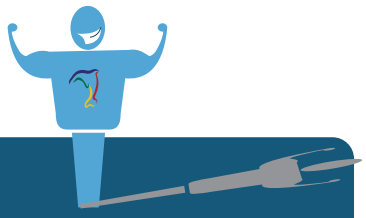
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

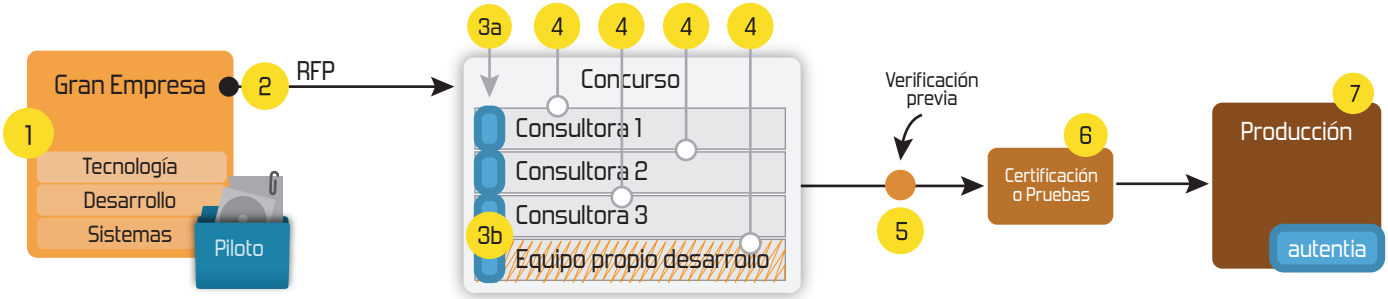
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

- Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery
- Gestor portales (Liferay)
- Gestor de contenidos (Alfresco)
- Aplicaciones híbridas
- Control de autenticación y acceso (Spring Security)
- UDDI
- Web Services
- Rest Services
- Social SSO
- SSO (Cas)
- JPA-Hibernate, MyBatis
- Motor de búsqueda empresarial (Solr)
- ETL (Talend)
- Dirección de Proyectos Informáticos.
- Metodologías ágiles
- Patrones de diseño
- TDD
- Tareas programadas (Quartz)
- Gestor documental (Alfresco)
- Inversión de control (Spring)
- BPM (jBPM o Bonita)
- Generación de informes (JasperReport)
- ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

Jose Manuel Sánchez Suárez

Consultor tecnológico de desarrollo de proyectos informáticos.

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/J2EE

Ver todos los tutoriales del autor

Fecha de publicación del tutorial: 2014-01-15 Tutorial visitado 1 veces Descargar en PDF

Mi primera vista en ZK como desarrollador JSF (II).

0. Índice de contenidos.

1. Introducción

En este segundo tutorial vamos a continuar con ZK, mostrando y recuperando información en la vista con el soporte de los componentes visuales del framework, desde el punto de vista de los conceptos que ya conocemos de JSF.

El entorno sigue siendo el mismo, pero ya hemos instalado el soporte de ZK studio para Eclipse, con el objetivo de disponer de autocompletado en el modo diseño.

2. Mostrar un listado de información paginado en base de datos.

ZK no dispone de un soporte nativo para realizar una paginación en base de datos, así como Primefaces nos proporciona la clase LazyDataModel que permite enganchar los eventos de paginación con búsquedas paginadas en base de datos, con ZK deberíamos hacer lo mismo, creando nuestro propio wrapper sobre Collection.

