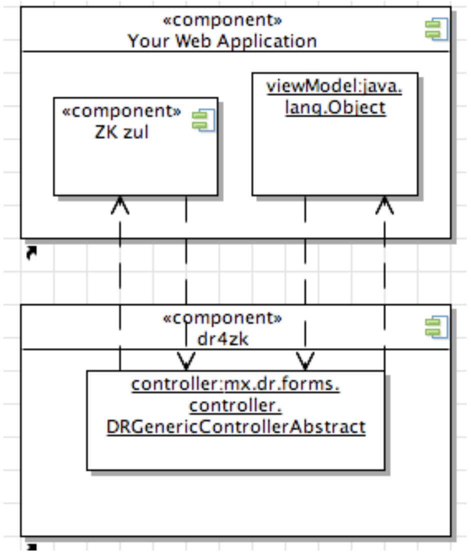
Dr4zk Essential Handbook
Versión 0.9

What is DR4ZK?

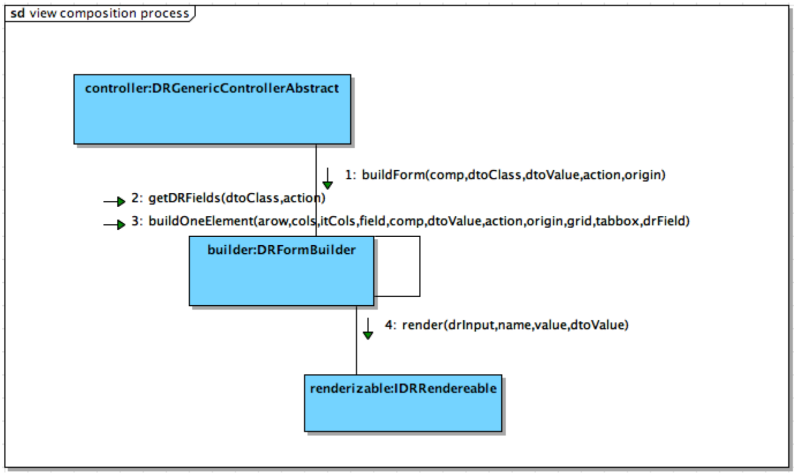
Dr4zk born from the need to have a quicker and simpler way to create and manage the web view from the same definition of the model of components of the view. It is for those developers who prefer to optimize the design activities and save time developing the business process. The basis of operation is the dr4zk ZK framework which allows model and manage the web view in a dynamic and programmatic way ignoring the conventional tags through of a simple *zul* file.

Arquitecture

Dr4zk implements to operate the *MVC* pattern, where the view is handled by *ZK zul* file, the view model is a *pojo* class, which is used to map the values obtained from the components that are drawn on the screen, and finally the driver is a class that extends from `DRGenericControllerAbstract` which processes the events requested by the user. The following diagram illustrates this:

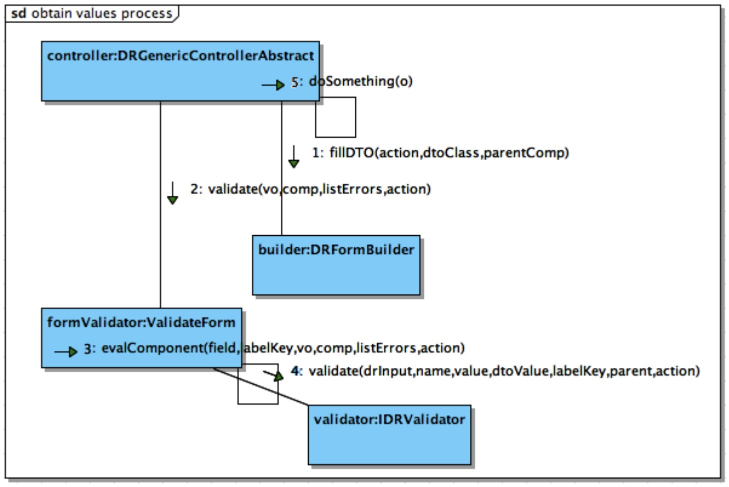

«component»
Your Web Application
«component»
ZK zul
viewModel:java.lang.Object
«component»
dr4zk
controller:mx.dr.forms.controller.DRGenericControllerAbstract

When a user requests a URL from your browser and this view is managed by *dr4zk, dr4zk* read the name of the view model class that contains the configuration of components that are drawed on the screen, you get the controls applicable to the type of action requested (ADD, EDIT, READ, SEARCH) and drawed each of them in the proper place. The following illustration shows the communication diagram of this process:


sd view composition process
controller:DRGenericControllerAbstract
1: buildForm(comp,dtoClass,dtoValue,action,origin)
2: getDRFields(dtoClass,action)
3: buildOneElement(arow,cols,itCols,field,comp,dtoValue,action,origin,grid,tabbox,drField)
builder:DRFormBuilder
4: render(drInput,name,value,dtoValue)
renderizable:IDRRendereable

Process of View Composing. 1

When the user enters values to a form handled by *dz4zk* executing the method `generalAction()` if (ADD, EDIT); or search () in the case of action (SEARCH), the values of the components of the form will be mapped to an instance of the class that defines the view model and will be validated according to the validation that the programmer has scored in the view model. Communication diagram of this process is shown below:

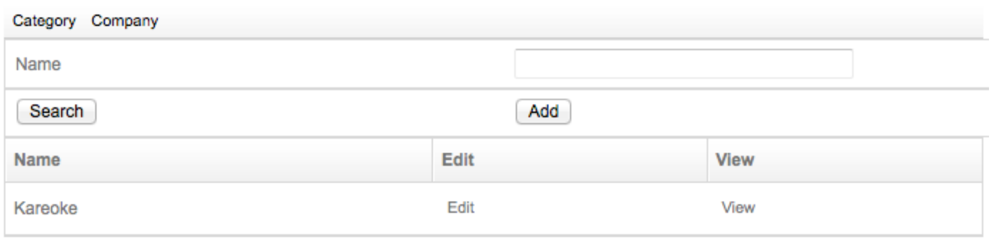

sd obtain values process
controller:DRGenericControllerAbstract
5: doSomething(o)
1: fillDTO(action,dtoClass,parentComp)
2: validate(vo,comp,listErrors,action)
builder:DRFormBuilder
formValidator:ValidateForm
3: evalComponent(field,labelKey,vo,comp,listErrors,action)
4: validate(drInput,name,value,dtoValue,labelKey,parent,action)
validator:IDRValidator

Starting with dr4zkdemo

Dr4zkdemo is a web application that demonstrates the overall functioning of *dr4zk* and will guide the developer. This demo can be obtained from its source code using the link https://github.com/jorgemfk/dr4zkdemo, or you can download the source code packaged in a war file through http://www.dr4zk.org (recommended). It is a common multilayer architecture based on the presentation layer in *ZK* and *dr4zk*, and in the business logic layer Spring and Hibernate like the persistence layer.

