Developers guide

Developers guide

- Creating a user interface data model
- Binding data model simple values to zk components
- Binding data model context
- Binding Data Model Collections
- Components as a data source
- Data model manipulation
- Building dynamic data model

Creating a user interface data model

There are three alternative ways to implement user interface data model. The first one is to retrieve information from an XML file. Its use is simple and easy to implement during user interface prototyping phase.

A second alternative is to create a set of glue classes, similar to a DAO object, that are responsible for retrieving user interface data from the storage system, and save changes back.

Finally, there is a way to get this glue layer in a declarative way, by means of an XML behavior descriptor.

As an example, a let's build a web page for querying countries and cities, the data will be modeled by the following XML model:

```
<?xml version = "1.0" encoding = "UTF-8">
<my-data>
 <title name = "Countries of the World" />
 <country name = "Spain" abbreviation = "en">
   <city name = "Palma de Mallorca" />
   <city name = "Madrid" />
 </country>
 <country name = "USA" abbreviation = "us">
   <city name = "Wasington" />
 </country>
 <country name = "Deutschland" abbreviation = "on">
   <city name = "Berlin" />
   <city name = "Bonn" />
   <city name = "Hamburg" />
 </country>
</my-data>
```

After creating the data model, you can make use of it from any zul page including a xmlDataSource component:

```
<?xml version="1.0" encoding="UTF-8">
<zk>
 <xmlDataSource id="mydata" src="/my-data.xml" />
</zk>
```

Binding data model simple values to zk components

In the previous section, we saw how to declare a sample data model. To bind a component to a data model element, the bind attribute can be used. The bind attribute value must be composed of the data source path to be used, a colon separator, and the xPath element to the data model element.

The data source is selected based on the zk path. Both simple paths (/model1) or full paths (//page1/window1/window2/model2).

The data model element is specified using its XPath. Internally, it's using an Apache commons JXPath version.

So, in order to display a label with the value of the "name" attribute of the XML document "title" entity, set the bind attribute of a new label the value "/mydata:/title/@name", as shown in the following example:

```
<?xml version = "1.0" encoding = "UTF-8">
<zk>
  <xmlDataSource id = "mydata" src = "/my-data.xml" />
  <label bind = "/mydata:/title/@name" />
</zk>
```

Mind that this binding is only one way, because the label object does not allow the user to change its content. When using a textbox, checkbox or any other input element, the binding will be bidirectional, allowing the user to change the associated DOM XML file. Anyway, the changes will not be serialized to the original XML file.

Binding data model context

To easiest the readability and maintainability of the code, and to shorten in XPath paths, some components can works as a relative xpath context for the contained ZK components. When a container type object is associated with a data model element, binds on child components that do not specify a data source are understood as parent container relative XPaths.

The components that can act as context data model implement the es.caib.zkib.BindContext interface. Currently, grid, lisbox, row and form components. Due to implementation constraints, the listiem object does not implement the aforementioned interface.

The simplest of those components is the Form object (which derives from Vbox). For context associations, the attribute to be used is called datapath, as shown in the following example:

```
<?xml version = "1.0" encoding = "UTF-8">
<zk>
   <xmlDataSource id = "mydata" src = "/ my-data.xml" />
   <form dataPath = "/mydata:/title">
      <label bind = "/@name" />
      <textbox bind = "/@name" />
      <button label = "Update" />
    </form>
</zk>
```

In this example, both the object label, as the textbox object refer to the same attribute @name, which is relative to the context /mydate:/title. When the user changes the textboxitem, the same text will be shown on the label. Mind the synchronization will only be done when the data is sent to the server, by means of any ZK event. In order to make user interface to be more responsive, simply add an onChange or onChanging null handler to the textbox component.

```
<?xml version = "1.0" encoding = "UTF-8">
<zk>
   <xmlDataSource id = "mydata" src = "/ my-data.xml" />
   <form dataPath = "/mydata:/title">
      <label bind = "/@name" />
      <textbox bind = "/@name" onChanging=""/>
      <button label = "Update" />
    </form>
</zk>
```

Binding Data Model Collections

Components of type listbox and grid behave as complex containers of object collections. These components can be assigned a Xpath expression that does not identify a single object, but a collection of objects or values. In this case, the system will generate a row for each of the values retrieved by the XPath query.

