



Manual esencial Dr4zk
Versión 0.9

Copyright 2011-2013
Jorge Luis Martínez Ramírez,
Josué Reyes Ramírez

Esta obra está bajo una [Licencia
Creative Commons Atribución-
CompartirIgual 3.0 Unported](https://creativecommons.org/licenses/by-sa/3.0/).

¿QUÉ ES DR4ZK?	3
ARQUITECTURA	3
COMENZANDO CON DR4ZKDEMO	4
SEARCH CATEGORY	5
ADD, EDIT CATEGORY	8
READ CATEGORY	11
SEARCH COMPANY	13
ADD, EDIT COMPANY	15
READ COMPANY	18
ACCIONES	21
DRACTIONS	22
DRFELLOWLINK	22
MAPEO DE ATRIBUTOS CLASES DE NEGOCIO Y CLASES DEL MODELO DE VISTA	25
DRROOTENTITY	25
CONTENEDORES	27
VALIDACIONES	27
VALIDADOR PERSONALIZADO	29
COMPONENTES	30
DRTEXTBOX	30
DRATTACHLIST	31
DRCAPTCHA	32
DRCOMBOBOX	32
DRDATEBOX	33
DRDECIMALBOX	33
DRFCKEDITOR	34
DRGMAPS	34
DRHTML	35
DRIMAGE	35
DRINTBOX	36

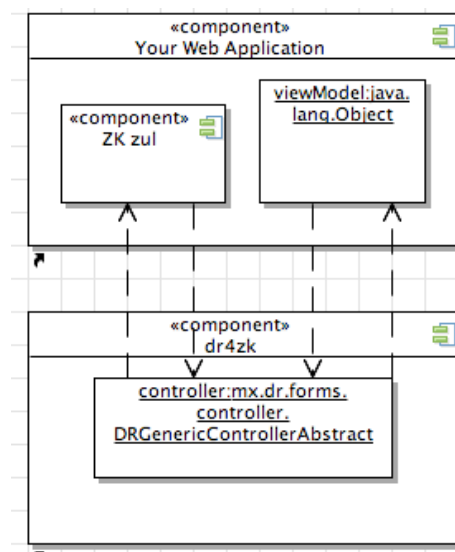
DRLABEL.....36
DRSPINNER.....36
DRLISTBOX36
GLOSARIO38

¿Qué es DR4ZK?

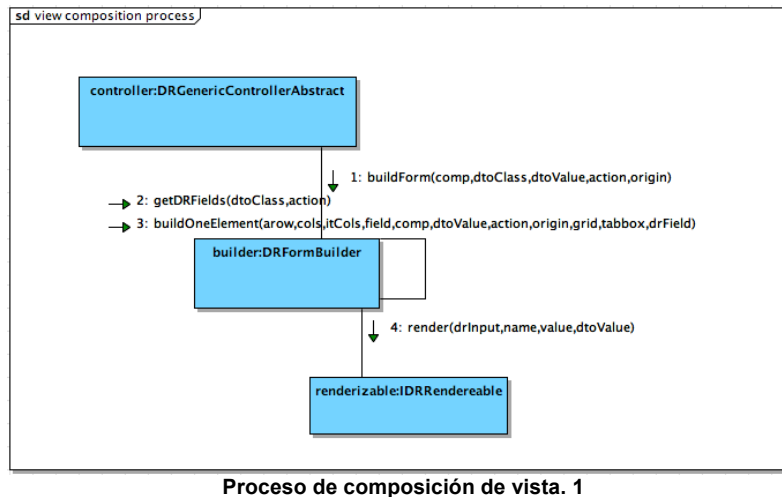
Dr4zk nace de la necesidad de tener una opción de crear y manejar la vista web de una forma más rápida y sencilla desde la misma definición del modelo de los componentes de una vista. Es para aquellos desarrolladores que prefieren agilizar las actividades de diseño y ahorrar tiempo para enfocarse en desarrollar el proceso del negocio. La base del funcionamiento de *dr4zk* es el framework *ZK* el cual permite modelar y manejar la vista web de una forma dinámica y programática, dejando de lado el manejo los correspondientes tags en un archivo *zul*.

Arquitectura

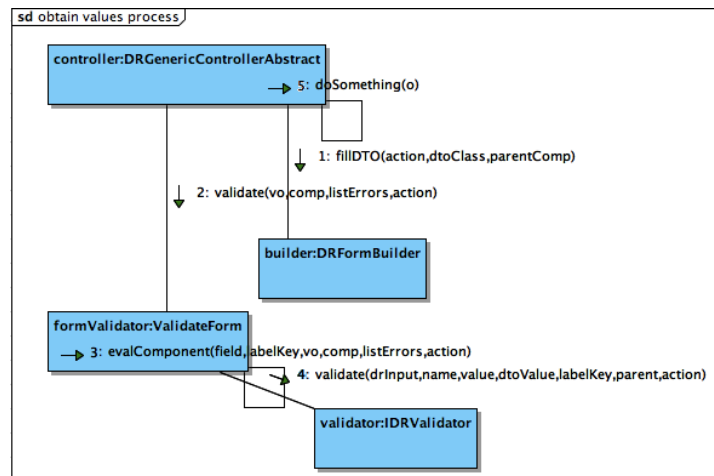
Dr4zk implementa para su funcionamiento el patrón *MVC*, donde la vista es el archivo *zul* manejado por *ZK*; el modelo de vista es una clase *pojo*, la cual se usa para mapear los valores que se obtienen de los componentes que se dibujan en pantalla; y finalmente el controlador que se trata de una clase que extiende de *DRGenericControllerAbstract* donde se procesan los eventos solicitados por el usuario. El siguiente diagrama ilustra lo anterior:



Cuando un usuario solicita una url desde su navegador y esta vista es manejada por *dr4zk*, *dr4zk* leerá el nombre de la clase de modelo de vista que contiene la configuración de componentes que se pintaran en pantalla, obtendrá los controles aplicables al tipo de acción solicitada (ADD, EDIT, READ, SEARCH) y pintará cada uno de ellos en el lugar que le corresponde. La siguiente ilustración muestra el diagrama de comunicación de este proceso:



Cuando el usuario introduce valores a una forma manejada por *dr4zk* ejecutando el método `generalAction()` en el caso (ADD, EDIT); o `search()` en el caso de la acción (SEARCH), los valores de los componentes de la forma serán mapeados a una instancia de la clase que define el modelo de vista y serán validados de acuerdo a las validaciones que el programador haya anotado en el modelo de vista. El diagrama de comunicación de este proceso es el mostrado a continuación:



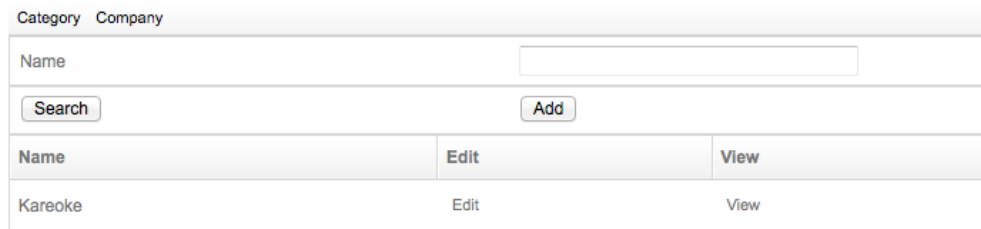
Comenzando con dr4zkdemo

Dr4zkdemo es una aplicación web que demuestra el funcionamiento general de *dr4zk* y que servirá de guía para el desarrollador. Este demo puede ser obtenido desde su código fuente mediante el enlace <https://github.com/jorgemfk/dr4zkdemo> o bien, puede descargarse empaquetado en war con código fuente en <http://www.dr4zk.org> (recomendado). Se trata de una arquitectura

multicapa común teniendo en la capa de presentación *dr4zk* y *ZK*, en la capa de lógica de negocio *Spring* y *Hibernate*, en la capa de persistencia.

SEARCH Category

El proyecto *dr4zkdemo* consta de dos módulos que ejemplifican el uso y ventajas de usar *dr4zk* en una aplicación web. Su objetivo es simular la necesidad de tener un directorio de empresas que se agrupan en distintas categorías. El primer módulo es *categoría* el cual administra las categorías a las que pertenecen las distintas empresas registradas. Este módulo consta de las siguientes acciones: ADD, EDIT, SEARCH y READ. Cuando elegimos la opción del menú Category lo que nos muestra es una pantalla donde podemos buscar nuestras categorías, editar, consultar alguna existente o agregar una nueva.



Name	Edit	View
Karaoke	Edit	View

Estudiemos el archivo *index.zul*, el cual nos direcciona a la página anteriormente mostrada. Notemos que al seleccionar la opción estamos haciendo que se incluya una página en una url que esta parametrizada por dos parámetros como sigue:

```
<toolbarbutton label="Category">
  <attribute name="onClick">
    <![CDATA[
myInclude.setSrc("/genericFind.zul?dto_class=mx.test.dr4zkdemo.view.model.RegisterCategory&action=SEARCH");
]]>
    </attribute>
  </toolbarbutton>
```

En estos parámetros `action` se refiere a la acción que se usará de nuestro modelo de vista (SEARCH para este caso) y que será tratada bajo esta acción por la clase controladora de *dr4zk*. El parámetro `dto_class` define la clase que detalla nuestro modelo de vista es decir los componentes que se dibujarán en pantalla:

(`mx.test.dr4zkdemo.view.model.RegisterCategory` en nuestro caso).