SEARCH Category

The project consists of two modules dr4zkdemo that exemplify the use and benefits of using a web application dr4zk. Your goal is to simulate the need for a directory of companies fall into different categories. The first module is *category* which manages the categories they belong to various registered companies. This module consists of the following: ADD, EDIT, SEARCH and READ. When we choose the Category menu option that shows is a screen where you can search our categories, edit, query an existing or add a new one.



Name	Edit	View
Kareoke	Edit	View

Let's review the *index.zul* file, which directs us to the page shown above. Note that when we select an option we are doing to include a page in a url that is parameterized by two parameters as follows:

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

These parameters action refers to the action that will be used in our model of view (SEARCH for this case) and will be treated under this class action dr4zk controller. The parameter defines the class dto_class detailing our view model ie the components that will be drawn on the screen: (`mx.test.dr4zkdemo.view.model.RegisterCategory` in our case).

Let us see how does our `genericFind.zul`, representing the search screen shown above. Note the simplicity and that is not defined any tag except button control with `id="search";` this identifier will execute the `search()` action of our controller `DRGenericControllerAbstract` abstract. For this example uses a generic controller class `DRGenericController` included in *dr4zk*. If you require special functionality provided our custom class must extend the abstract `DRGenericControllerAbstract.`

```
<zk 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>
</zk>
```

Let us now our class view model class `RegisterCategory` focused on the first action SEARCH; we will analyze each annotation present in this class.

```
@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.cla
ss, dtoResult=RegisterCategory.class))})
```

At first, `DRActions` found in the head of our class: this entry determines the action list `DRFellowLink` that our controller will run when an event occurs and what response to send link (another common page). Within `DRFellowLink` properties we have:

- `loadOnInit=true` indicates that the listbox search results will be loaded with an initial consultation, `header=true` makes visible the titles of the result columns of the *listbox*

- `id="resultado_categorias"` is the identifier used by dr4zk to identify the Result listbox

- `itemRenderer=mx.dr.forms.zul.DRResultsListRender` specifies the component to be used to write the results in the listbox. *Dr4zk* gives some generic renderers but you can use any custom.

- `dtoResult=RegisterCategory.class` defines the class that will be used as reference to convert persistent objects to view objects which *dr4zk* handle.

- `submitAction` is the action that will respond when you press the button with `id="search"` of our zul, it will be the definition of `DRFellowLink` with `FormActions.SEARCH` action as shown in the code will run `findByExampleDTO business action called from CatalogFacade`. This action will take the value added to our text component name and use it as a search filter.

- ***Note: It's important mentioning that all the actions defined in dr4zk recommended to be called by a facade, because dr4zk try to make an instance of this class to execute the action method.***

When the results *grid* is presented each of the search results may contain one or more actions parameterized to the asset; `DRFellowLink` use the LIST action as shown below in the two items configured: link action is to edit the item and the second is for reference.

```
@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"` is the identifier to pass to the following link as a parameter to find the item you want to edit or view.
- `url="/registerCategory.zul?dto_class=mx.test.dr4zkdemo.view.model.RegisterCategory&action=EDIT"` defines our next link to post the request.
- `componentPath` defines where the content page will be replaced, this component is an absolute path and generally refers to a component; *include,* `componentPath` is necessary if you require a `fellow=FellowType.SELF`, this means that the page replace url definite answer in light of our current view without refreshing the whole display.
- `listLabel=@DRLabel` defines the tags that will be written in the league for the given record.

In action SEARCH, ADD and EDIT components must be generated within a container, in most cases using a DRGrid as the annotation shown below. Note that is defined clas-level, and indicating that all components defined within this class (noted attributes) are within the grid.

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

Then we see the only field that shows in our search screen called *name*, which has an annotation`Drfield`. This tells us that this component will be drawn by *dr4zk* within the *grid* with the given *label* prefix defined component

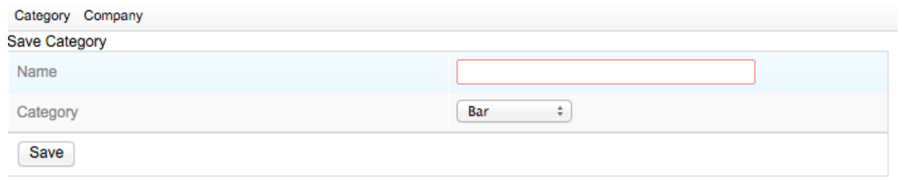
- `order` defines the order sheet will be drawn on the screen over other controls.
- `actions` defines actions which attribute will be visible to the driver..
- `searchOperator` is used as a support to the business layer we use this for the filter (optional); dr4zkdemo includes a sample to may be treated as though it depends on the architecture as it been designed.

`FormActions.LIST` action indicates that this attribute will be visible as a column in the listbox result, the value that you will receive the name attribute of a textbox as shown in the annotation.

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

ADD, EDIT Category

Now see what happens when we want to add (click the add button) or edit an element (click the edit link to edit item) gathered for this analysis because these functions are very similar.



The link that take the ADD is found in the zul genericFind.zul and is given as shown below.

```
<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=m
x.test.dr4zkdemo.view.model.RegisterCategory&action=ADD");
   }else if
   …
   }
    ]]>
 </attribute>
</button>
```

Consider the *zul* file `registerCategory.zul`. In this zul use the same driver as in the search, which makes the difference is that the only button this now points to `id=generalAction`, action is also in our abstract controller `DRGenericControllerAbstract` where our controllers inherit.

```
<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>
```

Now look at the view model class defined in the same class RegisterCategory, but this time we will focus on the actions ADD and EDIT. The `DRFellowLink action = FormActions.ADD`

- `url` indicates the page response sent after business perform the action defined in `submitAction`; `fellow=FellowType.SELF` es the mode like the page will be showed

- `submitAction` is the action to execute when we push the save button with `id=generalaction`

- `successMessage=@DRMessage()` is the message that will be displayed when the business operation has completed successfully, this message will be a *Messagebox* de *zk*.

For the EDIT action is defined the same also the property `param`, which is the identifier previously sent in `FormActions.LIST` action. This identifier is used by the method defined in the `paramAction` property, this business defines the action to be used to retrieve the element with the given identifier. Remember that it is recommended that all business activities to be accessed through a facade. We use the same container that the search grid. Finally `DRRootEntity` annotation is the persistent class type (usually) to retrieve the data or save the data. This annotation is essential actions READ, EDIT and ADD for the transformation between persistent objects and model objects of sight. For more information refer to the mapping of attributes.