In order to define how to render listbox elements (listiem) bound to each data model row, the dataitem component should be used. The dataitem component is used as a template for the generation of any listitems. It should be noted that the each generated listitem is bound to a single object from the data model collection. This object is used also as a data context for the listcell's bound xPath. More and more, the attribute bind on the dataitem is the XPath that point to the value of the listitem.

In a similar way, in order to define how a grid row should be rendered, the datarow component should be used. It is used as a template for the actual rows generated. A row will be generated for each single element obtained from the data model collection. This object is bound to it's corresponding row and serves as the data context for the row children components.

So, the following example illustrates how to show the countries on a list component:

```
<?xml version="1.0" encoding="UTF-8">
<?page title="ZK Data binding Demo">
<zk>
 <xmldatasource id="mydata" src ="/my-data.xml" />
 <vbox>
  <form dataPath = "/mydata:/title">
   <label bind="/@name" />
   <textbox bind = "/@name" />
   <button label = "Update" />
  </form>
  <listbox dataPath = "/country:/mydata">
   <listhead>
    <listheader label="Abbr" />
    <listheader label="Name" />
   </listhead>
```

```
    <dataitem bind = "/@abbreviation">
      <listcell bind = "/@abbreviation" />
      <listcell bind = "/@name" />
    </dataitem>
  </listbox>
 </vbox>
</zk>
```

Note that the dataitem component has a dual behavior regarding the data model. On one side, it serves as a context to the inner object, and by the other side, it is assigned a value of the data model, according to the specification of the listitem component.

Components as a data source

The listbox component has a dual role, as a data consumer and as a data source. We've seen how listbox component can act as a data consumer in the previous pages. No we'll see how is it acting as a data source.

Any zul component can use the listbox path as the left handside member of a bind expression. In such a case, the result would be the same than binding such components to the xpath relative to the listitem currently selected at the list box. When no listbox element is selected, the resulting XPath will be void, and thus the component will be disabled. When the selected listitem changes, automatically the listbox bound components will c

As an example, the following code will show the name of the selected country, allowing the user to modify it.

```
<?xml version="1.0" encoding="UTF-8"?>
<?page title="ZKIB Demo" ?>
<zk>
   <xmldatasource id="mydata" src="/my-data.xml" />
      <vbox>
      <form dataPath="/mydata:/title">
         <label bind="/@name"/>
         <textbox bind="/@name"/>
      </form>
      <listbox id="countries" dataPath="/mydata:/country">
         <listhead>
            <listheader label="Abbr"/>
            <listheader label="Name"/>
         </listhead>
         <dataitem bind="/@abbreviation">
            <listcell bind="/@abbreviation" />
            <listcell bind="/@name"/>
         </dataitem>
      </listbox>
      <hbox>
         <label value="Active country:"/>
```

```
        <textbox bind="/countries:/@name"/>
      </hbox>
      <button label="Update"/>
    </vbox>
</zk>
```

Additionally, you can display a grid with the names of the cities of the selected country. Using the previous listbox as the data source, the component synchronization is automatic:

```
<?xml version="1.0" encoding="UTF-8"?>
<?page title="ZKIB Demo" ?>
<zk>
   <xmldatasource id="mydata" src="/my-data.xml" />
      <vbox>
      <form dataPath="/mydata:/title">
         <label bind="@name"/>
         <textbox bind="@name"/>
      </form>
      <listbox id="countries" dataPath="/mydata:/country">
         <listhead>
            <listheader label="Abbr" />
            <listheader label="Name"/>
         </listhead>
         <dataitem bind="@abbreviation">
            <listcell bind="@abbreviation" />
            <listcell bind="@name" />
         </dataitem>
      </listbox>
      <hbox>
         <label value="Active country:"/>
         <textbox bind="/countries:/@name"/>
      </hbox>
      <grid dataPath="/countries:/city">
         <columns>
            <column label="City"/>
         </columns>
         <datarow>
            <label bind="@name" />
         </datarow>
      </grid>
```

```
        <button label="Update"/>
    </vbox>
</zk>
```

Data model manipulation

The data model can be manipulated directly using the JXPathContext interface or indirectly through components, Whenever the user changes the contents of a ZK component, which is bound to a data model object, the change is propagated to the model, which in turn, propagates it to any ZK component that is bound to the modified data model object.