Veamos ahora como se ve nuestro *genericFind.zul*, que representa la pantalla de búsqueda mostrada anteriormente. Nótese la simplicidad y que no está definido ningún tag de control excepto el botón con `id="search"`; este identificador hará que se ejecute la acción `search()` de nuestro controlador abstracto `DRGenericControllerAbstract`. Para este ejemplo se usa una clase controladora genérica `DRGenericController` incluida en *dr4zk*. Si se requiere una funcionalidad especial nuestra clase personalizada siempre debe extender de la abstracta `DRGenericControllerAbstract`.

```

<zkl xmlns="http://www.zkoss.org/2005/zul">
  <zscript>
    ...
  </zscript>
  <div apply="mx.dr.forms.controller.DRGenericController">
    <grid width="700px">
      <rows>
        <row>
          <button id="search" label="{c:l('catalogo.buscar')}" />
          <button id="newone" label="Add">
            ...
          </button>
        </row>
      </rows>
    </grid>
  </div>
</zkl>

```

Veamos ahora nuestra clase del modelo de vista `RegisterCategory` enfocados en la primera acción presentada `SEARCH`; analizaremos cada anotación presente en la clase.

```

@DRActions(actions={
    ...
    @DRFellowLink(action=FormActions.SEARCH,
    submitAction="mx.dr.ml.view.facade.CatalogFacade@findByExampleDTO",loadOnInit=true,
    resultsComponent=@DRListBox(header=true,id="resultado_categorias",itemRenderer=mx.dr.forms.zul.DRResultsListRender.class,
    dtoResult=RegisterCategory.class)))})

```

En primer lugar `DRActions` que se encuentra en la cabecera de nuestra clase: esta anotación determina la lista de acciones `DRFellowLink` que ejecutará nuestro controlador cuando ocurra un evento y qué respuesta de enlace enviará (otra página comúnmente). Dentro de las propiedades de `DRFellowLink` tenemos lo siguiente:

- `loadOnInit=true` indica que el `DRListbox` de resultados de la búsqueda se cargará con una consulta inicial; `header=true` hace que sean visibles los títulos de las columnas resultado del *listbox*.
- `id="resultado_categorias"` es el identificador que `dr4zk` usará para identificar el `DRListbox` de resultados.
- `itemRenderer=mx.dr.forms.zul.DRResultsListRender` especifica el componente que se usará para pintar los resultados encontrados en el `DRListbox`. `Dr4zk` otorga algunos *Renderers* genéricos pero se puede usar alguno personalizado.
- `dtoResult=RegisterCategory.class` define la clase que se usará como referencia al convertir los objetos de persistencia a objetos de vista los cuales manejará `dr4zk` y de obtendrá las columnas que se pintaran en el `DRListbox`.
- `submitAction` es la acción que responderá al presionar el botón con `id="search"` de nuestro `zul`; será la definición del `DRFellowLink` con acción `FormActions.SEARCH` como muestra el código se ejecutará la acción de negocio `findByExampleDTO`, llamada desde la fachada `CatalogFacade`. Esta acción tomará el valor introducido en nuestro componente de texto `nombre` y lo usará como filtro de búsqueda.

Nota: Cabe mencionar que todas las acciones definidas en dr4zk se recomienda que sean llamadas por medio de una fachada, pues dr4zk intentará hacer una instancia de esta clase para ejecutar el método de acción.

Cuando el *grid* de resultados es presentado cada uno de los resultados de la búsqueda puede contener una o más acciones parametrizadas al elemento en cuestión; éstos `DRFellowLink` usan la acción `LIST` como se muestra a continuación en los dos elementos configurados: una acción de liga es para editar el elemento y la segunda es para consultarlo.

```
@DRFellowLink(action=FormActions.LIST,
param="id", fellow=FellowType.SELF, componentPath="//main/myInclude", listLabel=@DRLabel(key="catalogo.editar"), url="/registerCategory.zul?dto_class=mx.test.dr4zkdemo.view.model.RegisterCategory&action=EDIT"),
    @DRFellowLink(action=FormActions.LIST,
param="id", fellow=FellowType.SELF, componentPath="//main/myInclude", listLabel=@DRLabel(key="catalogo.ver"), url="/viewCategory.zul?dto_class=mx.test.dr4zkdemo.view.model.RegisterCategory&action=READ"),
...
```

- `param="id"` es el identificador que pasaremos a la siguiente liga como parámetro para encontrar el elemento que deseamos editar o consultar.
- `url="/registerCategory.zul?dto_class=mx.test.dr4zkdemo.view.model.RegisterCategory&action=EDIT"` define nuestra siguiente liga donde se enviará la petición.
- `componentPath` define el lugar donde será remplazado el contenido de página, esta ruta de componente es absoluta y hace referencia generalmente a un componente; *include*, `componentPath` es necesario si se requiere un `fellow=FellowType.SELF`, esto quiere decir, que la página de respuesta definida en `url` remplazará la vista de nuestra vista actual sin refrescar todo la pantalla.
- `listLabel=@DRLabel` define las etiquetas que serán escritas en la liga para el registro dado.

En las acciones `SEARCH`, `ADD` y `EDIT` los componentes generados deben estar dentro de un contenedor, en la mayoría de los casos se usa un `DRGrid`, como en la anotación mostrada abajo. Note que está definida a nivel de clase lo que indicará que todos los componentes definidos dentro de esta clase (atributos anotados) estarán dentro del *grid*.

```
...
@DRGrid(id="comregGrid", width="700px")
...
public class RegisterCategory extends Base{
...
```

Luego vemos el único campo que se muestra en nuestra pantalla de búsqueda llamado *name*, que tiene una anotación `DRField`. Esta nos dice que este componente será dibujado por *dr4zk* dentro del *grid* con el *label* determinado antepuesto al componente definido

- `order` definirá el orden que será dibujado en pantalla respecto al resto de los controles.
- `actions` define para que acciones será visible el atributo ante el controlador.
- `searchOperator` es usado como soporte para que en la capa de negocio usemos esta operación para el filtro (es opcional); *dr4zkdemo* incluye un ejemplo de como podría ser tratado aunque ya depende de la arquitectura que se plantee.

La acción `FormActions.LIST` indica que este atributo será visible como columna en el `DRListbox` de resultados, el valor que tomará el atributo `name` lo recibirá de un `textbox` como muestra la anotación.

```
@DRField(label=@DRLabel(key="categoria.nombre"),order=2,columnListWidth="300px",searchOperator=DRField.Operator
.LIKE, liveValidate=true, actions={FormActions.ADD, FormActions.EDIT, FormActions.SEARCH, FormActions.LIST,
FormActions.READ},readParent="nameRow")
@DRTextBox(maxLength=50)
@DRValidateNotEmpty
@DRValidateBusinessResult(action="mx.dr.ml.view.facade.CategoryFacade@validateXName")
private String name;
...
```

ADD, EDIT Category

Ahora veamos qué sucede cuando deseamos agregar (clic en el botón *add*) o editar un elemento (clic en la liga *edit* del elemento a editar); se juntaron para este análisis estas funciones porque son muy similares.

La liga que lleva a la acción `ADD` la encontramos en el `zul genericFind.zul` y viene dada como se muestra a continuación.

```
<button id="newone" label="Add" >
<attribute name="onClick">
<![CDATA[
    if(dtoClassName.equals("mx.test.dr4zkdemo.view.model.RegisterCategory")){
        ((Org.zkoss.zul.Include)org.zkoss.zk.ui.Path.getComponent("/main/myInclude")).setSrc("/registerCategory.zul?dto_class=mx.test.dr4zkdemo.view.model.RegisterCategory&action=ADD");
    }else if
    ...
    }
    ]]>
</attribute>
</button>
```

Veamos ahora el `zul registerCategory.zul`. En este `zul` usamos el mismo controlador que en la búsqueda; lo que hace la diferencia es que el único botón presente ahora apunta a `id=generalAction`, acción que se encuentra también en nuestro controlador abstracto `DRGenericControllerAbstract` de donde heredan nuestros controladores.

```
<zk xmlns="http://www.zkoss.org/2005/zul">
<label value="Save Category"/>
<div apply="mx.dr.forms.controller.DRGenericController">
<grid width="700px" >
<rows>
<row >
<button id="generalAction" label="Save" />
</row>
</rows>
```



```

</grid>
</div>
</zk>

```

Ahora veamos la clase del modelo de vista definido en la misma clase `RegisterCategory`, aunque esta vez nos enfocaremos en las acciones `ADD` y `EDIT`. El `DRFellowLink` `action = FormActions.ADD`

- `url` indica la página donde mandamos la respuesta después de realizar la acción de negocio definida en `submitAction`; `fellow=FellowType.SELF` es el modo como se va a presentar esta página.
- `submitAction` es la acción de negocio a ejecutar cuando damos guardar al botón con `id=generalaction`
- `successMessage=@DRMessage()` es el mensaje que se mostrará en pantalla cuando la operación de negocio se ha cumplido con éxito; este mensaje saldrá como un *Messagebox* de *zk*.