En una primera aproximación, podemos resolver la problemática recuperando los eventos del componente de paginación de ZK en el controlador y notificando la recarga del listado como sigue:

```
@VariableResolver(org.zkoss.zkplus.spring.DelegatingVariableResolver.class)
public class CustomersView implements Serializable {

    @WireVariable
    private CustomerRepository customerRepository;

    int pageSize = 5;

    int activePage = 0;

    @NotifyChange("customers")
    public void setActivePage(int activePage){
      this.activePage = activePage;
    }

    public long getTotalSize(){
        return customerRepository.count();
    }

    public Integer getPageSize() {
        return pageSize;
    }

    public List<Customer> getCustomers(){
         final Pageable pageable = buildPageRequest(activePage, pageSize,
                    "name", true);
        return customerRepository.findAll(pageable).getContent();
    }

```

```
    private Pageable buildPageRequest(final int page, final int pageSize,
            String sortField, boolean ascending) {
        String sortBy = sortField;
        if (sortBy == null){
            sortBy = "name";
        }
        Direction direction = Direction.DESC;
        if (ascending){
            direction = Direction.ASC;
        }
        final Pageable pageable = new PageRequest(page, pageSize, new Sort(new Order(
                direction, sortBy)));
        return pageable;
    }
}
```

CustomerRepository es un repositorio de spring data que recibe un objeto Pageable; obtenemos una referencia del mismo gracias al soporte de inyección de dependencias de ZK.

El método getCustomers es el que tiene la lógica de recuperación de los registros actuales, inicialmente la página 0 con un tamaño 5. Hablando en términos de JSF los métodos getters no deberían tener lógica de control, pero en el caso de ZK solo se invoca una vez, dentro del ciclo de renderización de la respuesta (a no ser que hagamos más de una referencia al método desde la vista).

La vista tendría un código similar al siguiente:

```
<div self="@define(content)" apply="org.zkoss.bind.BindComposer"
    viewModel="@id('vm') @init('com.autentia.training.masters.views.CustomersView')">

    <listbox id="listbox"
        model="@load(vm.customers)">
        <listhead>
            <listheader hflex="2" label="${labels.name}" sort="auto(name)" />
        </listhead>
        <template name="model">
            <listitem>
                <listcell label="@load(each.name)" />
            </listitem>
        </template>
    </listbox>
    <paging pageSize="@load(vm.pageSize)" totalSize="@load(vm.totalSize)"
        activePage="@save(vm.activePage)" detailed="true" />
</div>
```

El "truco" y lo que nos hará comprender cómo funciona el framework es la vinculación entre los componentes a través de las anotaciones que ya vimos en el primer tutorial; el componente de paginación `paging` tiene las siguientes vinculaciones:

- `pageSize="@load(vm.pageSize)"`: que recupera/carga el valor del tamaño de página invocando al método `getPageSize`,
- `totalSize="@load(vm.totalSize)"`: que recupera/carga el valor del total de los registros invocando al método `getTotalSize`,
- `activePage="@save(vm.activePage)"`: que asigna la página actual invocando al método `setActivePage(int activePage)`

El método `setActivePage` al estar anotado con `@NotifyChange("customers")` provocará que se invoque al método `getCustomers` recargando la parte correspondiente de la página.

Con el atributo detailed nuestra tabla tendrá un aspecto similar al siguiente, con la información sobre paginación en la parte inferior derecha:

3. Presentar y recuperar información en un formulario.

Una vez tenemos nuestro listado paginado, el objetivo ahora es mantener la información del mismo, a modo de CRUD, preparando los eventos de "nuevo" y edición. Lo podíamos haber hecho de muchas formas: tabla editable, formulario en la parte inferior, edición en una nueva página, en una ventana modal... hemos optado por esa última opción, por aquello de complicarlo un poco.

Para incluir una ventana modal con la información de nuestra entidad, hemos añadido un componente `window` con el siguiente código:

```
        <window id="customerWindow" title="${labels.customer}" width="500px" border="nor
            visible="false"
            closable="true"
            onClose="self.visible = false; event.stopPropagation();"
            position="center"
            form="@id('frm') @load(vm.selectedCustomer) @save(vm.selectedCustomer, before
            <vlayout>
                <hlayout>
                    <label value="${labels.name}" />
                    <textbox id="name" value="@bind(frm.name)"/>
                    <label class="error"
                           value="@load(vmsgs[name]) " />
                </hlayout>
            </vlayout>
```

» Mi primera vista en ZK como desarrollador JSF (I).