To ease data manipulation, the listbox object, has two new methods: addNew and delete. The first one creates a new object in the data model, and therefore in the listbox, while the second one removes it from the list and the model. Additionally, the autocommit attribute determines whether listbox will try to perform a commit the data model each time the user changes the selected item.

Alternatively, the data model can be directly manipulated through a Apache's commons-JXPath derived package. Every data source component (including datamodel and lisbox) contains a JXPathContext. Through this context JXPath invocations can be made, usign getValue, setValue, removePath or createPath methods. For more information about JXPath use, please review the available docs at: http://jakarta.apache.org/commons/jxpath/

Mind that the components will be notified of any change made through JXPath API, but the won't be noticed of any change made directly to the underlying data model. On the undesirable case underlying data objects are modified directly, you can force the components to be notified by getting a Pointer to the modified object and calling the invalidate method on it.

As an example, the following example shows how to delete a city or ad a new one to the data model:

```
<?xml version="1.0" encoding="UTF-8"?>
<?page title="ZKIB Demo" ?>
<zk>
   <xmldatasource id="mydata" src="/my-data.xml" />
      <vbox>
      <form dataPath="/mydata:/title">
         <label bind="@name"/>
         <textbox bind="@name"/>
      </form>
      <listbox id="countries" dataPath="/mydata:/country">
         <listhead>
            <listheader label="Abbr" />
            <listheader label="Name"/>
         </listhead>
```

```
        <dataitem bind="@abbreviation">
            <listcell bind="@abbreviation" />
            <listcell bind="@name" />
        </dataitem>
    </listbox>
    <hbox>
        <label value="Active country:"/>
        <textbox bind="/countries:/@name"/>
    </hbox>
    <grid dataPath="/countries:/city">
        <columns>
            <column label="City"/>
            <column/>
        </columns>
        <datarow>
            <label bind="@name" />
            <image src="/img/remove.gif">
                <attribute name="onClick">
                    row = self.parent;
                    ctx = row.dataSource.jXPathContext;
                    ctx.removePath(row.xPath);
                    ctx.getPointer("/").invalidate();
                </attribute>
            </image>
        </datarow>
    </grid>
    <button label="Add City">
        <attribute name="onClick">
            ctx = countries.jXPathContext;
            pointer = ctx.createPath("/city[count(/city)+1]");
            ctx2 = ctx.getRelativeContext(pointer);
            ctx2.createPath("/@name").setValue("London");
            ctx.getPointer("/").invalidate();
        </attribute>
    </button>
  </vbox>
</zk>
```

Building dynamic data model

While building the data model using XML files is possible, it's advisable to use more dynamic data models in the production environment. Alternatively, ZKIB module provides a set of classes that ease interaction with data models based on JDBC databases and object-oriented layers like Hibernate or EJB. Additionally.

To make use of these dynamic data models, the developer must implement one or more instances of the DataNode object. These DataNode objects act as wrapping around the actual business bean, managing the persistence layer. In general, you should define a class for each XML file equivalent entity. Thus, in the previous example, the developer should implement classes for the root (my-data), title, country and city objects. In some cases you won't need to develop a new DataNode class as long as the data object needs not to be persisted. For those simple data objects, a SimpleDataNode object can be used.

Each class is responsible for encapsulating a business object, and for its serialization to the persistent repository, usually a database. Also, each class is responsible for retrieving child objects. The children population function is performed by the finder interface.

A DataNode object must implement the following methods:

Method	Attributes	Description
<constructor>	DataContext	Optionally, you must declare the "finders", by calling the method addFinder
doInsert		Insert a new object in the persistent storage
doUpdate		Update an object in the storage persitente
doDelete		Update an object in the storage persitente

It's noticeable, that doInsert, doDelete and doUpdate methods are always invoked in a transactional context, coordinated through the commit method of the DataSource. The DataNode derived objects can access the data of the underlying business object by calling the inherited method getInstance.