```
----
@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{
```

ADD and EDIT actions share components as the attribute *name* as we saw it will be drawed as a *textbox*. Let it also has some validations applicable for actions ADD and EDIT.

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

- `DRValidateNotEmpty` is a dr4zk validation that can be used to order the capture field as mandatory.

- `DRValidateBusinessResult` is another validation that allows you to implement custom validation business, in this case validates that the given name exists not previously recorded in the database.

- `liveValidate=true` in `DRField` allows validations are executed when the component loses focus, otherwise it is validated until you make the submit of the entire form.

The attribute or category *father* will be drown as a selectable and drop-down list with the annotation `DRListBox` with `mold=DRListBox.MOLD.SELECT`. This type of components need a model to fill the different options in the list. It is a collection of objects obtained from `model="mx.dr.ml.view.facade.CategoryFacade@findMainCategory"`. . This method returns the list of possible options that can make the list.
(Note that the data type is a `MainCategory` receiving it to be the type of the elements in the model list.)

```
	@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;
```

The *id* attribute is recorded as a component *intbox* and this will be visible under EDIT and READ actions but read-only mode.

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

EstatusEnum attribute is assignable only under the action EDIT.
Note that the list of the model is a list of *enum* type as the attribute type to assign.

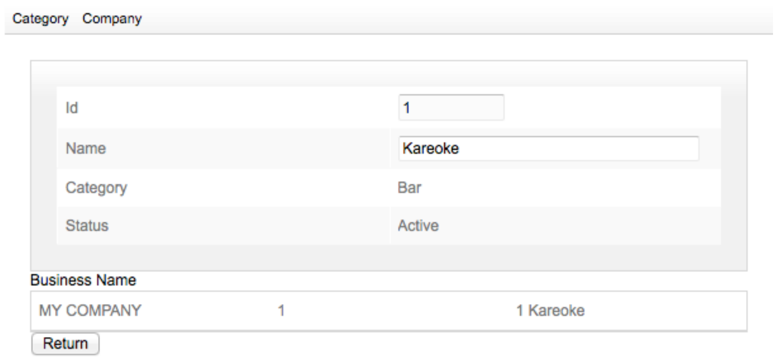
```
	@DRField(label=@DRLabel(key="catalogo.status"),order=6, actions={FormActions.EDIT})
	@DRListBox(mold=DRListBox.MOLD.SELECT, itemRenderer=mx.dr.forms.zul.DRResultsListSimpleRender.class,
	model="mx.dr.ml.view.facade.CatalogFacade@getCatalogStatus")
	@DRValidateNotEmpty
	private CatalogStatus estatusEnum;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

The attribute `will$not$pass` is just used to demostrate that is not necessary all of the class attributes to be mapped, additional attributes can be put to use for another purpose.

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

READ Category

Action is accessed from the link READ defined in `FormActions.LIST` action of one of the search results categories and shows the following reading screen.



Id	1
Name	Kareoke
Category	Bar
Status	Active

Business Name

MY COMPANY	1	1 Kareoke

The view class used remains the same `RegisterCategory` and zul is `viewCategory.zul`. Note that we use the same generic driver `DRGenericController` and now the important thing is to distribute READ action page as we want the user see and assign identifiers containers we use to insert our component in the right place.

```
<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>
```

The kind of view into READ action does not seek a general container was `DRGrid` above, now check each attribute or specify the identifier of the container in which to place it. READ action must be defined in the head of the class by a `DRFellowLink` and only requires two parameters.

- `param = "id" is the name of the identifier of control parameter sent in the previous page (defined in this case in the search result action FormActions.LIST)`

- `paramAction` is the business action that will be used to find the desired item reading.

Finally we have the DRRootEntity annotation, persistence class which get mapped values in our view class .

```
@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 …
```

The mapping attribute values takes place between the view class and defined persistence `DRRootEntity` annotation is given by the names and types of attributes. These must match to be taken by the mapping and so dr4zk copies the values from one to another. If the depth of the objects attributes is greater than one, the separation between names will be through the $ character, and the type of the attribute should be the last to be obtained.

Let's see the anotations in the READ action attributes beginning with the annotation `DRField`. The property `readParent` is the container where you place the embedded component as a child, in case of define a `DRLabel` (not mandatory), this will be prepended to control annotation is essential component.

Our attribute *name* will be drawed as a *textbox* editable only for illustration purposes and it will be drawed within the component with `id="nameRow"`.

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

The id will be set as a read-only intbox within the component with id = "KEYRow".

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

The attribute `estatusEnum$labelDescription` and `father$name` will be drawed as a *label.* Note that there is depth in the attribute mapping, which means that `estatusEnum$labelDescription` will be allocated from the attribute `labelDescription` of attribute `estatusEnum` from object `Category` consulted; and `father$name` will be assigned from the father attribute *name* attribute of Category.

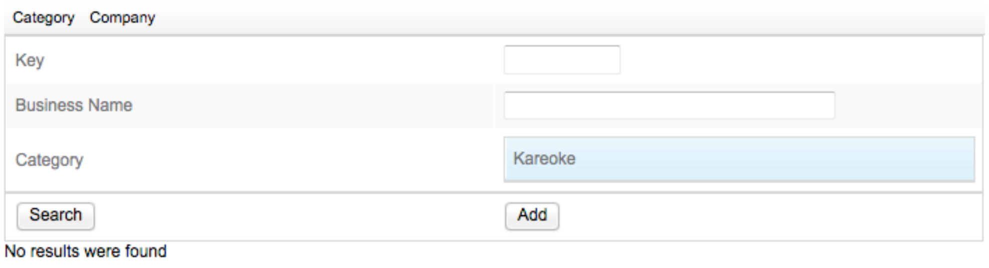
```
        @DRField(actions= FormActions.READ, readParent="status", label =
                        @DRLabel(key = "catalogo.status"))
        @DRLabel(key = DRLabel.NO_LABEL)
        private String estatusEnum$labelDescription;



        @DRField(actions= FormActions.READ, readParent="categoryRow", label =
                        @DRLabel(key = "categoria.padre"))
        @DRLabel(key = DRLabel.NO_LABEL)
        private String father$name;
```

Finally, companies will be drawn as a list of companies that belong to the category searched. This attribute is not mapped to Category, its value will be assigned by the model defined in the business action:
`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"},dtoRes
ult=SearchCompanyMain.class)
        private List companies;
```

SEARCH Company

The second module called **company** is slightly more complex than category, and it's closer to a real case that we could present our enterprise web application. Screen for action SEARCH is as shown below.



Key	
Business Name	
Category	Kareoke

No results were found

The source code for this is in the zul file `genericFind.zul`. Note that this page is the same as that used category, **this demonstrates the reuse of common pages which is possible in dr4zk even if it is the same type of view model class.**