Para la acción `EDIT` están definidos los mismos anteriores más la propiedad `param`, que es el identificador que anteriormente pasamos en la acción `FormActions.LIST`. Este identificador es utilizado por el método definido en la propiedad `paramAction`; este define la acción de negocio que se usará para recuperar el elemento con el identificador dado. Recordemos que se recomienda que todas las acciones de negocio se accedan mediante una fachada. Usamos el mismo contenedor *grid* que para la búsqueda. Finalmente la anotación `DRRootEntity` es el tipo de clase persistente (comúnmente) para recuperar el dato o guardar el dato. Es esencial esta anotación en las acciones `READ`, `EDIT` y `ADD` para la transformación entre objetos de persistencia y objetos del modelo de vista para mas información refiérase a la sección de mapeo de atributos.

```

@DRActions(actions={
    ...
    @DRFellowLink(action=FormActions.ADD,
fellow=FellowType.SELF,componentPath="//main/myInclude",url="/genericFind.zul?dto_class=mx.test.dr4zkdemo.view.model.RegisterCategory&action=SEARCH",submitAction="mx.dr.ml.view.facade.CatalogFacade@save",successMessage=@DRMessage(label=@DRLabel(key="anuncio.msg.alta"))),
    @DRFellowLink(action=FormActions.EDIT,
fellow=FellowType.SELF,componentPath="//main/myInclude",url="/genericFind.zul?dto_class=mx.test.dr4zkdemo.view.model.RegisterCategory&action=SEARCH",submitAction="mx.dr.ml.view.facade.CatalogFacade@save",
successMessage=@DRMessage(label=@DRLabel(key="anuncio.msg.update")), param="id",
paramAction="mx.dr.ml.view.facade.CatalogFacade@boById"),
    ...
})
@DRGrid(id="comregGrid",width="700px")
@DRRootEntity(entity=mx.test.vo.Category.class)
public class RegisterCategory extends Base{

```

Las acciones `ADD` y `EDIT` comparten componentes tal es el caso del atributo *name* que como vimos se pintara como un *textbox*. Veamos que también tiene anotadas algunas validaciones aplicables para las acciones `ADD` y `EDIT`.

```

@DRField(label=@DRLabel(key="categoria.nombre"),order=2,columnListWidth="300px",searchOperator=DRField.Operator.LIKE,
liveValidate=true, actions={FormActions.ADD, FormActions.EDIT, FormActions.SEARCH, FormActions.LIST,
FormActions.READ},readParent="nameRow")
@DRTextBox(maxlength=50)

```

```
@DRValidateNotEmpty
@DRValidateBusinessResult(action="mx.dr.ml.view.facade.CategoryFacade@validateXName")
private String name;
```

- `DRValidateNotEmpty` es una validación que otorga dr4zk que se puede usar para pedir el campo como de captura obligatoria.
- `DRValidateBusinessResult` es otra validación que permite implementar una validación personalizada de negocio, para este caso se valida que el nombre dado no exista previamente registrado en la base de datos.
- `liveValidate=true` en `DRField` permite que las validaciones sean ejecutadas cuando el componente pierda el foco; de otra forma se valida hasta que se haga el *submit* de todo el formulario.

El atributo *father* o categoría padre se pintara como una lista seleccionable y desplegable bajo la anotación `DRListBox` con `mold=DRListBox.MOLD.SELECT`. Este tipo de componente necesita un modelo para llenar las diferentes opciones de la lista; se trata de una colección de objetos obtenidos de `model="mx.dr.ml.view.facade.CategoryFacade@findMainCategory"`. Este método regresará la lista de posibles opciones que podrá tomar la lista.
(Note que el tipo de dato a recibir es un `MainCategory` mismo que debe ser el tipo de los elementos de la lista del modelo.)

```
@DRField(label=@DRLabel(key="categoria.padre"),order=5, actions={FormActions.EDIT, FormActions.ADD})
@DRListBox(mold=DRListBox.MOLD.SELECT, itemRenderer=mx.dr.forms.zul.DRResultsListSimpleRender.class,
model="mx.dr.ml.view.facade.CategoryFacade@findMainCategory")
@DRValidateNotEmpty
private MainCategory father;
```

El atributo *id* es anotado como un componente *intbox* y este será visible bajo las acciones EDIT y READ pero de modo sólo lectura.

```
@DRField(label=@DRLabel(key="catalogo.id"),order=1, actions={FormActions.EDIT,
FormActions.READ},readParent="keyRow")
@DRIntBox(readOnly=true)
private Integer id;
```

El atributo *estatusEnum* es asignable solo bajo la acción EDIT.

Note que la lista del modelo es una lista de tipo *enum* al igual que el tipo de atributo a asignar.

```
@DRField(label=@DRLabel(key="catalogo.status"),order=6, actions={FormActions.EDIT})
@DRListBox(mold=DRListBox.MOLD.SELECT, itemRenderer=mx.dr.forms.zul.DRResultsListSimpleRender.class,
model="mx.dr.ml.view.facade.CatalogFacade@getCatalogStatus")
@DRValidateNotEmpty
private CatalogStatus estatusEnum;
```

El atributo `willnotpass` solo se usa para demostrar que no necesariamente todos los atributos de la clase son mapeados, se pueden poner atributos adicionales para usarlos con otro propósito.

```
private String will$not$pass;
```

READ Category

Se accede a la acción READ desde el enlace definido en la acción `FormActions.LIST` de uno de los resultados de búsqueda de categorías, y muestra la siguiente pantalla de lectura.

Category Company		
Id	1	
Name	Kareoke	
Category	Bar	
Status	Active	
Business Name		
MY COMPANY	1	1 Kareoke
<button>Return</button>		

La clase de vista que se usa sigue siendo la misma `RegisterCategory` y el `zul` es `viewCategory.zul`. Note que usamos el mismo controlador genérico `DRGenericController` y ahora lo importante en la acción READ es distribuir la página como deseamos que la vea el usuario y asignar a los contenedores los identificadores que usaremos para insertar nuestro componente en el sitio correcto.

```
<zk xmlns=http://www.zkoss.org/2005/zul xmlns:html=http://www.w3.org/1999/xhtml">
<div apply="mx.dr.forms.controller.DRGenericController" width="100%">
<panel>
<panelchildren style="padding: 20px">
<div>
<grid style="padding: 20px">
<rows>
<row id="keyRow" />
<row id="nameRow" />
<row id="categoryRow" />
<row id="status" />
</rows>
</grid>
<div id="companiesRow"></div>
<button id="returned" label="Return">
...
</button>
</div>
</panelchildren>
</panel>
</div>
</zk>
```

La clase de vista en la acción READ no busca un contenedor general como fue un `DRGrid` anteriormente, pues ahora cada atributo o control especificará el identificador del contenedor en el cual lo colocará. La acción READ debe estar definida en la cabecera de la clase por medio de un `DRFellowLink` y solo requiere dos parámetros.

- `param = "id"` es el nombre del identificador del parámetro que se mando en la página anterior (definido para este caso en la acción `FormActions.LIST` del resultado de búsqueda).

- `paramAction` es la acción de negocio que se usará para encontrar el elemento deseado de lectura.

Por ultimo esta la anotación `DRRootEntity`, clase de persistencia de donde se obtendrá los valores mapeados en nuestra clase de vista.

```
@DRActions(actions={
    ...
    @DRFellowLink(action= FormActions.READ,param = "id", paramAction = "mx.dr.ml.view.facade.CatalogFacade@boById"),
    ...
})
...
@DRRootEntity(entity=mx.test.vo.Category.class)
public class RegisterCategory ...
```

El mapeo de valores de atributos que se lleva acabo entre la clase de vista y la de persistencia definido por la anotación `DRRootEntity` está dado por los nombres y tipos de los atributos. Éstos deben coincidir para que sean tomadas por el mapeo y así `dr4zk` copiará los valores de una a otra. En caso que la profundidad de atributos en los objetos sea mayor a uno, la separación entre nombres será por medio del carácter \$, y el tipo del atributo debe ser el del último que se obtendrá.

Veamos las anotaciones en los atributos en la acción `READ` iniciando por la anotación `DRField`. La propiedad `readParent` es el contenedor donde se colocará el componente incrustado como hijo, en caso de definir un `DRLabel` (no obligatorio); este será antepuesto al control, la anotación del componente es imprescindible.

Nuestro atributo *name* será pintado como un *textbox* editable solo para propósitos del ejemplo y será pintado dentro del componente con `id="nameRow"`.

```
@DRField(label=@DRLabel(key="categoria.nombre"),order=2,columnListWidth="300px",searchOperator=DRField.Operator.LIKE,
liveValidate=true, actions={FormActions.ADD, FormActions.EDIT, FormActions.SEARCH, FormActions.LIST,
FormActions.READ},readParent="nameRow")
@DRTextBox(maxlength=50)
...
private String name;
```

El `id` será puesto como un *intbox* de sólo lectura dentro del componente con `id="KeyRow"`.

```
@DRField(label=@DRLabel(key="catalogo.id"),order=1, actions={FormActions.EDIT, FormActions.READ},readParent="keyRow")
@DRIntBox(readOnly=true)
private Integer id;
```

El atributo `estatusEnum$labelDescription` y `father$name` serán dibujados como *label*. Note que hay profundidad en el mapeo del atributo, lo que quiere decir que `estatusEnum$labelDescription` será asignado desde el atributo `labelDescription` del atributo `estatusEnum` del objeto `Category` consultado; y `father$name` será asignado desde el atributo `name` del atributo `father` de `Category`.

```
@DRField(actions= FormActions.READ, readParent="status", label =
    @DRLabel(key = "catalogo.status"))
@DRLabel(key = DRLabel.NO_LABEL)
```

```
private String estatusEnum\$labelDescription;
```

```
@DRField(actions= FormActions.READ, readParent="categoryRow", label =
    @DRLabel(key = "categoria.padre"))
@DRLabel(key = DRLabel.NO\_LABEL)
private String father\$name;
```

Por último, `companies` será dibujado como una lista de empresas que pertenecen a la categoría consultada. Este atributo no está mapeado en `Category`; su valor será asignado mediante el modelo definido en la acción de negocio:

```
model="mx.dr.ml.view.facade.CompanyFacade@requestByCategory".
```

```
@DRField(actions= FormActions.READ, readParent="companiesRow", label =
    @DRLabel(key = "empresa.nombre"))
@DRListBox(header=false,model="mx.dr.ml.view.facade.CompanyFacade@requestByCategory",modelParams={"id"},dtoResult=SearchCompanyMain.class)
private List companies;
```

SEARCH Company

El segundo módulo llamado compañía es ligeramente más complejo que categoría, y se asemeja más a un caso real que se nos podría presentar en nuestra aplicación web empresarial. La pantalla para la acción SEARCH es como se muestra a continuación.

