ua.jtech.jsf.i18n;

import java.text.MessageFormat;
import java.util.ResourceBundle;
import javax.faces.context.FacesContext;

public class Messages {
  public static String getMessage(String id){
    final FacesContext context = FacesContext.getCurrentInstance();
    final ResourceBundle msgs =
context.getApplication().getResourceBundle(context, "msgs");
```

```
    return msgs.getString(id);
  }

  public static String getMessage(String id, Object...args){
    final FacesContext context = FacesContext.getCurrentInstance();
    final ResourceBundle msgs =
context.getApplication().getResourceBundle(context, "msgs");
    final MessageFormat mf = new MessageFormat(msgs.getString(id));

    return mf.format(args);
  }
}
```

Vemos que tenemos sobrecargado el método `getMessage`. Con la segunda opción, se nos permite el uso de *placeholders* en las cadenas. Un sitio donde es interesante usarlo es en la validación del login, para decir si éste se encuentra utilizado (en el properties: `registerController_checkLogin_error=El alias {0} ya existe`)

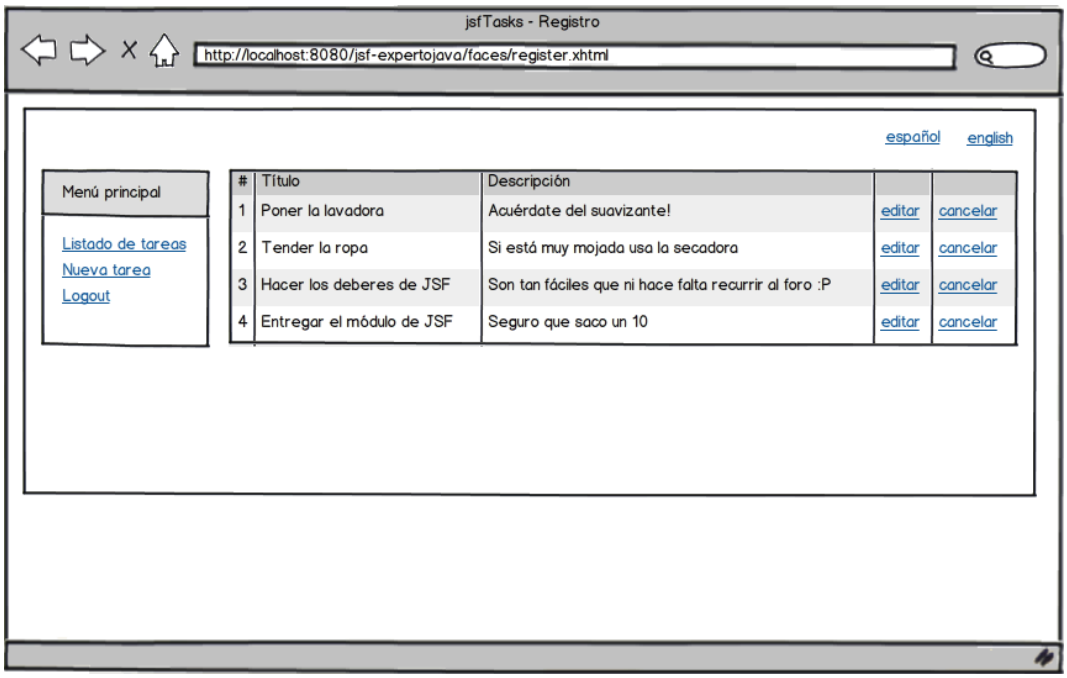
```
if("admin".equals(loginStr)){
    Object[] args = {loginStr};
    final String msg =
Messages.getMessage("registerController_checkLogin_error", args);
    FacesMessage message = new FacesMessage(FacesMessage.SEVERITY_ERROR,
msg, msg);
    FacesContext context = FacesContext.getCurrentInstance();
    context.addMessage(loginInput.getClientId(context), message);
    context.renderResponse();
}
```

8.2. CRUD de tareas (2 puntos)

En esta sesión vamos a realizar el CRUD de tareas.

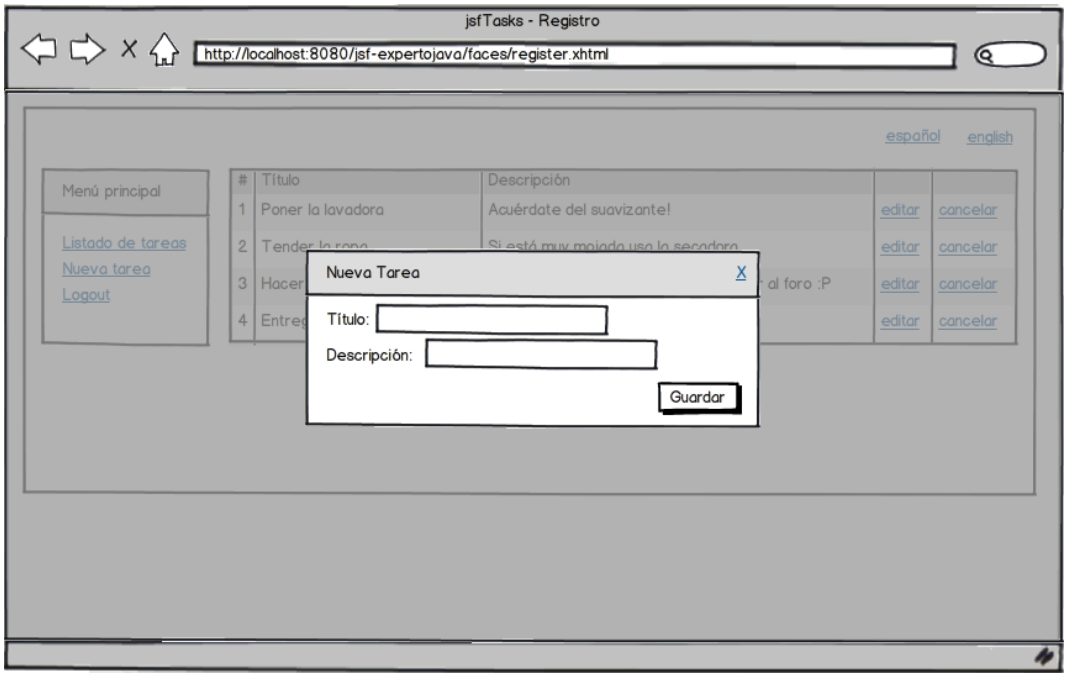
Las pantallas a implementar serán las siguientes:

El listado de tareas tendrá la siguiente forma:

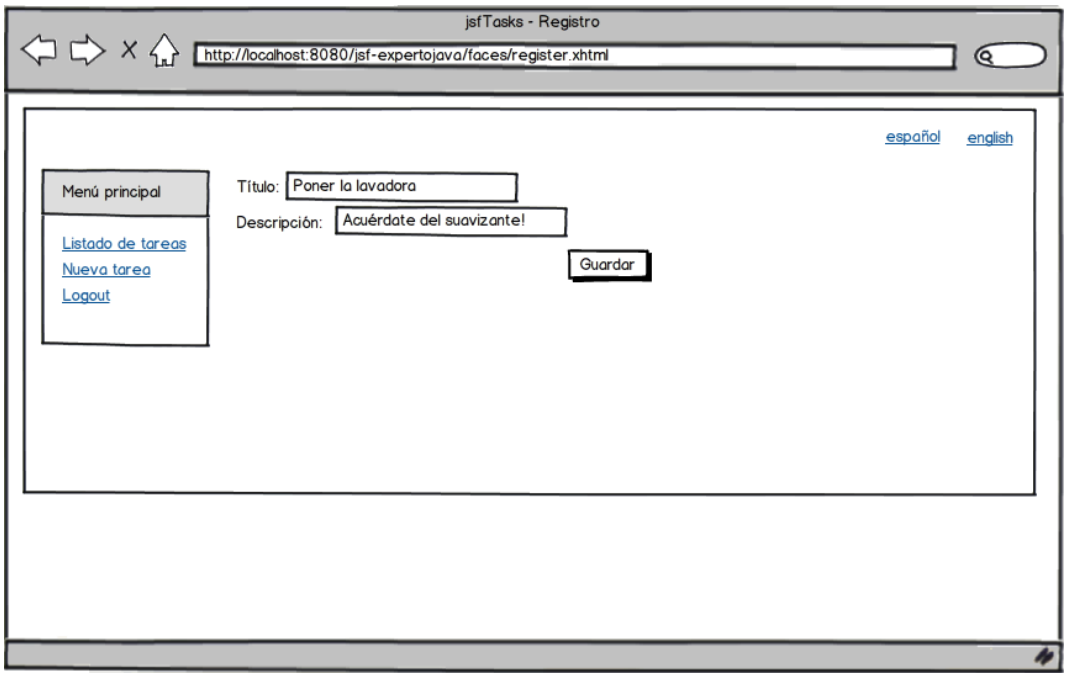


Usaremos un componente `rich:dataTable` para este listado. Esta tabla deberá tener un máximo de 10 entradas y estar paginada con un `rich:dataScroller`.

Para la creación de una nueva tarea, utilizaremos un componente `rich:popUpPanel`.



Para editar la tarea, lo haremos en una página aparte. Ya que el formulario es exactamente igual que el de creación, vamos a intentar reutilizarlo en lugar de hacer uno nuevo:



A la hora de hacer nuestra plantilla de facelets, vamos a crear una plantilla que utilice la base de la plantilla anterior, y que inserte una tabla con dos columnas: una con el menú a la izquierda (fijo y por eso estará en la plantilla), y el contenido a la derecha.

