El MVC en JavaServer Faces

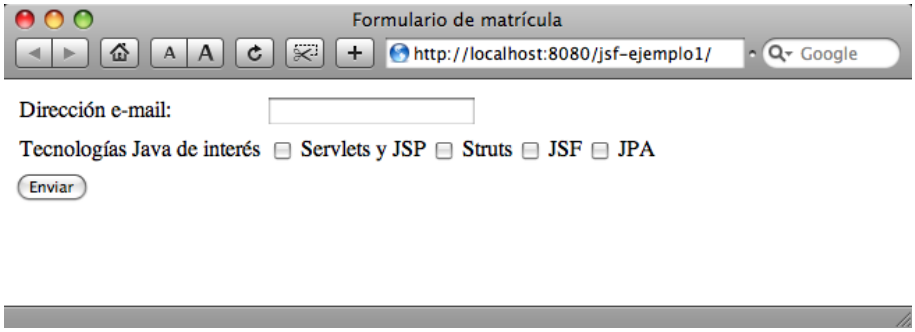
Índice

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.

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.

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;
...
```

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.

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>
```

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.

1.2.3.1. 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;
  }
}
```

1.2.3.2. 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;
  }
}
```

1.2.3.3. 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
```

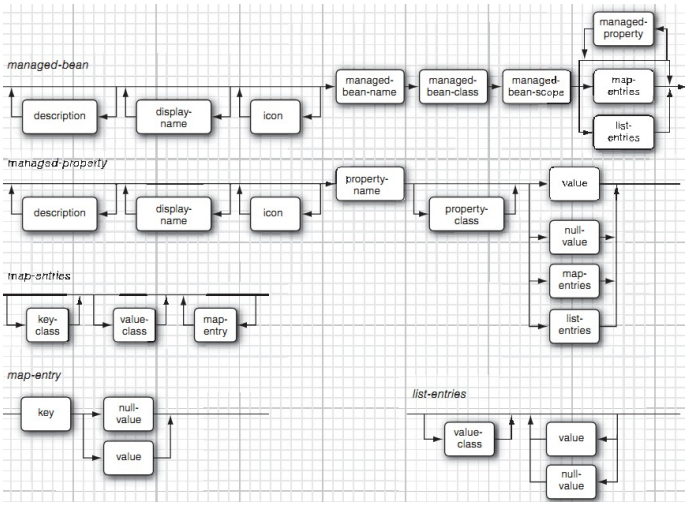
```
    </managed-property>
  </managed-bean>

  <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

1.2.3.4. 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
}
```

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.

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`

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>
```

1.3.2.1. 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

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.

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`.

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}"/>
```

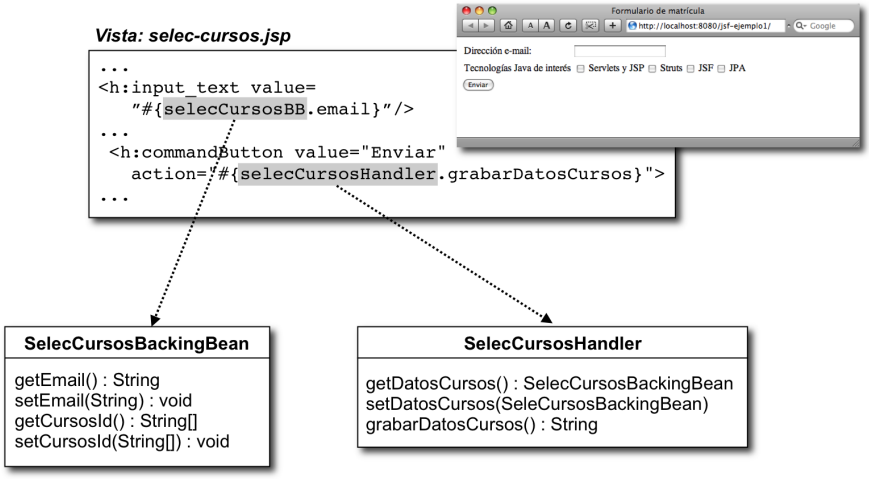
JSF obliga a que todas las acciones devuelvan una cadena. Esa cadena es el valor que termina guardándose en el atributo `action` y será evaluado en las reglas de navegación.

1.5. Resumen de los elementos de JSF

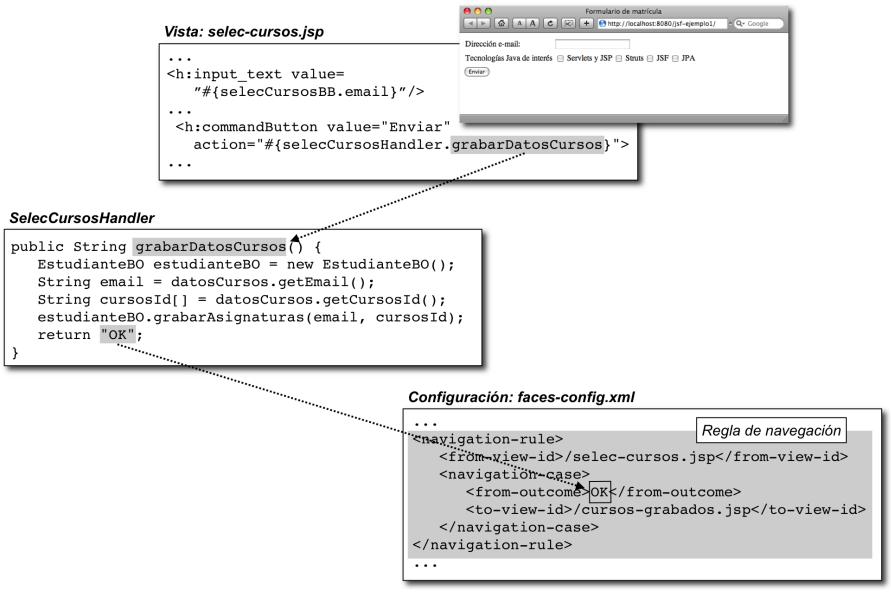
Veamos un resumen rápido de cómo se relacionan los distintos elementos de JSF.

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`.



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. 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

3.1. 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.

3.2. 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. 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>
```

3.4. 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>
```