El código fuente de este zul se encuentra en el archivo `genericFind.zul`. Note que esta página es la misma que utiliza categoría, **esto demuestra la reutilización de páginas comunes lo cual es posible en dr4zk aún si no se trata del mismo tipo de clase de modelo de vista.**

```
<zul xmlns="http://www.zkoss.org/2005/zul">
  <zscript>
    <![CDATA[
      ...
    ]]>
  </zscript>
  <div apply="mx.dr.forms.controller.DRGenericController">
    <grid width="700px">
      <rows>
        <row>
          <button id="search" label="{c:l('catalogo.buscar')}" />
          <button id="newone" label="Add" >
            <attribute name="onClick">
              <![CDATA[
                if ...
              ]>
            </attribute>
          </button>
        </row>
      </rows>
    </grid>
  </div>
</zul>
```

```

        ((org.zkoss.zul.Include)
org.zkoss.zk.ui.Path.getComponent("//main/myInclude")).setSrc("/registerCompany.zul?dto_class=mx.test.dr4zkdemo.view.mod
el.RegisterCompanyMain&action=ADD");
    }
]]>
</attribute>
</button>
</row>
</rows>
</grid>
</div>
</zk>

```

Para el modulo compañía las acciones están repartidas en diferentes clases de vista a diferencia de categoría en donde una contenía la configuración de todas la acciones.

La clase para la acción SEARCH se llama SearchCompanyMain. Analizando las propiedades de la anotación DRFellowLink en esta clase podemos ver que esta vez no se llamará a una consulta inicial al cargar la página esto es dado por loadOnInit=false que es el valor por defecto. Recordemos que nuestros resultados serán convertidos a la clase definida en dtoResult=SearchCompanyMain.class, de donde se tomarán los atributos a pintar en las columnas definidos por la anotación DRField con acción LIST.

```

@DRActions(actions={
@DRFellowLink(action= FormActions.SEARCH, submitAction="mx.dr.ml.view.facade.CatalogFacade@findByExampleDTO",
resultsComponent=@DRListBox(header=true,id="resultado_empresas",itemRenderer=mx.dr.forms.zul.DRResultsListRender.class
, dtoResult=SearchCompanyMain.class))

```

Las anotaciones DRFellowLink con acción LIST, ahora una forma distinta de invocar la página de respuesta ésta vez será un *popup* (fellow=FellowType.POPUP), en el caso de editar el elemento; y una nueva página (fellow=FellowType.NEW) en el caso de consultar. Las demás propiedades son las ya explicadas en el modulo categoría. Recordemos que es necesario anotar la entidad de persistencia que representa el modelo de vista

@DRRootEntity(entity=mx.test.vo.Company.class), y su contenedor (DRGrid para este caso).

```

@DRFellowLink(action= FormActions.LIST,
param="id",fellow=FellowType.POPUP,listLabel=@DRLabel(key="catalogo.editar"),url="/editCompany.zul?dto_class=mx.test.dr4
zkdemo.view.model.EditCompanyMain&action=EDIT")
, @DRFellowLink(action= FormActions.LIST,
param="id",fellow=FellowType.NEW,listLabel=@DRLabel(key="catalogo.ver"),url="/viewCompany.zul?dto_class=mx.test.dr4zkdem
o.view.model.ViewCompany&action=READ")
})
@DRGrid(id="emregGrid",width="700px")
@DRRootEntity(entity=mx.test.vo.Company.class)
public class SearchCompanyMain extends Base{

```

El atributo brand o nombre de la empresa en SearchCompanyMain se dibujará como un *textbox*, y id como un *Intbox*. Note que esta vez tenemos presente una anotación de validación que queremos que se ejecute bajo la acción SEARCH, se hace especificando la propiedad applySearch=true.

```

@DRValidateNotEmpty(applySearch=true)
@DRField(actions= {FormActions.SEARCH, FormActions.LIST},
label=@DRLabel(key="empresa.nombre"),order=2,searchOperator=DRField.Operator.LIKE)

```

```

@DRTextBox(maxlength=50)
private String brand;

@DRField(actions= {FormActions.SEARCH, FormActions.LIST},
label=@DRLabel(key="catalogo.clave"),order=1,searchOperator=DRField.Operator.EQUALS)
@DRIntBox(maxlength=50)
private Integer id;

```

El atributo `category` se pintará como lista seleccionable (`MOLD.PAGING`), es decir, todos los elementos serán mostrados en la lista, éste es el molde por defecto.

```

@DRField(actions= {FormActions.SEARCH, FormActions.LIST},
label=@DRLabel(key="categoria.padre"),order=3,searchOperator=DRField.Operator.EQUALS)
@DRListBox(model="mx.dr.ml.view.facade.CategoryFacade@FindActive")
private Category category;

```

`brand`, `id` y `category` serán pintados como filtros de búsqueda (`FormActions.SEARCH`) y como columnas en el `listbox` de resultados (`FormActions.LIST`). Recordemos también que los operadores de los filtros (`searchOperator=...`) son sólo de soporte y sirven como guía para el método de negocio que realizara la acción de búsqueda; `dr4zk` no los requiere de forma estricta. Recordemos que el orden en que serán pintados los componentes está dado por la propiedad `order`.

ADD, EDIT Company

Accedemos a la pantalla de la acción `ADD` haciendo clic sobre el botón `Add` de la pantalla de búsqueda, esta pantalla nos permite registrar una nueva Compañía.

El código fuente se encuentra en el archivo `registerCompany.zul`, vemos que es igualmente simple de componentes, solo hay un botón con `id="generalAction"` que nos permitirá guardar los datos de la compañía que introdujo el usuario. Note que en el `tag iframe` hay una inclusión de otra sección o pantalla que maneja `dr4zk`, se trata del contrato de usuario, el cual no se analizará

en este documento por su simplicidad, pero se menciona para demostrar la convivencia de distintas secciones manejadas por dr4zk en una sola vista de pantalla.

```
<zk xmlns="http://www.zkoss.org/2005/zul">
<div apply="mx.dr.forms.controller.DRGenericController">
<grid width="880px" >
<rows>
<row>
<iframe id="iframe"
src="companyContract.zul?dto_class=mx.test.dr4zkdemo.view.model.ViewContract&action=READ&type=E"/>
</row>
<row >
<button id="generalAction" label="Save" />
</row>
</rows>
</grid>
</div>
</zk>
```

La clase de vista RegisterCompanyMain es la utilizada para la acción ADD; podemos ver que no hay más parámetros adicionales a los ya vistos en categoría. Cabe resaltar que nuestro contenedor es un *grid* de una columna (`DRGrid(cols=1)`), y la clase de persistencia para el mapeo de la transformación de objetos de vista y persistencia es ahora:

```
DRRootEntity(entity=mx.test.vo.Company.class) .
```

```
@DRFellowLink(action=
FormActions.ADD,fellow=FellowType.SELF,componentPath="//main/myInclude",url="/genericFind.zul?dto_class=mx.test.dr4zkdemo.view.model.SearchCompanyMain&action=SEARCH",submitAction="mx.dr.ml.view.facade.CompanyFacade@saveNewCompany")
@DRRootEntity(entity=mx.test.vo.Company.class)
@DRGrid(id="registerComGrid", cols=1,width="950px")
public class RegisterCompanyMain {
```

En los atributos de la clase RegisterCompanyMain vemos algo distinto a nuestros casos anteriores, y es que dr4zk permite la inclusión de contenedores dentro de un contenedor principal; esto se refleja en una composición de clases. Para este ejemplo nuestros contenedores hijos son `DRGroupBox` y todos los componentes anotados en los atributos de la clase contenida serán incluidos en *groupbox* y la propiedad `label` será el título del *groupbox*.

```
@DRField(actions= FormActions.ADD,isField=false, order=1, label=@DRLabel(key="registro.reg.contacto"))
@DRGroupBox(mold=DRGroupBox.MOLD._3D,width="900px")
private RegisterCompanyContact contact;

@DRField(actions= FormActions.ADD,isField=false, order=2, label=@DRLabel(key="registro.reg.comensal"))
@DRGroupBox(mold=DRGroupBox.MOLD._3D,width="900px")
private RegisterCompany company;

@DRField(actions= FormActions.ADD,isField=false, order=3, label=@DRLabel(key="registro.reg.localizacion"))
@DRGroupBox(mold=DRGroupBox.MOLD._3D,width="900px")
private RegisterCompanyAddress address;
```

Una clase contenida en la anterior composición es RegisterCompany. Vemos que no se requiere volver a anotar el `DRActions`, `DRFellowLink` o `DRRootEntity`, pues estos son previamente leídos de la clase contenedora, solo es requerido especificar el contenedor para los componentes incluidos en esta clase, para este caso un `DRGrid`.

```
@DRGrid(id="emplegGrid",width="880px")
public class RegisterCompany extends Base {
```


El común de las anotaciones en los atributos que describen los componentes de la pantalla ya se han analizado en categoría; revisemos solo los componentes nuevos o propiedades no mencionadas anteriormente.

```
@DRValidateNotEmpty
@DRField(actions= FormActions.ADD, label=@DRLabel(key="registro.marca"), order=-1)
@DRTextBox(maxlength=99, uppercase=true, cols=40)
private String brand;
```

El atributo *category* será la categoría que asignemos a nuestra compañía, y esta se asigna por medio de un componente *Combobox*. Este al igual que el *listbox* requiere de un modelo, el cual es obtenido por el método escrito en la propiedad `model=mx.dr.ml.view.facade.CategoryFacade@findActive`. *DRCombobox* cuenta con una propiedad `action=mx.dr.ml.view.facade.CategoryFacade@save` la cual definirá qué hacer cuando el valor que nos da el usuario no está en nuestra lista de opciones del modelo. Para este específico caso, si la descripción no coincide, se guardará como un nuevo registro de categoría.

```
@DRValidateNotEmpty
@DRField(actions= FormActions.ADD, label=@DRLabel(key="registro.reg.giro"), order=0)
@DRComboBox(model="mx.dr.ml.view.facade.CategoryFacade@findActive",
action="mx.dr.ml.view.facade.CategoryFacade@save")
private Category category;
```