```
<zk 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 …
         }else if(dtoClassName.equals("mx.test.dr4zkdemo.view.model.SearchCompanyMain")){
          ((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>
```

For the module company shares are spread over different classes unlike category view where one configuration containing all the actions.

The class action is called `SearchCompanyMain` SEARCH. Analyzing the properties of the annotation `DRFellowLink` in this class we can see that this time it will be called an initial consultation to load the page this is given by `loadOnInit=false` which is the default value. Recall that our results will be converted to the class defined in `dtoResult=SearchCompanyMain.class`, whence it takes to drawed the attributes defined in the annotation columns with action `DRField` 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))
```

The annotations `DRFellowLink` with LIST action, now a different way to invoke the response page this time will be a popup (`fellow=FellowType.POPUP`), if you edit the item, and a new page (`fellow=FellowType.NEW`) in for consulting. The other properties are already described in the module category. Remember that it is necessary to note the persistence entity that represents the view model
`@DRRootEntity(entity=mx.test.vo.Company.class)`, and container (`DRGrid` for this case).

```
 @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{
```

The attribute `brand` or company name in `SearchCompanyMain` be drawn as a *textbox*, and id like a *Intbox*. Note that this time we present an annotation validation we want to run under the action SEARCH, is done by specifying the property `applySearch=true`.

```
    @DRValidateNotEmpty(applySearch=true)
    @DRField(actions= {FormActions.SEARCH, FormActions.LIST},
label=@DRLabel(key="empresa.nombre"),order=2,searchOperador=DRField.Operator.LIKE)
    @DRTextBox(maxlenght=50)
    private String brand;


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

The `category` attribute is drawed as selectable list (`MOLD.PAGING`), ie, all items will be displayed in the list, this is the default template.

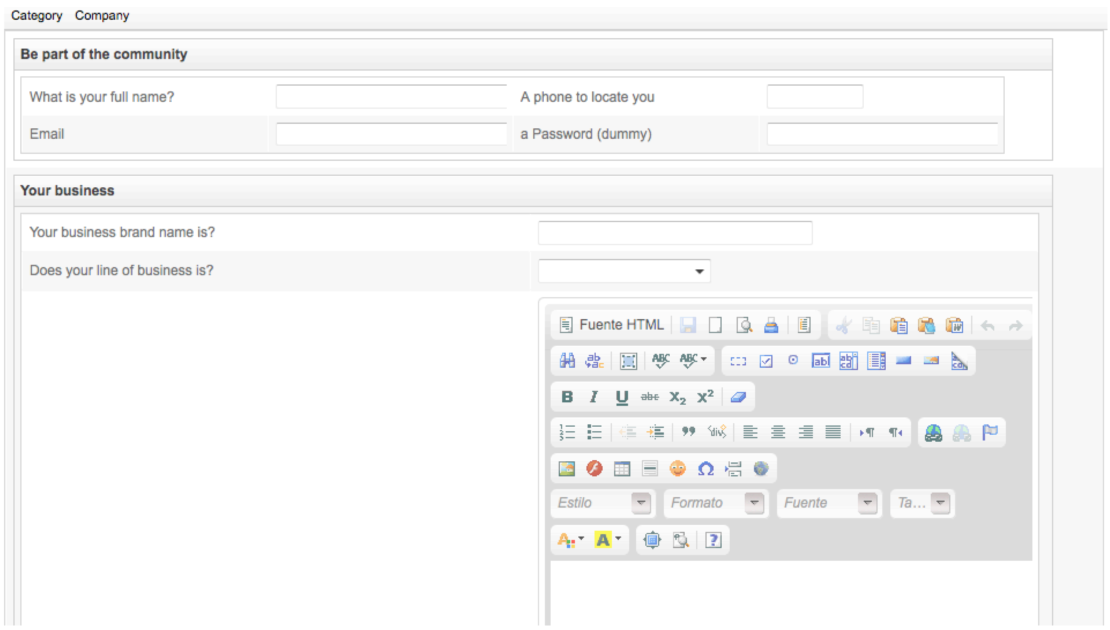
```
    @DRField(actions= {FormActions.SEARCH, FormActions.LIST},
label=@DRLabel(key="categoria.padre"),order=3,searchOperador=DRField.Operator.EQUALS)
    @DRListBox(model="mx.dr.ml.view.facade.CategoryFacade@findActive")
    private Category category;
```

`brand`, `id` and `category` will be drawed as search filters (`FormActions.SEARCH`) and as columns in the *listbox* of results (`FormActions.LIST`). Recall also that the operators of filters (`searchOperador=…`) are just support and serve as a guide for business method to conduct the search action; dr4zk not strictly required.

Recall that the order will be drawed components property is given by the order.

ADD, EDIT Company

Access the ADD action screen by clicking on the *Add* button on the search screen, this screen allows us to register a new company.



Be part of the community

What is your full name?		A phone to locate you	
Email		a Password (dummy)	

Your business

Your business brand name is?	
Does your line of business is?	

The source code is in the file `registerCompany.zul`, we see that is equally simple components, there is only a button with `id="generalAction”` that will allow us to save the data entered by the user company. Notice that in the *iframe* tag there is an inclusion of another section or screen dr4zk handles, it is the user agreement, which will not be discussed in this document in its simplicity, but is mentioned to show the coexistence of different sections dr4zk handled by a single screen view.

```
<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&amp;action=READ&amp;type=E"/>
    </row>
    <row >
     <button id="generalAction" label="Save" />
    </row>
   </rows>
  </grid>
 </div>
```

```
</zk>
```

The view class `RegisterCompanyMain` is used for ADD action, we can see that there is no additional parameters to those already seen in class. It should be noted that our container is a *grid* of a column (`DRGrid(cols=1)`), and the kind of persistence for the mapping of the transformation of objects and persistence viw is now:
`DRRootEntity(entity=mx.test.vo.Company.class).`

```
@DRFellowLink(action=
FormActions.ADD,fellow=FellowType.SELF,componentPath="//main/myInclude",url="/genericFind.zul?dto_class=mx.test.dr4zkdem
o.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 {
```

In class attributes `RegisterCompanyMain` see something different from our previous cases, is that dr4zk allows the inclusion of containers within a parent container, this is reflected in a composition class. For this example our children are `DRGroupBox` container and all components listed in the contained class attributes will be included in the label property *groupbox* and will be the title of the *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;
```

A class contained in the above composition is `RegisterCompany`. We are not required to record the `DRActions` back, `DRFellowLink` or `DRRootEntity`, as these are previously read from the wrapper class only is required to specify the container for the components included in this class, so this case a `DRGrid`.

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

The common notations on the attributes that describe the components of the display of them have already been discussed in class, we review only new components or properties not mentioned above.