» ApacheDS: tests de integración contra un servidor LDAP embebido.

» ¿Mockear métodos estáticos?, con el soporte de PowerMock.

» Integración de la gestión de proyecto de Redmine en Eclipse con el soporte de Mylyn.

» Omnifaces: una librería de utilidades para JSF2

```
            <separator style="margin: 5px;" />
            <hlayout>
                <button label="${labels.save}" onClick="@command('save', window=customerW
                <button label="${labels.delete}" onClick="@command('delete', window=custo
                    visible="@load(vm.selectedCustomer.uniqueId ne null)" />
            </hlayout>
        </window>
    </div>
</zk>
```

Lo más importante es el atributo `form` del componente `window` en el que asignamos un id "frm" a la carga del cliente seleccionado con `@load(vm.selectedCustomer)` y además indicamos que queremos guardar en el mismo objeto la información del formulario antes de guardar `@save(vm.selectedCustomer, before='save')`. Lo que hará ZK es crear un objeto intermedio en el que se irá asociando la información de las propiedades de la entidad y antes de guardar (before='save'), cargará (@save) la información de ese objeto en la entidad (vm.selectedCustomer).

Incluimos un componente de formulario `textbox` asignando el valor de una propiedad de nuestra entidad `value="@bind(frm.name)"`

Marcamos dos eventos, para guardar y borrar, condicionando la visibilidad del botón de este último a la existencia de un identificador en la entidad `visible="@load(vm.selectedCustomer.uniqueId ne null)"`. En la invocación al evento además, vamos a enviar un párametro a los métodos, la instancia del componente de ventana `window`; esta es la manera de invocar a métodos con parámetros en zk `@command('save', window=customerWindow)`, ahora veremos por qué lo necesitamos.

En el controlador recibiremos el evento de guardar con el siguiente método:

```
@NotifyChange("customers")
@Command("save")
public void saveSelectedCustomer(@BindingParam("window")  Window window){
    final String msgKey = (selectedCustomer.getUniqueId() != null) ? "actions.save.ok" : "
    try {
        customerRepository.save(selectedCustomer);
        Messagebox.show(
                Labels.getLabel(msgKey),"", Messagebox.OK, Messagebox.INFORMATION
            );
        window.setVisible(false);
    }
    catch(Exception e){
        Messagebox.show(
                e.getMessage(), Labels.getLabel("generic.error_title"),
                Messagebox.OK, Messagebox.ERROR
            );
    }
}
```

Tenemos que comentar lo siguiente:

- notificamos el evento de cambio al método getCustomers con la anotación `@NotifyChange("customers")`, para que al guardar se repinte; sería como un rerender o update de JSF/Primefaces.
- con la anotación `@BindingParam("window")` recibimos la instancia de la ventana en cliente y, con ello, si todo va bien podremos cerrarla `window.setVisible(false);`.
- ZK no tiene un componente de mensajes ni el concepto de mensajes hacia el cliente como tal; si existen mensajes de error en las validaciones aunque para mostrarlos todos en forma de tabla solo existe un componente en la versión EE. Tampoco existe un componente de growl, aunque podríamos añadir uno propio o utilizar un componente de terceros que extienda los de ZK. Como solo queremos hacer uso de los componentes nativos de ZK, para notificar mensajes al cliente estamos haciendo uso del componente `Messagebox`; este componente solo tiene api en el lado servidor.