El atributo *media* anota un componente *DRAttachList*, que pintará un componente personalizado de *dr4zk* para cargar y descargar archivos; con su propiedad `maxRow=1` define el número máximo de archivos para adjuntar, en este caso solo acepta 1. Note que el tipo del atributo no es un tipo mapeado en el *DRRootEntity* (*Company.class*), puesto que este debe ser una lista de objetos de clase *DRMedia* aun cuando el elemento sea solo uno, y para que tenga un tratamiento especial de archivo; es imprescindible la anotación *DRIsMedia*.

```
@DRValidateNotEmpty
@DRField(actions= FormActions.ADD, label=@DRLabel(key="registro.reg.logo"), order=5)
@DRAttachList(maxRow=1)
@DRIsMedia
private List<DRMedia> media;
```

El atributo *link* que espera una *url* válida por medio de un *textbox*, anota un tipo de validación para hacer esto *DRValidateStringPattern*; esta anotación valida que el valor obtenido tenga el patrón especificado.

```
@DRValidateNotEmpty
@DRValidateStringPattern(pattern = "\\b(https?):\\/[-a-zA-Z0-9+&@#/%?=_!:,.;]*[-a-zA-Z0-9+&@#/%?=_!:]")
@DRField(actions= FormActions.ADD, label=@DRLabel(key="registro.link"), order=6)
@DRTextBox(maxlength=99)
private String link;
```

Ahora veamos la clase que se utiliza en la acción *EDIT*, esta es *EditCompanyMain*; la intención de dividir la clase de vista distinta para la acción *ADD* y *EDIT* es demostrar los diferentes contenedores disponibles, pues en la acción *EDIT* se usa un contenedor: un *DRTabBox*, el cual creará una estructura de *tabs* para contener sus atributos.

```

@DRFellowLink(action= FormActions.EDIT,fellow=FellowType.POPUP,
componentPath="//main/myInclude",url="/genericFind.zul?dto_class=mx.test.dr4zkdemo.view.model.SearchCompanyMain&action=SEARCH",
    submitAction="mx.dr.ml.view.facade.CompanyFacade@updateCompany",
    param="id",
    paramAction="mx.dr.ml.view.facade.CatalogFacade@boById",
    successMessage=@DRMessage(label=@DRLabel(key="anuncio.msg.alta"))))
@DRRootEntity(entity=mx.test.vo.Company.class)
@DRTabBox(width="900px")
public class EditCompanyMain {

```

Cada uno de sus atributos representa un *tab* con título dado por la propiedad `label` de la anotación `DRField`. Note la reutilización de clases que se usaron en la acción `ADD` (`RegisterCompanyAddress`, `RegisterCompanyContact`); ésto demuestra la fácil reutilización de código de vista en `dr4zk`.

```

@DRField(actions= FormActions.EDIT,isField=false, label=@DRLabel(key="registro.reg.info.fiscal"), order=4)
@DRTabPanel
private RegisterCompanyFiscal fiscal;

@DRField(actions= FormActions.EDIT,isField=false, label=@DRLabel(key="registro.reg.localizacion"), order=3)
@DRTabPanel
private RegisterCompanyAddress address;

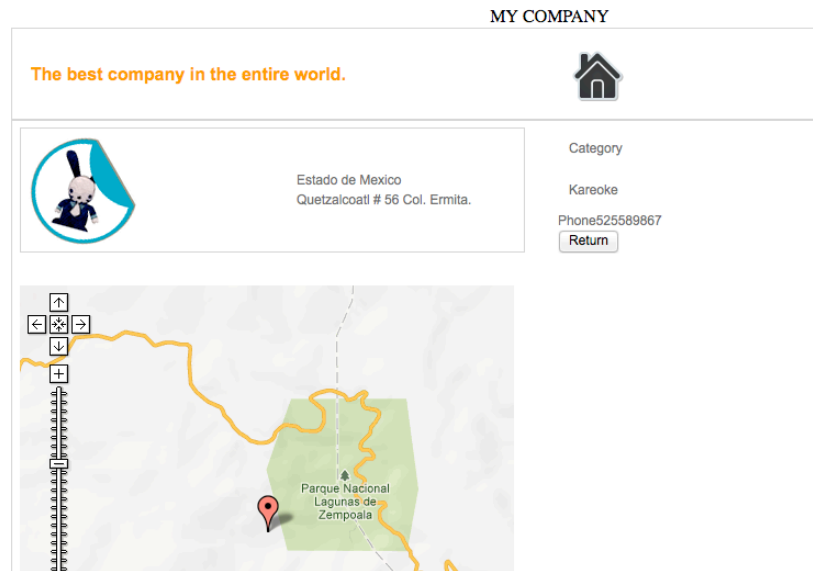
@DRField(actions= FormActions.EDIT,isField=false, label=@DRLabel(key="registro.reg.comensal"), order=2)
@DRTabPanel
private EditCompany company;

@DRField(actions= FormActions.EDIT, isField=false, label=@DRLabel(key="registro.reg.contacto"), order=1)
@DRTabPanel
private RegisterCompanyContact contact;

```

READ Company

Después de buscar una compañía y al hacer clic sobre la liga *view* nos llevará a la pantalla que controla la acción `READ`, la cual es una página personalizada para mostrar los datos de la compañía como queremos que la vea nuestro cliente. El código de esta pagina puede verse en el archivo `viewCompany.zul`



Para la acción read, la clase `ViewCompany` no es muy diferente a lo ya mencionado en categoría para la acción READ. Recordemos que la propiedad `readParent` de `DRField` indica el nombre del identificador del componente contenedor donde se dibujará nuestro componente anotado.

```
@DRRootEntity(entity = mx.test.vo.Company.class)
@DRFellowLink(action= FormActions.READ,param = "id", paramAction = "mx.dr.ml.view.facade.CompanyFacade@companyById")
public class ViewCompany {
```

Note que el atributo `id` no está anotado con algún componente; esto es porque se trata del identificador que se usó como parámetro para ser recuperado del `DRRootEntity` (`Company.class`).

```
private Integer id;

@DRField(actions= FormActions.READ, readParent = "nombreRow", label =
@DRLabel(key = DRLabel.NO_LABEL))
@DRLabel(key = DRLabel.NO_LABEL,sclass="uptitulo")
private String brand;

@DRField(actions= FormActions.READ, readParent = "sloganRow", label =
```

`DRImage` pintará una imagen desde la ruta definida en él, del valor recuperado del atributo `uri` del valor del atributo `logo` de nuestro `DRRootEntity` (`Company.class`); a esta ruta se le colocará el prefijo dado en la propiedad `prepend = "."`.

```
@DRField(actions= FormActions.READ, readParent = "imgPerfil", label =
@DRLabel(key = DRLabel.NO_LABEL))
@DRImage(width = "100px", prepend = ".")
private String logo$uri;
```

`DRGmaps` anotada en el atributo `address$latitude` pintará un componente mapa de *google maps*. Note que hace referencia a más de un atributo de la clase por medio de sus propiedades `latitude = "address$latitude"`, `longitude = "address$longitude"`, `gmarkContent = "Brand"`, por lo que éstos atributos deben existir valuados en la clase, caso contrario se generará un error. Por último esta anotación requiere una llave de *google* válida incrustada en el `zul` mediante un `js`.

```

@DRField(actions= FormActions.READ, readParent = "mapRow", label =
@DRLabel(key = DRLabel.NO_LABEL))
@DRGmaps(showLargeCtrl=true, width = "450px", height = "450px", latitude = "address$latitude", longitude =
"address$longitude", gmarkContent="brand")
private Double address$latitude;
private Double address$longitude;

```

La anotación personalizada `DRLink` del atributo `link` es explicada en la sección llamada control personalizado.

```

@DRField(actions= FormActions.READ, readParent = "linkRow", label =
@DRLabel(key = DRLabel.NO_LABEL))
@DRLink(image="/img/home_link.png")
private String link;

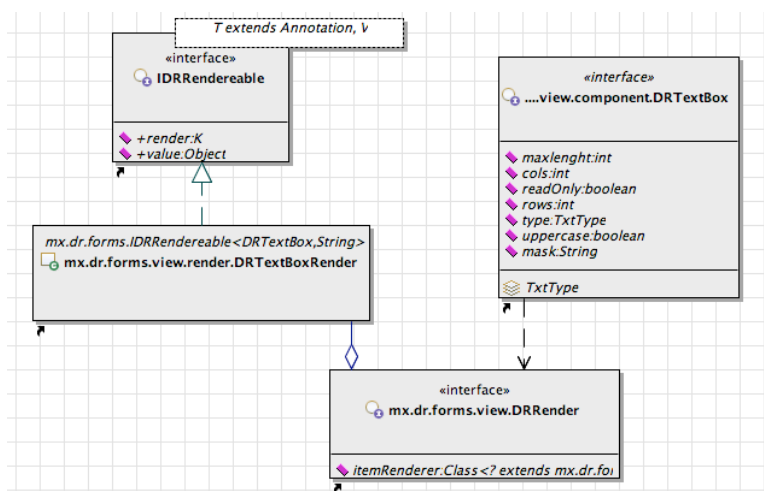
```

Con lo anterior se ha dado un vistazo a las capacidades de `dr4zk` para la generación de vistas web en una aplicación empresarial. Los siguientes capítulos del documento explican a detalle cada parte de las anotaciones y clases de `dr4zk`.

Control personalizado.

`Dr4zk` permite la implementación de clases personalizadas de validación o clases personalizadas de controles; ambos esencialmente siguen el patrón mostrado en la siguiente imagen. Tomaremos el caso de un control personalizado, para ello se deben hacer dos cosas:

- 1.- Crear una clase que implemente la interfaz `IDRRenderable`. Al implementar su método `render`, nos dirá como se va a pintar el componente en la pantalla; y el método `value` dirá como vamos a recuperar el valor del componente que será asignado al atributo anotado.
- 2.- Crear una anotación; esta definirá los parámetros que podemos tener para pintar nuestro componente y será la que se anote en el atributo de nuestra clase de vista. Es imprescindible anotar `DRRender` en esta clase, ya que ésta hará referencia a nuestra clase creada en el punto 1.