The Finder interface must implement the following methods:

Method	Result	Description

find	java.util.Collection	Retrieves a collection of business objects that are descendants of the current object. Note that this method should not construct DataNode objects that will wrap the business object later.
newInstance	Object	Creates a new business object, filling the default value of its attributes, depending on the context in which they are creating.

Thus, to implement the same countries data model, equivalent to the former XML file you should define the following business classes:

Country business class

```
package com.soffid.sample;

public class Country {
    private String name;
    private String abbreviation;
    public String getName () {return name;}
    public void setName (String name) {this.name = name;}
    getAbbreviation public String () {return abbreviation;}
    public void setAbbreviation (String abbreviation) {
        this.abbreviation = abbreviation;
    }
}
```

City business class

```
package com.soffid.sample;

public class City {
    private String name;
    public String getName () {return name;}
    public void setName (String name) {this.name = name;}
}
```

Title business class

```
package com.soffid.sample;
```

```
public class Title {
    private String name;
    public String getName () {return name;}
    public void setName (String name) {this.name = name;}
}
```

Now that we have the needed business classes, the next step is to create the a class that manages the root of the data tree. This class derives from SimpleDataNode because they do not allow the execution of the methods doInsert, doUpdate or doDelete. In its constructor defines two Finders, responsible for retrieving the list of countries (through a existing countryDAO class) and title (with code):

```
package com.soffid.sample;
import java.util.Vector;
import es.caib.zkib.datamodel.*;
import es.caib.zkib.datasource.*;
public class RootNode extends SimpleDataNode {
    public RootNode(DataContext ctx) {
        super(ctx);
        // Title
        addFinder("title",
                new Finder () {
                    public java.util.Collection find() throws Exception {
                        Title t = new Title ();
                        t.setName ( "World countries" );

                        Vector v = new Vector();
                        v.add (t);
                        return v;
                    };
                    public Object newInstance() throws Exception {
                        throw new UnsupportedOperationException();
                    }
                },
                SimpleDataNode.class);
        // Countries
        addFinder("country",
            new Finder () {
                public java.util.Collection find() throws Exception {
                    return CountryDAO.findAll ();
                };
```

```
            public Object newInstance() throws Exception {
                throw new UnsupportedOperationException();
            }
        },
        CountryNode.class);
    }
}
```

The CountryNode object class referenced in the previous class will wrap for Country objects got by DAO. This class is responsible to retrieve and instantiate City objects from the Country. It is derived from SimpleDataNode because no update, insert or delete of countries is allowed:

```
package com.soffid.sample;
import java.util.Vector;
import es.caib.zkib.datamodel.*;
import es.caib.zkib.datasource.*;

public class CountryNode extends DataNode {
    public CountryNode(DataContext ctx) {
        super(ctx);
        addFinder("city",
                new Finder () {
                    public java.util.Collection find() throws Exception {

                        Country c = (Country) getInstance();
                        return CityDAO.findByCountry (c.abbreviation);
                    };
                    public Object newInstance() throws Exception {
                        Country country = (Country) getInstance();

                        City city = new City();
                        city.setCountryAbbreviation (country.getAbbreviation());
                        return city;
                    }
                },
                CityNode.class);
    }
}
```

Finally, the CityNode class is responsible for managing City persistent objects. In this case there is no finder instance because the City has no children available:

```
package com.soffid.sample;
import java.util.Vector;
import es.caib.zkib.datamodel.*;
import es.caib.zkib.datasource.*;
public class CountryNode extends DataNode {
    public CountryNode(DataContext ctx) {
        super(ctx);
    }
    protected void doInsert() throws Exception {
        CityDAO.insert ((City) getInstance());
    }
    protected void doUpdate() throws Exception {
        CityDAO.update ((City) getInstance());
    }
    protected void doDelete() throws Exception {
        CityDAO.delete ((City) getInstance());
    }
}
```

In summary, we have had to generate four kinds of objects corresponding to the three types of element of the XML document:

- my-data: RootNode class derived from SimpleDataNode . It contains two finders, one for title, and one for country.
- title: is using SimpleDataNode class.
- country: CountryNode class derived from SimpleDataNode. It contains a finder that allows the instantiation of City objects.
- city: CityNode class derived from DataNode. Implements methods to persist City object. It has no finder.