Ahora vamos con el evento de borrado que tiene una complejidad adicional, queremos solicitar la confirmación de borrado al cliente:

```
@SuppressWarnings("unchecked")
@Command("delete")
public void deleteSelectedCustomer(@BindingParam("window") final  Window window){
    Messagebox.show(
            Labels.getLabel("actions.delete.confirm"), Labels.getLabel("actions.delete.co
            Messagebox.OK | Messagebox.CANCEL,
            Messagebox.QUESTION,
            new EventListener() {
                @Override
                public void onEvent(Event evt) throws InterruptedException {
                    if (evt.getName().equals("onOK")) {
                        try {
                            customerRepository.delete(selectedCustomer);
                            window.setVisible(false);
                            BindUtils.postNotifyChange(null,null, CustomersView.this,"cust
                            Messagebox.show(
                                    Labels.getLabel("actions.delete.ok"),"",
                                    Messagebox.OK, Messagebox.INFORMATION
                                );
                        }
                        catch(Exception e){
                            Messagebox.show(
                                    e.getMessage(), Labels.getLabel("generic.error_title"
                                    Messagebox.OK, Messagebox.ERROR
                                );
                        }
                    }
                }
            }
        );
}
```

Además de lo comentado en el método anterior, tenemos que añadir lo siguiente:

- estamos usando un primer componente de `Messagebox` para pedir confirmación, capturando un evento sobre el botón `OK`
- en la captura de ese evento realizamos la lógica de control y con la invocación al método estático `BindUtils.postNotifyChange(null,null, CustomersView.this,"customers")` marcamos el rerender del listado para que se actualice al borrar; no podemos usar la anotación `@NotifyChange("customers")` porque se actualizaría al invocar al método y no cuando se pulsa sobre la confirmación.

Para obtener el registro seleccionado solo tenemos que añadir un atributo al listbox apuntando a la propiedad del

controlador y para mostrar la ventana jugamos con el evento `onSelect`; al crear la ventana no hay una forma de indicar que queremos que sea modal, por eso hay que invocar a ambos métodos.

```
<listbox id="listbox" selectedItem="@bind(vm.selectedCustomer)" onSelect="customerWindow.
...
```

y en el controlador, para mantener la instancia del registro a mantener:

```
@VariableResolver(org.zkoss.zkplus.spring.DelegatingVariableResolver.class)
public class CustomersView implements Serializable {

    private Customer selectedCustomer;

    public Customer getSelectedCustomer() {
        return selectedCustomer;
    }

    public void setSelectedCustomer(Customer selectedCustomer) {
        this.selectedCustomer = selectedCustomer;
    }

    ...

}
```

Por último, el botón que maneja el evento de "nuevo" en la vista tendría el siguiente código:

```
<button label="${labels.new}"
            onClick="@command('initialize', window=customerWindow)"/>
```

y en el controlador:

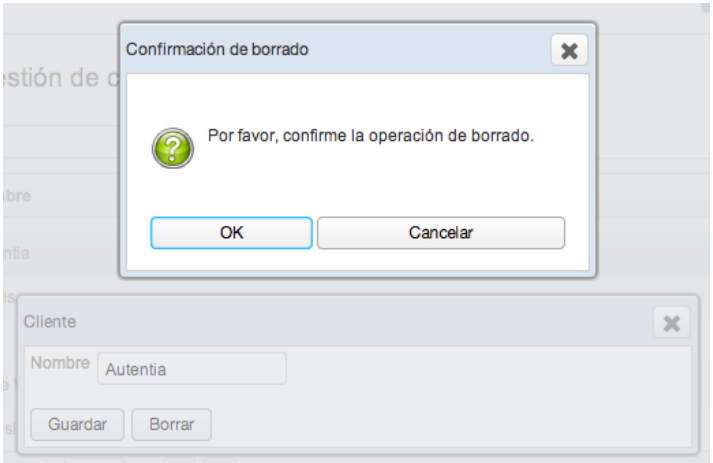
```
@Command("initialize")
@NotifyChange("selectedCustomer")
public void initializeSelectedCustomer(@BindingParam("window")  Window window){
    selectedCustomer = new Customer();
    window.doPopup();
    window.doHighlighted();
}
```

Es lo mismo que hacemos al seleccionar un registro del listado, pero en el evento del servidor.