Retornemos el caso de `dr4zkdemo`, y supongamos que queremos crear un componente de solo lectura que coloque un enlace en una imagen que tomará el valor de la `URL` de nuestro atributo de alguna forma. La clase `DRLinkRender` implementa la solución sobre cómo se pintará el

componente en la pantalla. Note que el método `value` no regresa un valor pues este es de solo lectura y en su lugar solo leerá el valor del atributo anotado. El método `render` para esta solución en específico crea un objeto `Toolbarbutton` y le coloca como *url* el valor obtenido de nuestro objeto de vista valuado (exclusivo para una acción READ o EDIT) y es colocado una imagen al `Toolbarbutton` tomado de la propiedad `image` definida en la anotación `DRLink` que también crearemos.

```
public class DRLinkRender implements IDRRendereable<DRLink,String> {

    public Component render(final DRLink drLink,final String name,final String value,final Object dtoValue)throws
    Exception {
        Toolbarbutton toolbarbutton = new Toolbarbutton();
        if (value != null) {
            toolbarbutton.setHref(value);
            toolbarbutton.setImage(drLink.image());
            toolbarbutton.setTarget("_blank");
        }
        toolbarbutton.setId(name);
        return toolbarbutton;
    }

    public Object value(final DRLink drLink, final Component comp, final Class<String> expectedType) {
        return null;
    }

}
```

`DRLink` es la anotación que resuelve la necesidad bastante simple, su única propiedad es `image` que es la ruta absoluta a la imagen a desplegar, esta se usará como icono del enlace. En la definición de la anotación note que está presente la anotación `DRRender` (`itemRenderer = DRLinkRender.class`), ésta es la liga entre la anotación y la clase anterior, es decir, aquí se especifica qué clase se usará para pintar el componente y la forma de cómo obtener el valor del componente.

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@DRRender(itemRenderer=DRLinkRender.class)
public @interface DRLink{

    String image();
}
```

Con lo anterior `dr4zk` leerá esta anotación que hemos hecho y pintará el componente especificado en nuestra pantalla, un ejemplo de cómo implementarla en un atributo es como sigue:

```
@DRLink(image="/img/home_link.png")
private String link;
```

Acciones

El acceso a las funciones de `dr4zk` para desplegar una forma deseada en una vista es mediante la *url* del *zul* donde se encuentra el controlador manejado por *dr4zk*; esta *url* determina el comportamiento de la vista mediante dos parámetros obligatorios.

```
myView.zul?dto_class=ANOTATED_DR4ZK_VIEWCLASS&action=FORM_ACTION
```

- `dto_class`: es la clase de vista manejada por dr4zk en donde se anota el comportamiento de las diferentes acciones y los componentes que son visibles para cada acción.
- `action`: es el comportamiento que se invocará del `dto_class` especificado esta acción puede ser SEARCH, ADD, EDIT, READ. Si no es especificada por defecto será tratada como ADD.

Se recomienda que la `url` se coloque dentro de un componente `Include`, esto es para que sean transparente para el usuario final la clase y acciones usados en la pantalla.

Una o más acciones pueden ser anotada en la clase de vista, si es única la que se requiere se puede apoyar directamente con la anotación de `DRFellowLink`, si son varias se anota un `DRActions`.

DRActions

Es una anotación que permite una colección de `DRFellowLink`, donde cada una de ellas define una acción distinta. Las acciones definidas deben ser únicas por clase, por ejemplo solo debe ser definida una sola acción ADD; la acción LIST es una excepción ya que esta si puede ser definida de forma repetida. La propiedad única de la anotación es la siguiente:

- ```
public DRFellowLink[] actions();
```

  
Conjunto de acciones que manejará el modelo de vista.

### ***DRFellowLink***

Anotación que define un comportamiento de acción en nuestra clase de vista, esta anotación se coloca en la definición de la clase. Existen cinco tipos de acciones que se definen, y están dadas en la enumeración `FormActions`, éstas son: SEARCH, ADD, EDIT, READ y LIST; hay propiedades comunes para todas ellas y también hay otras que atienden a una acción en específico. El siguiente cuadro de propiedades define que propiedades se pueden utilizar en que acción.

| Propiedad                                                       | Descripción                                                                                                                     | ADD | EDIT | READ | SEARCH | LIST |
|-----------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|-----|------|------|--------|------|
| <code>public FormActions action();</code>                       | Acciones: SEARCH, ADD, EDIT, READ y LIST                                                                                        | X   | X    | X    | X      | X    |
| <code>String componentPath() default DefaultValues.NONE;</code> | Ruta absoluta de la página y componente del tipo <code>Include</code> específico donde se cargará la nueva sección de pantalla; | X   | X    |      |        | X    |

|                                                                                 |                                                                                                                                                                                                                                                                                                                           |   |   |   |   |   |
|---------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---|---|---|---|---|
|                                                                                 | solo es requerido para el <code>FellowType.SELF</code> .<br>Un ejemplo de ruta es:<br><code>//main/myWindow/myInclude</code>                                                                                                                                                                                              |   |   |   |   |   |
| <code>FellowType<br/>fellow() default<br/>FellowType.NEW;</code>                | Define como será presentado al usuario la <i>url</i> de respuesta de la acción por defecto será en una nueva pantalla.                                                                                                                                                                                                    | X | X |   |   | X |
| <code>String<br/>submitAction()<br/>default<br/>DefaultValues.NONE<br/>;</code> | Referencia absoluta al nombre del método de negocio a ejecutar tras la acción, se recomienda que se use el patrón fachada ya que <i>dz4zk</i> tratará de crear una instancia de la clase definida para ejecutar el método, el formato del nombre es el siguiente:<br><a href="#">your.package.YourFacade@actionMethod</a> | X | X |   | X |   |
| <code>String url()<br/>default<br/>DefaultValues.NONE<br/>;</code>              | Ésta será la URL donde se dirigirá la respuesta al realizarse la acción                                                                                                                                                                                                                                                   | X | X |   |   | X |
| <code>String[]<br/>sessionParams()<br/>default {};</code>                       | Contiene los identificadores de los objetos a extraer que se encuentran contenidos en el mapa de la http sesión de usuario.                                                                                                                                                                                               | X | X | X | X | X |
| <code>DRMessage<br/>successMessage()<br/>default<br/>@DRMessage;</code>         | Mensaje a mostrar cuando se realice la acción de forma exitosa.                                                                                                                                                                                                                                                           | X | X |   | X |   |

|                                                                                         |                                                                                                                                                                                                                                                                                                             |  |   |   |   |   |
|-----------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--|---|---|---|---|
|                                                                                         |                                                                                                                                                                                                                                                                                                             |  |   |   |   |   |
| <code>@DRLabel<br/>listLabel()<br/>default<br/>@DRLabel(key=DefaultValues.NONE);</code> | Para el caso exclusivo de una acción en la lista de resultados, se trata de la etiqueta que deberá tener el enlace.                                                                                                                                                                                         |  |   |   |   | X |
| <code>String param()<br/>default<br/>DefaultValues.NONE<br/>;</code>                    | Es el nombre del atributo del objeto de vista que se pasará como parámetro de entrada para la recuperación del registro deseado en la acción EDIT, READ o LIST                                                                                                                                              |  | X | X |   | X |
| <code>String<br/>paramAction()<br/>default<br/>DefaultValues.NONE<br/>;</code>          | Acción que se realizará para recuperar el resultado; usa como parámetro el valor definido por el atributo <code>param</code> anteriormente mencionado, este es aplicable en las acciones EDIT, READ o LIST. Se recomienda que esta sea un método de una fachada:<br><code>package.FacadeClass@method</code> |  | X | X |   | X |
| <code>DRListBox<br/>resultsComponent()<br/>default<br/>@DRListBox;</code>               | Es la definición del <code>Listbox</code> donde se pintarán los resultados de la búsqueda, este es exclusivo de la acción de SEARCH                                                                                                                                                                         |  |   |   | X |   |
| <code>boolean<br/>loadOnInit()<br/>default false;</code>                                | Determina si habrá un pre-llenado de resultados en el                                                                                                                                                                                                                                                       |  |   |   | X |   |



|  |                                                                                                                                                     |  |  |  |  |  |
|--|-----------------------------------------------------------------------------------------------------------------------------------------------------|--|--|--|--|--|
|  | Listbox<br>resultCodeComponent<br>Éste es exclusivo de<br>la acción de SEARCH<br>El submitAction<br>debe tener un<br>comportamiento por<br>defecto. |  |  |  |  |  |
|--|-----------------------------------------------------------------------------------------------------------------------------------------------------|--|--|--|--|--|

## Mapeo de atributos clases de negocio y clases del modelo de vista.

*Dr4zk* es capaz de copiar el valor de los atributos de una clase de vista a una clase de negocio (comúnmente de persistencia) y viceversa simulando así una transformación de objeto de vista a objeto de negocio. En las acciones READ y EDIT se usa este procedimiento para obtener los valores de los atributos y controles que vamos a mostrar en pantalla de la clase de negocio a la clase de vista. Y en el caso de ADD y EDIT va de vista a objeto de negocio y se usa para obtener los objetos de negocio ya valuados que vamos a usar en la lógica de negocio y en el común de los casos terminar con una acción de persistencia. Para esto se determina cuál clase de negocio es representada en una clase de vista especificada con la anotación `DRRootEntity`.

### ***DRRootEntity.***

Al marcar una clase de vista con esta anotación indicará a *dr4zk* que los atributos definidos en la clase de vista tienen su análogo en la clase de persistencia especificada en `DRRootEntity`, y copiará los valores de los atributos del modelo de vista a un objeto de la clase definida y viceversa (en acciones READ y EDIT). Lo siguiente es un ejemplo de su uso:

```
@DRRootEntity(entity=mx.test.vo.Company.class)
```

En casos especiales se requiere mapear diferentes entidades de negocio independientes en un solo modelo de vista. Existe la opción de usar un mapeo múltiple de entidades de negocio restringiendo el mapeo a los atributos contenidos en la clase de vista anotada con `DRRootEntity`, a continuación se ilustra esto.