```
<?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">
<ui:composition template="/templates/template.xhtml"
                xmlns="http://www.w3.org/1999/xhtml"
                xmlns:ui="http://java.sun.com/jsf/facelets"
                xmlns:f="http://java.sun.com/jsf/core">
  <ui:define name="title">#{user.name}</ui:define>

  <ui:define name="body">
    <table border="0" cellspacing="0" cellpadding="0">
      <tr>
        <td valign="top">
           <ui:insert name="leftContent">
           <ui:include src="/menu.xhtml" />
           </ui:insert>
         </td>
         <td width="10"></td>
         <td valign="top">
           <ui:insert name="mainContent">
             main content
           </ui:insert>
         </td>
       </tr>
      </table>
    </ui:define>
```

```
</ui:composition>
```

En resumidas cuentas, haremos una plantilla sobre otra plantilla

Para esta parte crearemos un controlador nuevo llamado `TasksController`. Por si os sirve de ayuda, el que se proporcionará en la solución tendrá el siguiente esqueleto:

```
package es.ua.jtech.jsf.controller;

import es.ua.jtech.jsf.model.TaskBean;
import es.ua.jtech.jsf.model.UserBean;
import java.io.Serializable;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.ManagedProperty;
import javax.faces.bean.SessionScoped;

@ManagedBean
@SessionScoped
public class TasksController implements Serializable {
  @ManagedProperty(value="#{user}")
  UserBean user;

  TaskBean task;

  int page=1;

  public String addTask(){
    ...
  }

  public TaskBean getTask() {
    ...
  }

  public void removeTask(){
    ...
  }

  //GETTERS Y SETTERS
}
```