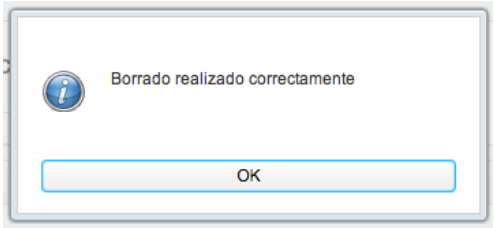
El resultado lo tenemos en las siguientes capturas, para el alta:

En una modificación:

La confirmación del borrado:



El mensaje hacia el cliente:



3.1. Validaciones.

Existen dos tipos de validaciones en ZK:

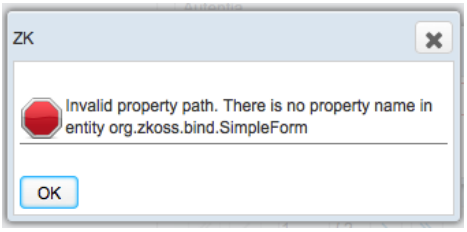
- constraints: asociadas al atributo con el mismo nombre de los componentes de formulario y que admite una serie de validaciones predefinidas junto al uso de expresiones regulares:

```
<label value="${labels.name}" />
<textbox id="name" value="@bind(frm.name)"
  constraint="no empty"/>
<label class="error"
  value="@load(vmsgs[name]) " />
```

  Estas constraints son como los componentes de validación de JSF, que anidamos a los componentes de formulario.

- custom constraints: que permiten, también como en JSF, disponer de un validador propio para poder asociarlo a un componente de formulario.

Existe una tercera posibilidad, la de enganchar las validaciones del framework con la especificación de bean validator, de modo que podríamos incluir las anotaciones correspondientes a nivel de entidad y las validaciones se realizarían dentro del ciclo de asignación de valores que configuremos. La mala noticia es que esa posibilidad solo la tenemos si nos encontramos en la versión EE. Sin las dependencias de la versión EE, tendremos el siguiente error:



Para añadir tanto una validación propia como la validación de beanValidator distibuida en la versión EE, podríamos añadirla a nivel de value en el campo de formulario:

```
<textbox id="name" value="@bind(frm.name) @validator('beanValidator')"
```

Desde el punto de vista de JSF:

- en ambos todas las validaciones se realizan en servidor,
- con JSF el soporte de bean validator lo tenemos en la propia especificación.

3.2. Conversiones.

El concepto de conversión de tipos de datos es similar a JSF pero aunque existe el concepto de conversor a nivel de componente, lo que proporciona el framework son diferentes componentes visuales para cada tipo de dato:

```
<textbox id="name" value="2000.02" />
<doublebox format="#,###.00"  value="2000.02" />
<decimalbox format="#,###.00" value="2000.02"/>
```

Conversores predefinidos, como tales, solo existen dos, para tipo fecha y numérico:

```
<label value="@load(item.price) @converter('formatedNumber', format='###,##0.00')"/>
<label value="@load(item.creationDate) @converter('formatedDate', format='yyyy/MM/dd')"/>
```

y basados en la anotación @converter podemos definir los nuestros implementando la interfaz `Converter`, solo que en vez de `convertAsString` o `convertAsObject` como lo haríamos en JSF, los métodos se llaman `coerceToUi` y `coerceToBean`.

```
public class MyConverter implements Converter {

    public Object coerceToUi(Object val, Component comp, BindContext ctx) {
        ...
    }

    public Object coerceToBean(Object val, Component comp, BindContext ctx) {
        try {
            ...
        } catch (ParseException e) {
            throw UiException.Aide.wrap(e);
        }
    }
}
```

Para hacer referencia al mismo:

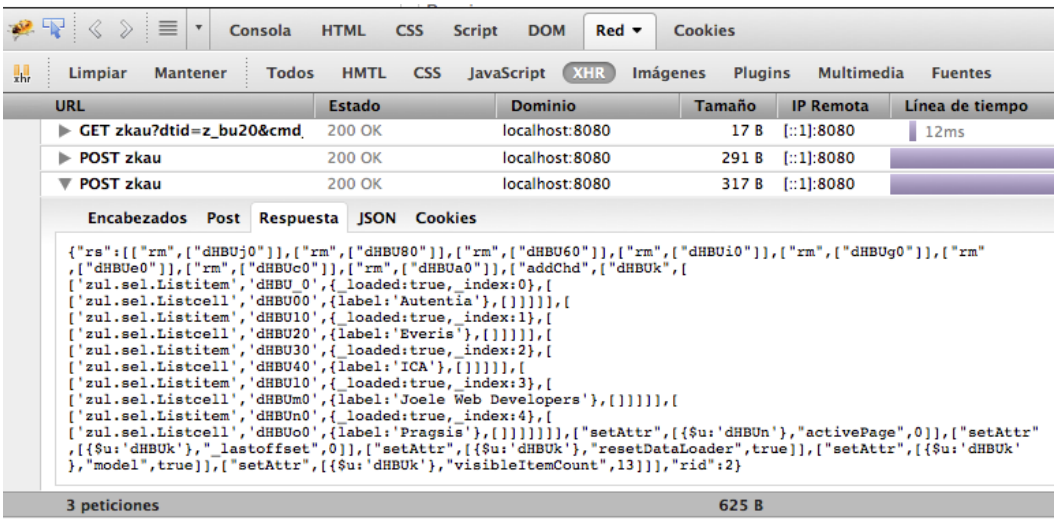
```
<label value="@load(vm.message) @converter(vm.myConverter)"/><!-- manteniendo una instanc
<label value="@load(vm.message) @converter('com.foo.MyConverter')"/>
```

En la versión EE se pueden registrar conversores a nivel de aplicación para evitar tener que hacer referencia a los mismos con el paquete y nombre de la clase o incluirlos en el controlador de referencia.

3.3. Transporte.

Llegados a este punto vamos a hablar sobre cómo se realiza el transporte o la comunicación entre el cliente y el servidor usando ZK, con la aproximación MVVM que hemos elegido y la vinculación de propiedades en el formulario, Form Binding, que hemos configurado.

Para responder solo tenemos que usar firebug:



Todo es Ajax/JSON, las peticiones se realizan por POST y siempre que el valor de un campo en el formulario cambia (onChange) se produce una petición en segundo plano al servidor para asignarlo a la propiedad correspondiente del objeto intermedio.

Comparándolo con JSF y su soporte para Ajax, con lo que llevamos visto, el resultado es cómo:

- lo que era el partialSubmit de ICEfaces,
- si activásemos el ajaxSingle de Richfaces, o
- configurásemos el soporte nativo de Ajax en JSF2 para todos los campos del formulario.

4. Referencias.

- https://www.google.es/#q=www.adictosaltrabajo.con+zk
- http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=zk_install_zkstudio_eclipse
- http://architects.dzone.com/articles/serverside-pagination-zk
- http://java.dzone.com/articles/zk-gritter-growl-notifications
- http://books.zkoss.org/wiki/ZK_Component_Reference/Base_Components/InputElement#Custom_Constraint
- http://books.zkoss.org/wiki/ZK_Developer%27s_Reference/MVVM/Data_Binding/Validator

5. Conclusiones.

Seguimos viendo que los conceptos son similares a los de JSF, como lo podrían ser los de cualquier framework de presentación. Por otro lado, ya hemos visto alguna limitación de la versión community

Continuamos allanando el camino para hacer una comparativa entre ZK y Primefaces, ya veremos si merece la pena y lo más probable, una prueba de rendimiento entre ambos.

Un saludo.

Jose

jmsanchez@autentia.com

A continuación puedes evaluarlo:

Regístrate para evaluarlo

Por favor, vota +1 o compártelo si te pareció interesante

Share |

Anímate y coméntanos lo que pienses sobre este **TUTORIAL:**

» Registrate y accede a esta y otras ventajas **«**