```
public class A{
 private String name;
}

public class B{
 private String street;
}
```

```

@DRRootEntity(entity=A.class)
@DRGrid
public class View{
 @DRField
 private String name;
 @DRField
 @DRGroupbox
 private ViewSection section
}

@DRRootEntity(entity=B.class)
@DRGrid
public class ViewSection{
 @DRField
 private String street;
}

```

El mapeo de atributos que hay entre las clases de modelo vista y clases de negocio se realiza por medio del nombre del atributo, este debe ser igual, en caso de haber composición la profundidad del atributo deseado se divide por medio de un carácter \$ el siguiente es un ejemplo de mapeo.

```

public class A{
 private String name;
}

public class B{
 private A a;
}

@DRRootEntity(entity=A.class)
public class View{
 @DRField
 private String a$name;
}

```

El resultado de la transformación de objetos vista a persistencia es entregado como parámetro en el método `doSomething` (para la acción ADD y EDIT) y `search` (acción SEARCH) de nuestra clase controladora que implemente la clase abstracta `DRGenericControllerAbstract`.

El resultado del proceso de transformación de objetos vista a persistencia viene encapsulado como una lista de objetos de negocio (o persistencia) contenidos en el objeto `DRGenericDTOIN` bajo su atributo `bos`; en el común de los casos esta lista tendrá un solo elemento. El atributo `viewDTO` es una instancia de la clase de vista con los atributos valuados según lo introducido por el usuario en la forma de la pantalla.

## Contenedores.

Los componentes generados por los atributos anotados en la clase de vista para las acciones de ADD, EDIT y SEARCH deben estar organizados en contenedores que se anota en la cabecera de la clase de forma obligatoria. Lo anterior otorga forma a la presentación de los componentes; el más común es `DRGrid`. Las clases de modelo de vista anotadas con `DRGrid` construirán el conjunto de componentes definidos dentro de un *Grid* de ZK una columna por su `label` definido y otra columna por el componente mismo, las propiedades son las siguientes:

- `String id();`  
Identificador único del *Grid* a crear.
- `String width();`  
Es la anchura del Grid.
- `int cols();`  
Número de columnas que tendrá el *Grid*.

El siguiente es un ejemplo de su uso.

```
@DRGrid(id="my_grid", cols=4, width="500px")
public class View{..
```

Existen contenedores especiales como los siguientes mencionados.

- `DRTabbox-DRTabpanel` el cual dibujará un `Tabbox` con sus `Tabpanel` de ZK. Para su implementación se debe crear una clase modelo de vista principal que será marcada por un `DRTabbox`; luego cada atributo del tipo de clase contenida en esta clase principal estará marcado por un `DRTabpanel`.
- `DRGroupbox` dividirá en segmentos los conjuntos de componentes mediante `Groupbox` de ZK. De igual manera los atributos de la clase principal que estén especificados como clase de componentes contenida pueden ser anotados por esta anotación para manejar agrupaciones de componentes.

La definición de ambos es muy similar, el detalle y ejemplo de su implementación puede ser consultada en la sección del análisis a la aplicación `dr4zkdemo`.

## Validaciones.

`Dr4zk` provee de un conjunto de validaciones básicas que pueden ser combinadas sobre un atributo que representa un componente en la clase del modelo de vista esto es anotado como

`DRField` y su respectiva anotación de componente (`DRTextbox`, `DRIntbox` etc). A continuación se detallan las diferentes clases de validación disponibles como anotaciones.

Una propiedad común en ellas es `applySearch() default false`; por defecto las validaciones solo aplican para las acciones `ADD` y `EDIT` asignando un valor de `true` a esta propiedad hace que la validación en cuestión también aplique para bajo la acción `SEARCH` para el atributo anotado.

- `DRValidateCaptcha` define el comportamiento de la validación de coincidencia de texto en un `captcha` esta validación debe estar presente de forma obligada en un componente `DRcaptcha` y su uso es exclusivo.
- `DRValidateBussinessResult` define el comportamiento de la validación que utiliza el valor del atributo anotado como parámetro de entrada a un método de negocio donde se realizara la validación correspondiente, esta anotación se subscribe para realizar una validación de negocio sobre el valor introducido por el usuario, un ejemplo es un registro que no se debe repetir en la base de datos para el valor dado. Este método de negocio debe regresar un valor verdadero si pasa la validación o falso si no la pasa. Sus propiedades son las siguientes:
  - `String action()`;  
Define la acción que realizara la operación de negocio para determinar si el valor introducido es valido, el valor devuelto deberá ser un booleano con valor verdadero y la validación se cumple y falso si no se cumple, `package.Facade@method`
- `DRValidateLenght` es la anotación que define el comportamiento de la validación de longitud mínima o máxima del valor introducido, este es aplicable para atributos tipo `String`, `Integer`, `Double` o `BigDecimal`.
  - `int lenght()`;  
Longitud definida del valor.
  - `MODE mode() default MODE.MIN`;  
Determina si la validación de longitud definida aplica para un valor en su longitud mínima o máxima
- `DRValidateNotEmpty` validación que indica que un campo es obligatorio en su captura por el usuario. Normalmente esta validación no requiere valuar ninguno de sus propiedades, sin embargo para el caso especial de validar valores en `Listbox` o `Combobox` el primer valor del modelo debe ser el considerado como vacío ejemplo un objeto del modelo con valor despliegue "seleccione una opción", para este caso es necesario definir las siguientes propiedades:
  - `String property() default DefaultValues.NONE`;  
Es el nombre de un atributo a validar encapsulado en el objeto que conforma el modelo

- `String emptyValue() default "-1";`  
El valor del atributo referenciado en `property` que representa una selección vacía.
- `DRValidateReferenceDate` es la anotación que aplica la validación de comparar que el valor del atributo anotado sea mayor o menor a otro valor de otro atributo referenciado (por el momento solo aplicable a fechas).
- `String property();`  
Nombre del atributo referenciado como punto de comparación.
- `MODE mode() default MODE.MIN;`  
Define si el valor del atributo referenciado debe ser menor o mayor.
- `DRValidateStringPattern` es la anotación que define el comportamiento de la validación de formato a cumplir en un atributo `String`, la validación se hace mediante un `matches` a la expresión regular inscrita.
  - `String pattern();`  
Patrón de formato a verificar, se espera una expresión regular.

### ***Validador personalizado.***

En `dr4zk` se puede implementar una validación personalizada aun cuando la validación `DRValidateBussinessResult` no resuelva la validación deseada. Para construir su propio validador siga los siguientes pasos:

1. Crear una clase que defina el comportamiento de la validación, esta clase debe implementar la interfaz `IDRValidator`. Esta interfaz le pedirá implemente el método `validate`, en este debe de definir como se ejecutara la validación, si esta es exitosa, el método debe regresar un valor nulo; de lo contrario regresará el mensaje que se visualizará en la pantalla asociado al componente que describa el atributo en el modelo de vista.

```
public class DRStringPatternValidator implements IDRValidator<DRValidateStringPattern, String> {

 public String validate(DRValidateStringPattern drInput, String name, String value, Object dtoValue, String
 labelKey, Component parent, FormActions action) throws Exception {
 if(value ==null){
 return null;
 }

 if(action.equals(FormActions.SEARCH) && !drInput.applySearch()){
 return null;
 }
 }
}
```

```

 if (!((String) value).trim().equals("") && !((String) value).matches(drInput.pattern())) {
 Object[] param = {Labels.getLabel(labelKey)};
 return Labels.getLabel("dr.forms.label.pattern", param);
 }

 return null;
 }
}

```

2. Crear una nueva anotación (`@interface`) que describa los parámetros necesarios para ejecutar la validación. Por último asigne a la validación creada la clase que hizo primeramente mediante la anotación

`@DRValidate(itemValidate=clase validadora creada) .`