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

The attribute *category* is the category we assign to our company, and this is assigned by a *Combobox* component. This like the *listbox* requires a model, which is obtained by the method written in the property `model=mx.dr.ml.view.facade.CategoryFacade@findActive.`

`DRCombobox` has a property `action=mx.dr.ml.view.facade.CategoryFacade@save`, which defines what to do when the value given by the user is not on our list of model options. For this specific case, if the description does not coincide, it is saved as a new record category.

```
    @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;
```

The media attribute makes `DRAttachList` component, which draw a custom component dr4zk to upload and download files, with property `maxRow` = 1 defines the maximum number of files to attach, in this case only accepts 1. Note that the type attribute is not mapped to a type `DRRootEntity (Company.class)`, since this should be a list of objects of class `DRMedia` even when the element is just one, and to have a special treatment records; is essential `DRIsMedia` annotation.

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

The *link* attribute that expects a valid *url* using a *textbox*, write a validation type to do this `DRValidateStringPattern`, this annotation validates that the value obtained has the specified pattern.

```
    @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(maxlenght=99)
    private String link;
```

Now review the class to use the EDIT action, this is EditCompanyMain, the intention of dividing the different view class for ADD and EDIT action is to demonstrate the different containers available, as in action EDIT containers are used: one `DRTabBox` which *tabs* will create a structure to contain their attributes.

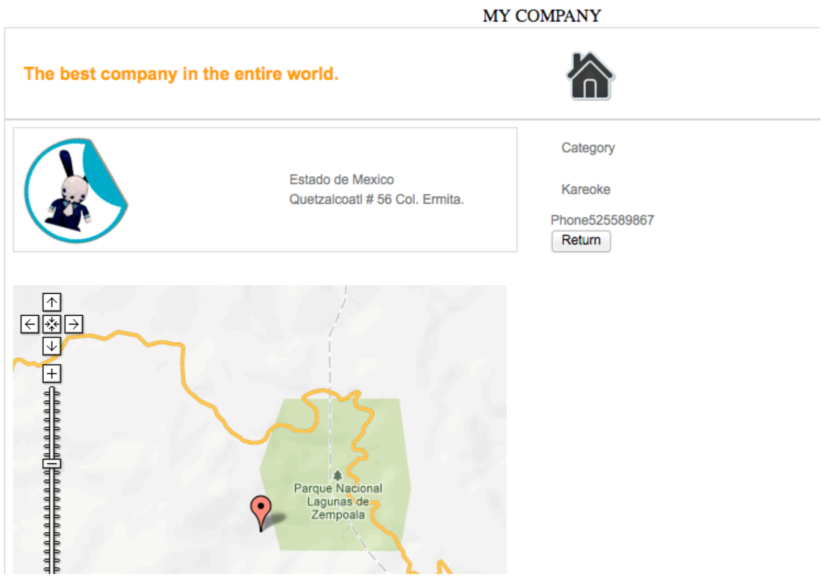
```
@DRFellowLink(action= FormActions.EDIT,fellow=FellowType.POPUP,
componentPath="//main/myInclude",url="/genericFind.zul?dto_class=mx.test.dr4zkdemo.view.model.SearchCompanyMain&action=S
EARCH",
        submitAction="mx.dr.ml.view.facade.CompanyFacade@updateCompany",
        param="id",
        paramAction="mx.dr.ml.view.facade.CatalogFacade@boById",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

Each one of this attributes is a *tab* with title given by the `label` property `DRField` annotation. Note the reuse of classes that were used in the action ADD (`RegisterCompanyAddress`, `RegisterCompanyContact`), this demonstrates the easy reuse of view code in dr4zk.

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

After searching for a company and when you click on the link will take you to *view* the screen that controls the action READ, which is a custom page to display the data of the company as we want see our customer. The code for this page can be seen in the file `viewCompany.zul`


MY COMPANY
The best company in the entire world.
Estado de Mexico
Quetzalcoatl # 56 Col. Ermita.
Category
Kareoke
Phone525589867
Return
Parque Nacional Lagunas de Zempoala

To READ action, `ViewCompany` class is not very different from what already mentioned in the action category for READ. Recall that `readParent` property `DRField` indicates the name of the container component identifier which draw our component scored.

```
@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 that the *id* attribute is not annotated with any component, this is because it is the identifier that is used as a parameter to be recovered from `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` will draw a picture from the path defined therein, the retrieved value of the `uri` attribute value of `logo` from `DRRootEntity (Company.class);` this route prefix will be placed on the property since `prepend` = "."

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

`DRGmaps` noted in the `address$latitude` attribute will draw a map component of *google maps*. Note that refers to more than one attribute of the class through their properties `latitude = "address$latitude", longitude = "address$longitude", gmarkContent = "Brand"`, so they should be valued attributes in class , otherwise an error is generated. Finally this annotation requires a valid *google* key embedded in the *zul* by *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;
```

The custom annotation `DRLink` attribute link is explained in the section called custom control.

.

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

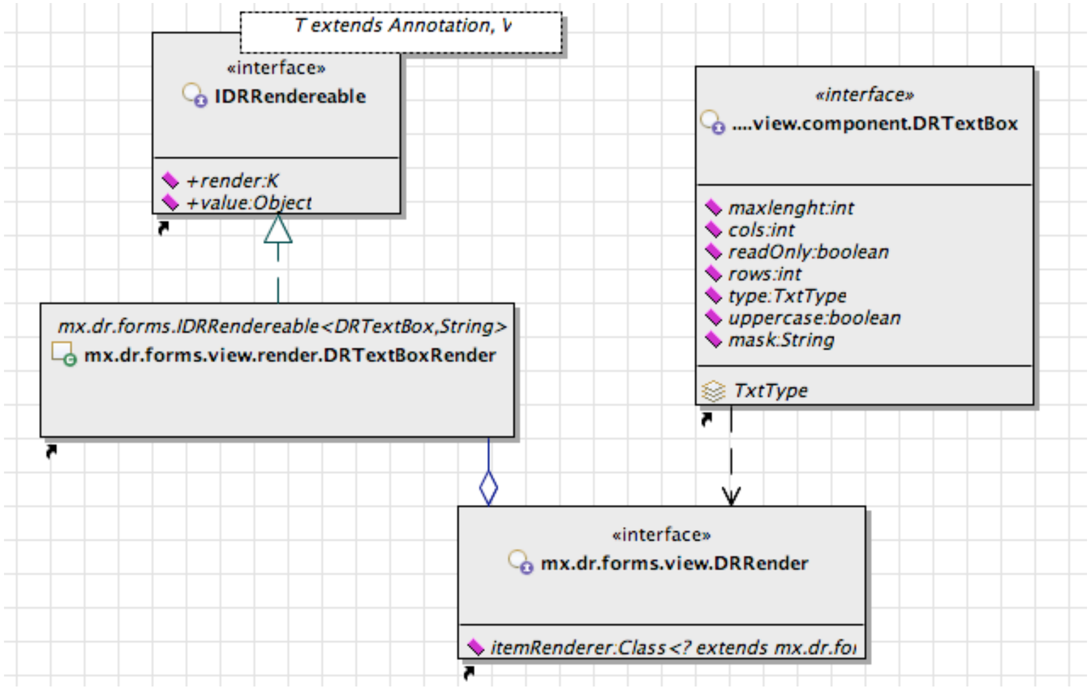
With the above we view the dr4zk capabilities for generating web views in a business application. The following sections below describe in detail each part of the annotations and dr4zk classes.

Customized Control.

Dr4zk allows custom classes implementing validation controls or custom classes, both essentially follow the pattern shown in the following image. We will take the case of a custom control, for it must do two things:

1. - Create a class that implements the interface `IDRRenderable`. By implementing its `render` method, tell us how you will draw the component on the screen and tell you how we `value` method to retrieve the value of the component to be assigned to the attribute scored.

2. - Create an annotation, this defines the parameters we have to draw our component and will be the one record in the attribute of our view class. DRRender score is essential in this class, as it will be referred to our class created in step 1.



Recall the case of dr4zkdemo, and suppose we want to create a read-only component to place a link in an image that will be set to the URL of our attribute somehow. `DRLinkRender` class implements the solution on how to draw the component on the screen. Note that the method returns no value because this value is read-only and instead just read the attribute value scored. The *render* method for this specific solution creates a `Toolbarbutton` and placed as url the value obtained from our valued view object (exclusive to READ or EDIT action) and placed an image to ToolbarButton *image* taken from the property defined in the `DRLink` annotation also create.