```

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@DRValidate(itemValidate=DRStringPatternValidator.class)
public @interface DRValidateStringPattern{
 /**
 * patron de formato a verificar, se espera una expresion regular.
 */
 String pattern();
 boolean applySearch() default false;
}

```

Su anotación esta lista para ser invocada por dr4zk. Solo anótela en aquellos atributos de la clase del modelo de vista que lo requiera.

## Componentes

Dr4zk implementa los controles más comunes otorgados por ZK, adicionalmente a esto como ya se explicó, se puede construir un componente a la medida. En el común de los casos las anotaciones se llaman igual que las etiquetas y clases de los componentes en ZK anteponiendo el prefijo DR.

### **DRTextbox**

Esta anotación implementa un `Textbox` de ZK, el atributo de la clase de vista debe ser un tipo `String` para poder asignar el valor. Las propiedades más comunes de este control fueron agregadas a la anotación.

- `int maxlenght() default DefaultValues.NO_LIMIT;`  
Máximo número de caracteres permisibles en el campo de texto.
- `int cols() default 40;`  
Número de columnas a mostrar en el campo de texto.
- `boolean readOnly() default false;`  
Define si es de solo lectura.
- `int rows() default DefaultValues.ONE_ROW;`  
Número de filas en el campo de texto.

- `TxtType type() default TxtType.text;`  
Tipo que define si el texto será visible o oculto.
- `boolean uppercase() default false;`  
Define si el texto introducido será transformado a mayúsculas automáticamente.

## ***DRAttachList***

Dr4zk otorga un componente que funciona para cargar archivos en una lista. Se trata de un `Listbox` personalizado al cual se le pueden agregar y quitar elementos que representan un archivo cargado. La lista de resultados entregada dentro de una instancia de la clase de vista deberá ser manejada en el método de negocio para la finalidad que se requiera. Las propiedades de la anotación son las siguientes:

- `int maxRow();`  
Determina el numero máximo de archivos cargados.
- `Class itemRenderer() default DRUploadFilesRender.class;`  
Generalmente no se debería cambiar. Es la clase que define como será dibujado el control como lista.

El tipo de atributo anotado en la clase de vista debe ser una lista de elementos tipo `DRMedia`. Los atributos disponibles en la clase `DRMedia` son los siguientes:

- `private String name;`  
Nombre del medio cargado.
- `private Media media;`  
Contenido del archivo cargado.
- `private DRAttachMedia bo;`  
Objeto que encapsula los datos originales del archivo cargado.

Adicionalmente es necesario que el atributo de la lista en la clase de vista se anote con `DRIsMedia`. Esta anotación evita que dr4zk trate de mapear directamente el valor a un atributo de la clase de negocio definida en `DRRootEntity` sin que antes ocurra una transformación previa.

La clase que maneja la información del archivo a nivel de negocio debe implementar la interface `DRAttachMedia`. Lo anterior es necesario para que dr4zk convierta entre el objeto `DRAttachMedia`(objeto de negocio o persistencia) a `DRMedia`(objeto de vista) cuando se traduce

la información al ser consultada en una acción EDIT, los métodos a implementar son los siguientes:

- `public String getName();`  
Es el *getter* del nombre original del archivo.
- `public String getFormat();`  
Es el *getter* de la extensión del archivo.
- `public String getContentType();`  
Es el *getter* del tipo mime del archivo.
- `public String getUri();`  
Es el *getter* de la *uri* de la obtención del archivo. Ruta del sistema de archivos desde la cual se podrá recuperar el archivo cuando ya ha sido cargado y guardado correctamente.

Un ejemplo de cómo implementar este componente es mostrado en la aplicación dr4zkdemo.

### ***Drcaptcha***

Esta anotación se usa para especificar un componente de validación de texto basado en imagen. El atributo anotado debe ser un `String` y no necesariamente debe estar mapeado a un atributo del `DRRootEntity`, es forzoso para que la forma no pase sin validar que se anote en el atributo la validación `DRValidateCaptcha`.

### ***DRComboBox***

Define una anotación que representa un `Combobox` de ZK. Esta anotación es aplicable en la clase de vista a un atributo del mismo tipo de clase que los objetos obtenidos en la lista del modelo que llenará sus valores. Sus propiedades son las siguientes:

- `boolean autodrop() default true;`  
Despliega automáticamente la lista de valores posibles al colocar el foco en el componente.
- `MOLD mold() default DRComboBox.MOLD.ROUNDED;`  
Estilo bajo el cual se va a pintar el `Combobox`.
- `Class itemRenderer() default DRComboSimpleRender.class;`  
Clase que indica cómo se va a pintar el componente.
- `boolean buttonVisible() default true;`  
Determina si el botón para mostrar la lista de valores es visible.



- `String model();`  
Llama a la acción de negocio usada para llenar los valores del combo, esta acción debe regresar una colección de datos del tipo de atributo anotado en la clase vista `package.Facade@method` el atributo del modelo que se requiera como valor despliegue del Combobox debe estar anotado con `DRField` con acción `LIST`.
- `String action();`  
Es posible tratar un valor no coincidente con el modelo esta propiedad nombra a la acción que se realiza cuando el valor seleccionado en el combo no existe, `package.Facade@method`
- `String[] modelParams() default {};`  
Lista de nombres de atributos de la clase del modelo cuyos valores serán pasados como parámetros para la acción del modelo.
- `Class dtoResult() default Object.class;`  
Clase resultado

### ***DRDateBox***

Esta anotación representa un componente calendario de ZK. Este componente se debe anotar en un atributo tipo `Date` en la clase del modelo de vista.

- `String format() default DefaultValues.DATE_FORMAT;`  
Text format which will be set visually in the component. The default value is `dd / mm / yyyy`

### ***DRDecimalBox***

Este componente capta un valor numérico como texto. Debe estar anotado a un atributo tipo `Float`, `Double` o `BigDecimal`. Sus propiedades son las siguientes:

- `int maxlenght() default DefaultValues.NO_LIMIT;`  
Máximo número de caracteres permitidos.
- `String format();`  
Formato del texto desplegado como numero por ejemplo `#,##0.00`
- `boolean readOnly() default false;`  
Determina si el campo es de solo lectura.

## ***DRFCKEditor***

Esta anotación representa al componente de edición de texto como *html* del tipo *WYSWYG*. Debe anotarse a un atributo tipo `String` en la clase del modelo de vista.

- `String width();`  
Anchura total del componente.
- `String height();`  
Altura total del componente.

## ***DRGmaps***

Anotación que representa una localización en *google maps*. El tipo de atributo al cual se anota en arbitrario sin embargo se debe poner énfasis en asignar un valor de nombre de atributo válido en sus propiedades `latitude` y `longitude`. Ambos atributos deben estar presentes en la clase de modelo de vista y deben ser del tipo `Double`. Adicionalmente en el *zul* donde se desplegará el mapa debe agregarse el llamado al *js* propiamente de *google* ejemplo:

```
<script type="text/javascript"
src="http://maps.google.com/maps?file=api&v=YOUR_VERSION&key=YOUR_KEY" />
```

Sus propiedades son las siguientes:

- `String width();`  
Anchura del mapa.
- `String height();`  
Altura del mapa.
- `String latitude();`  
Nombre del atributo del objeto vista cuyo valor será asignado como latitud del mapa, este debe ser del tipo `Double`.
- `String longitude();`  
Nombre del atributo del objeto vista cuyo valor será asignado como coordenada longitud del mapa, este debe ser del tipo `Double`.
- `String gmarkContent();`  
Nombre del atributo del cual se obtendrá el texto de la marca en el mapa.
- `boolean showSmallCtrl() default false;`

Determina si muestra un control pequeño

- `boolean showLargeCtrl() default false;`  
Determina si se muestra el control largo del mapa.

## ***DRHtml***

Anotación de solo lectura que incrusta el valor del atributo anotado en el *zul* esperando que este sea un código *html* valido, el tipo de atributo debe ser `String`.

## ***DRImage***

Esta anotación representa un componente personalizado de dr4zk que otorga desde la visión de una sola imagen hasta la rotación y carrusel de imágenes; esto depende del tipo de atributo anotado y los valores de sus propiedades. Si en un el atributo es un tipo `List` y `rotate=true` cambiará la imagen mostrada cada 5seg a todos los elementos de la lista, si `rotate =false` se desplegaran como un carrusel horizontal.

- `String width();`  
Anchura de la imagen.
- `String prepend() default "";`  
Antepone este texto a la ruta donde se localiza la imagen.
- `boolean rotate() default false;`  
Si verdadero, rota cada 5 segundos la imagen a mostrar.
- `String nullimage() default "";`  
Ruta de una imagen por defecto en caso de que en la ruta no exista la imagen. Aplicable para el comportamiento de carrusel (`rotate =false`)
- `String href() default "";`  
*url* a la que redirige en caso de que se de clic sobre la imagen.
- `String[] hrefParams() default {};`  
Atributos y sus valores que se pegaran en la *url* de la imagen como parámetros.
- `int minimages() default DefaultValues.NO_LIMIT;`  
De ser una lista de imágenes se desplegara la lista de imágenes a lo ancho, se refiere al mínimo a mostrar.

### ***DRIntBox***

Esta anotación dibuja un `Intbox` debe estar anotado a un atributo tipo `Integer`. Sus propiedades son las siguientes:

- `int maxlenght() default DefaultValues.NO_LIMIT;`  
Máximo de caracteres permitidos.
- `boolean readOnly() default false;`  
Define si el campo es de solo lectura.

### ***DRLabel***

Con esta anotación el valor (mediante un `toString()`) del atributo será puesto en un `Label` de `ZK`. Esta anotación también puede ser usada dentro la anotación `DRField` bajo la propiedad `label` que sirve para anteponer una etiqueta al componente anotado al atributo

- `String key();`  
Identificador en el *i3\*.properties* de donde se obtendrá la etiqueta.
- `String sclass() default DefaultValues.NONE;`  
Clase *css* que define el estilo.

### ***DRSpinner***

Esta anotación dibujara un `Spinner` numérico y debe estar anotado a un atributo tipo `Integer`.

- `int maxlenght() default DefaultValues.NO_LIMIT;`  
Tamaño máximo de caracteres permisibles.
- `int cols() default 1;`  
Número de columnas.

### ***DRListBox***

Anotación que pintará una lista seleccionable de elementos. Esta anotación debe ir asignada a un atributo con el tipo igual a los elementos contenidos en la lista con la que se llena el modelo. Note que existe dependencia de tipo de clase entre el tipo de atributo anotado, los objetos del modelo, y el `dtoResult`, que se unen para mostrar los atributos del modelo en la lista seleccionable o de resultados. Esta anotación también puede ir en un `DRFellowLink` con acción `SEARCH` y será para este caso la lista de resultados.

- `String id() default DefaultValues.NONE;`  
Identificador del `ListBox`.

- `MOLD mold() default DRListBox.MOLD.PAGING;`  
Estilo del molde para ser pintado. `PAGING` o `SELECT`
- `Class itemRenderer() default DRResultsListSimpleRender.class;`  
Clase que define como pintar las filas del `ListBox`.
- `String model() default DefaultValues.NONE;`  
Acción que determina como se cargara el modelo para llenar las filas del `ListBox`, este método debe regresar una colección de datos. `package.Facade@method`
- `String messageSclass() default DefaultValues.NONE;`  
Estilo `css` del mensaje de respuesta.
- `String[] modelParams() default {};`  
Nombre de los atributos que cuyos valores serán pasados como parámetros de la acción de la carga de modelo.
- `Class dtoResult() default Object.class;`  
Define la clase en la que se define los `DRField` con acción `LIST` utilizado para transformar los objetos de persistencia a este tipo para poderlos pintar en pantalla.
- `boolean header() default false;`  
Determina si el `listbox` llevara o no encabezado en sus columnas.
- `String sclass() default "";`  
Clase de estilos `css`.

## ***Glosario***

**Clase de negocio:** es la clase con la que trabajan los métodos que implementan la lógica de negocio, podría ser que estas clases son directamente las que se persisten en la base de datos.

**Clase de modelo de vista:** es la clase que define el comportamiento de una vista web, son la que se marcan con las anotaciones de dr4zk.