```
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` is the anotation that solves the need simplely, its only property is image that is the absolute path to the image to display, this will be used as link icon. In the definition of the annotation note annotation is present `DRRender (itemRenderer = DRLinkRender.class)`, this is the link between the entry and the previous class, ie what kind specified here will be used to draw the component and how to get the value of the component.

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

  String image();
}
```

With the above dr4zk read this notation we have done and will drown the specified component in our screen, an example of how to deploy an attribute is as follows:

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

Actions

Access to dr4zk functions to deploy the view in an expected way is through the zul's url, where the driver is managed by dr4zk. This url determines the behavior of view through two mandatory parameters.

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

- dto_class: The view class handled by dr4zk where is annotated the behavior of the different actions and components that are visible to each action.

- action: This is the behavior invoked of dto_class. This action may be: SEARCH, ADD, EDIT, READ. If not specified the default will be treated as ADD.

It is recommended that the *url* is placed within a component `Include`, this is to be transparent to the end user, the class and actions used in the display.

One or more actions can be annotated in the view class, if is required only one can be supported directly by `DRFellowLink` annotation; else you may use `DRActions` annotation.

DRActions

Is an annotation that allows a collection of DRFellowLink where each defining a different action. The actions defined must be unique per class, for example should be defined only one action ADD; LIST action is an exception to this if it can be defined repeatedly. The unique property of the annotation is:

- `public DRFellowLink[] actions();`
  Set of actions to handle the view model.

DRFellowLink

Annotation that defines a behavior class action in our view, this annotation is placed in the class definition. There are five types of actions defined, and are given in the enumeration FormActions, these are: SEARCH, ADD, EDIT, READ and LIST. There are properties common to all of them and there are others that execute to a specific action. The following table defines properties that properties can be used in each action.

Property	Description	ADD	EDIT	READ	SEARCH	LIST
`public FormActions action();`	Actions: SEARCH, ADD, EDIT, READ and LIST	X	X	X	X	X
`String componentPath() default DefaultValues.NONE;`	Absolute path of the page and component of type `Include` specific which will load the new screen section, only required for `FellowType.SELF`. An example path is: `//main/myWindow/myInclude`	X	X			X
`FellowType fellow() default FellowType.NEW;`	Defines how will be presented to the user as the response *url* default action will be in a new screen.	X	X			X
`String submitAction() default DefaultValues.NONE;`	Absolute reference to the name of the business method to be executed after the action, it is recommended to use the facade pattern as *dz4zk* to create an instance of the class	X	X		X	

	defined for performing the method, the name format is as follows: `your.package.YourFacade@actionMethod`					
`String url() default DefaultValues.NONE;`	This will be the URL where you direct the action response to be made	X	X			X
`String[] sessionParams() default {};`	Contains the identifiers of the objects to be extracted which are contained in the map http user session.	X	X	X	X	X
`DRMessage successMessage() default @DRMessage;`	Message to display when the action is performed successfully.	X	X		X	
`@DRLabel listLabel() default @DRLabel(key=DefaultValues.NONE);`	For the exclusive case of an action in the list of results, this is the label that should be the link.					X
`String param() default DefaultValues.NONE;`	Is the name of the view object attribute to be passed as an input parameter for the recovery of the required record in the action EDIT, READ or LIST		X	X		X
`String paramAction() default DefaultValues.NONE;`	Action to retrieve the result. It uses as a parameter the value of `param` attribute defined above, this applies to actions EDIT, READ or LIST. It is recommended that this is a method of a facade: `pakage.FacadeCla`		X	X		X

	ss@method					
`DRListBox resultsComponent() default @DRListBox;`	Is the definition of `Listbox` which will be showed the search results, this is exclusive action SEARCH				X	
`boolean loadOnInit() default false;`	Determines whether there will be pre-filled in the results of `Listbox resultComponent` This is exclusive or the SEARCH action. The `submitAction` must have a default behavior.				X	

Mapping attributes of business classes and view model classes.

Dr4zk is able to copy the value of the attributes of a view class to business class (commonly persistence) and back to simulate an object processing business object view. In READ and EDIT actions is used this procedure to obtain the values of the attributes and controls that we will show on screen, business class to the view class. And in the case of ADD and EDIT go from view to a business object and is used to retrieve the valued business objects that we will use in business logic and in ordinary cases end with a persistent action. For this, determine what kind of business is displayed in a specified view class with the annotation `DRRootEntity`.

DRRootEntity.

By marking a view class with this annotation will tell *dr4zk* that the attributes defined in the view class has an analogue in the persistence class specified in `DRRootEntity`, and will copy the values of view model attributes to an object class defined and vice versa (in READ and EDIT actions). The following is an example of its use:

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

In special cases requires map different separate business entities in one view model. There is the option of using multiple mapping business entities to restrict the mapping to the attributes in the view class annotated with `DRRootEntity.` The following illustrates this:

```
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;
}
```

The mapping of attributes between the model view classes and business classes is through the attribute name, this should be the same, in case of composition the desired attribute depth is divided by a $ character. The following is an example of mapping.

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

public class B{
 private A a;
}


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

The result of the transformation of view objects to persistence objects is given as a parameter in the method `doSomething` (for ADD and EDIT actions), and `search` (SEARCH action) of our controller class that implements the abstract class `DRGenericControllerAbstract`.

The result of the transformation of view objects to persistent objects is encapsulated as a list of business objects (or persistence) contained in the object `DRGenericDTOIN` under the attribute `bos`, in ordinary cases this list will have a single element. `viewDTO` attribute is an instance of a view with valued attributes as entered by the user in the form of the screen.

Containers.

The components generated by the annotated attributes in the view class for actions ADD, EDIT and SEARCH must be organized in containers to be annotated in the header of a class mandatorily. This gives form to the presentation of the components, the most common is `DRGrid`. The view model classes annotated with `DRGrid` will build a set of defined components into a *ZK* grid, a column ZK defined by their `label` and another column by the same component, the properties are:

- `String id();`
  Unique identifier of the *Grid* to create.

- `String width();`
  Is the width of the *Grid*.

- `int cols();`
  Number of columns that have the *Grid*.

The following is an example of its use.

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

Special containers are mentioned as follows..

- `DRTabbox-DRTabpanel` which draw a `Tabbox` in their `Tabpanel` with ZK. For this implementation you must create a main model view class that will be marked by a `DRTabbox`, then each attribute of the class type contained in this main class will be marked by a `DRTabpanel`.

- `DRGroupbox` will divide into segments the sets of components through *ZK* `Groupbox`. Likewise, the main class attributes that are specified as contained components class may be annotated by this annotation to handle groups of components.

The definition of both is very similar, detail and example of its implementation can be found in the section of the dr4zkdemo application analysis.

Validations.

Dr4zk provides a set of basic validations that can be combined on an attribute that represents a component in the view model class that is annotated as `DRField` and respective annotation component (`DRTextbox`, `DRIntbox`, etc). Below are the different types of validation available as annotations.

A common property in them is `applySearch() default false`; default validations only apply for actions ADD and EDIT assigning a true value to this property causes the validation in question also applied under the SEARCH action for the attribute annotated.

- `DRValidateCaptcha` defines the validation behavior matching text in a captcha. This validation must be present to mandatory way in a `DRcaptcha` component and this use is exclusive.

- `DRValidateBussinessResult` defines the validation behavior using the annotated attribute value as an input parameter to a method of business where you will realize the appropriate validation, this annotation subscribe for a business validation on the value entered by the user. An example is a record that should not be duplicated in the database for a given value. This business method must return true if this passes the validation, or false if it doesn't pass. These properties are:

  - `String action();`
    Defines the action to conduct business operations to determine if the value is valid, the return value must be a Boolean: true if validation is true and false if it doesn't
    `package.Facade@method`

- `DRValidateLenght` is the annotation that defines the validation behavior of minimum or maximum length of the value entered, this attribute applies to `String`, `Integer`, `Double` or `Bigdecimal`.

  - `int lenght();`
    Lenght defined value.

  - `MODE mode() default MODE.MIN;`
    Determines whether validation of defined length applied to a value in the minimum or maximum length

- `DRValidateNotEmpty` validation indicating that a field is required in their capture by the user. Normally this validation is not required to any value of these properties, but for the special case of validating values in `Listbox` or `Combobox` the first value of the model should be considered void. An example is a model object with value display "Choose an option", then you need to set the following properties:

- `String property() default DefaultValues.NONE;`
  It is the name of an attribute to validate encapsulated in the object that forms the model

- `String emptyValue() default "-1";`
  The referenced attribute value in `property` that represents an empty selection.

- `DRValidateReferenceDate` is the validation annotation applied to compare the attribute value is higher or lower than another value of another attribute referenced (currently only applicable to dates)

- `String property();`
  Attribute name referenced as a comparison.

- `MODE mode() default MODE.MIN;`
  Defines whether the referenced attribute value must be lower or higher.

- `DRValidateStringPattern` is the annotation that defines the behavior of the meet format validation on a `String` attribute, validation is done using a regular expression `matches` the enrollee.

  - `String pattern();`
    Format-check pattern, a regular expression is expected.

Custom validator.

In dr4zk you can implement custom validation even if `DRValidateBussinessResult` doesn't resolve the expected validation. To build your own validator follow these steps:

1. Create a class that defines the validation behavior, this class must implement the interface `IDRValidator`. This interface requires you implement the `validate` method, in it, you must define how the validation will be executed, if this is successful, the method must return null, otherwise return the message to be displayed on the screen associated with the component that describes the attribute in the view model.

```
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. Create a new annotation (`@interface`) that describes the parameters needed to run the validation. Finally assign validation class created at first through the annotation `@DRValidate(itemValidate=` validator class).

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@DRValidate(itemValidate=DRStringPatternValidator.class)
public @interface DRValidateStringPattern{
    /**
        * format-check pattern, a regular expression is expected.
        **/
    String pattern();
    boolean applySearch() default false;
        }
```

Your annotation is ready to be invoked by dr4zk. Only enter it on those attributes of the view model class that requires it.

Components

Dr4zk implements the most common controls provided by ZK, in addition to this, you can build a component to measure. In ordinary cases the annotations are called as labels and classes of ZK components putting the prefix DR.

DRTextbox

This annotation implements a ZK `Textbox`, the view class attribute should be a `String` type to assign the value. The most common properties of this control were added to the annotation.

- `int maxlenght() default DefaultValues.NO_LIMIT;`
  Maximum number of characters permitted in the text field.

- `int cols() default 40;`
  Number of columns in the text field.

- `boolean readOnly() default false;`
  Determines whether it is read-only.

- `int rows() default DefaultValues.ONE_ROW;`
  Number of rows in the text field.

- `TxtType type() default TxtType.text;`
  Type that defines whether the text will be visible or hidden.

- `boolean uppercase() default false;`

Defines whether the entered text will be automatically converted to uppercase.

DRAttachList

Dr4zk provides a component that works to upload files to a list. This is a custom `Listbox` to which you can add and remove items that represent an uploaded file. The list of results delivered within an instance of the view class should be handled in the method of business for the purpose required. Annotation properties are:

- `int maxRow();`
  Determines the maximum number of files loaded.

- `Class itemRenderer() default DRUploadFilesRender.class;`
  Usually you should not change. It is the class that defines how it will be drawn the control as a list.

The attribute type entered in the view class should be a list of items `DRMedia`. The attributes available in the `DRMedia` class are:

- `private String name;`
  Name of the media loaded.

- `private Media media;`
  Content uploaded file.

- `private DRAttachMedia bo;`
  An object that encapsulates the original data file loaded.

Additionally it is necessary that the attribute from the list in the view class is annotated with `DRIsMedia`. This annotation prevents dr4zk try to map directly the value to a business class defined in `DRRootEntity` without prior transformation occurs first.

The class that handles the file information at business level must implement the `DRAttachMedia` interface. This is necessary for dr4zk to convert between the object `DRAttachMedia` (business object or persistence) to `DRMedia` (view object) when it translates the information to be viewed on an action EDIT. The methods to implement are:

- `public String getName();`
  The original file name *getter* method

- `public String getFormat();`

The file extension *getter* method

- `public String getContentType();`
  The mime type of the file *getter*

- `public String getUri();`
  The Uri of obtaining file *getter*. File system route from which you can retrieve the file once it has been loaded and saved correctly.

An example of how to implement this component is shown in dr4zkdemo application.

Drcaptcha

This annotation is used to specify a validation component based on text image. The annotated attribute must be a `String` annotated and should not necessarily be mapped to an attribute of `DRRootEntity`, it is required for the form to no passing without validate, was annotated on the validation attribute `DRValidateCaptcha`.

DRComboBox

Defines an annotation representing a ZK `Combobox`. This annotation is applicable in the view class to an attribute of the same class type of objects retrieved in the model list that fill their values. These properties are:

- `boolean autodrop() default true;`
  Automatically displays the list of possible values when focus on the component.

- `MOLD mold() default DRComboBox.MOLD.ROUNDED;`
  Style which they will paint the Combobox.

- `Class itemRenderer() default DRComboSimpleRender.class;`
  Class that indicates how you will paint the component.

- `boolean buttonVisible() default true;`
  Determines whether the button to display the list of values is visible.

- `String model();`
  Call to business action used to fill the combo values, this action should return a collection of annotated attribute type data in `package.Facade@method` view class, the model attribute required as `Combobox` deployment value must be annotated with action `DRField` LIST.

- `String action();`
  It is possible to try a mismatch value with the model, this property names the action performed when the selected value selected doesn't exist in the combo, `package.Facade@method`

- `String[] modelParams() default {};`
  List of model class attribute names whose values will be passed as parameters to the action model.

- `Class dtoResult() default Object.class;`
  Class result

DRDateBox

This annotation represents a ZK Calendar component. This component should be annotated in a Date type attribute in the view model class.

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

DRDecimalBox

This component captures a numeric value as text. Must be annotated with an attribute of type `Float, Double` or `BigDecimal`. Their properties are:

- `int maxlenght() default DefaultValues.NO_LIMIT;`
  Maximum number of characters allowed.

- `String format();`
  Text format displayed as number, for example #, # # 0.00

- `boolean readOnly() default false;`
  Determines whether the field is read-only.

DRFCKEditor

This annotation represents the text edit component as an *html* of type WYSWYG. It should be annotated a `String` attribute in the view model class.

- `String width();`
  Total width of the component.

- `String height();`
  Total height of the component.

DRGmaps

Annotation representing a location in *google maps.* The attribute type which is annotated in arbitrary however should emphasize to assign a name attribute value valid in the `latitude` and `longitude` properties. Both attributes must be present in the view model class and must be of type Double. Additionally in the zul where the map will display you must add the call of *js* properly to *google*, for example:

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

These properties are:

- `String width();`
  Map width.

- `String height();`
  Map height.

- `String latitude();`
  Name of the view object attribute whose value will be assigned as the map latitude, this must be of `Double` type.

- `String longitude();`
  Name of the view object attribute whose value will be assigned as the map length, this must be of `Double` type.

- `String gmarkContent();`
  Attribute name which will output the text of the map mark.

- `boolean showSmallCtrl() default false;`
  Determines whether small control is shown.

- `boolean showLargeCtrl() default false;`
  Determines whether large control is shown.
  .

DRHtml

Read-only annotation that embeds the attribute value annotated in the *zul* validating an *html* code, the attribute type must be `String`.

DRImage

This annotation is a dr4zk custom component that gives from the perspective of a single image to the rotation and image carousel; this depends on the type of attribute annotated and their properties values. If an attribute is a `List` and `rotate` = true this will change the displayed image every 5sec to all elements of the list, if `rotate = false` it will be deployed as a horizontal carousel.

- `String width();`
  Image width.

- `String prepend() default "";`
  Prepend this text to the path where the image is located.

- `boolean rotate() default false;`
  If it is *true*, rotate every 5 seconds to display the image.

- `String nullimage() default "";`
  Default image route in the case that the route does not exist in the image. Applicable for carousel behavior (`rotate =false`)

- `String href() default "";`
  Url that redirects if they click on the image..

- `String[] hrefParams() default {};`
  Attributes and values that will be added to the *url* of the image as parameters.

- `int minimages() default DefaultValues.NO_LIMIT;`
  From a list of images will be displayed the list of wide pictures, it refers to the minimum to display.

DRIntBox

This annotation draws an `Intbox.` It must be annotated to an `Integer` type attribute. The properties are:

- `int maxlenght() default DefaultValues.NO_LIMIT;`
  Maximum characters allowed.

- `boolean readOnly() default false;`
  Defines whether the field is read-only

DRLabel

With this notation the value (through `toString` ()) the attribute will be put into a ZK `Label`. This annotation can also be used within the `DRField` annotation under the `label` property used to putting a label to the component attribute annotated.

- `String key();`
  Identifier in the *i3 *. roperties* where the label will be obtained.

- `String sclass() default DefaultValues.NONE;`
  Css class that defines the style.

DRSpinner

This annotation will draw a numerical `Spinner` and this should be annotated to a `Integer` attribute type.

- `int maxlenght() default DefaultValues.NO_LIMIT;`
  Maximum size of allowable characters.

- `int cols() default 1;`
  Number of columns.

DRListBox

Annotation to paint a selectable list of items. This annotation should be assigned to an attribute with the same type of elements contained in the list which is filled the model. Note that there is class type dependence between the attribute type annotated, objects model, and `dtoResult`, coming together to show the model's attributes in the results list or selectable. This annotation can also go on a `DRFellowLink` with SEARCH action and will in this case the list of results.

- `String id() default DefaultValues.NONE;`
  `Listbox` id.

- `MOLD mold() default DRListBox.MOLD.PAGING;`
  Mold style to be painted. PAGING o SELECT

- `Class itemRenderer() default DRResultsListSimpleRender.class;`
  Class that defines how to paint the rows of the Listbox. .

- `String model() default DefaultValues.NONE;`
  Action that determines how the model will load the model to fill the rows of the `Listbox`, this method should return a collection of data. `package.Facade@method`

- `String messageSclass() default DefaultValues.NONE;`
  Css style response message.

- `String[] modelParams() default {};`
  Name of the attributes whose values will be passed as parameters to the action of the load model.

- `Class dtoResult() default Object.class;`
  Defines the class that defines the `DRField` with LIST action used to transform the persistence objects to be painted on the display.

- `boolean header() default false;`
  Determines whether or not the `listbox` take header columns.

- `String sclass() default "";`
  Css style class.

Glossary

Business class: The class which working with the methods that implement the business logic, it could be the classes which persist directly in the database.

View model class: the class that defines the behavior of a web view, which are marked with *dr4zk* annotations.