ZK Essentials

For ZK 6

Contents

Articles

References

ZK Essentials

Documentation:Books/ZK_Essentials

If you have any feedback regarding this book, please leave it here.
<comment>http://books.zkoss.org/wiki/ZK_Essentials</comment>

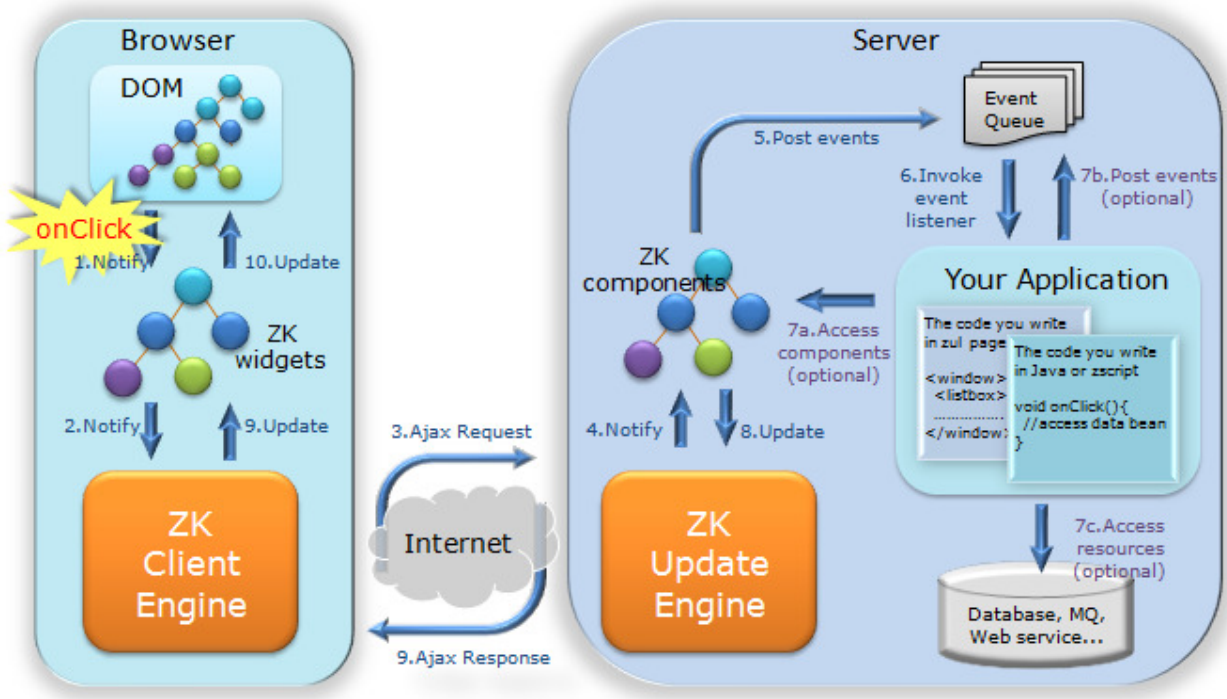
Introduction

The aim of ZK Essentials is to provide a step by step resource for a developer to address the creation of a substantial application using ZK. Each chapter navigates through a particular ZK topic and explains in detail why certain choices and implementation techniques were chosen over others in context of the application.

The book chooses a shopping cart application as a platform to explore ZK. By the end of this book you should be able to gather enough information about ZK techniques and practices to implement a custom Web application.

The book starts off by introducing ZK and its architecture then moves into the creation of the application.

An Introduction to ZK's Server client Fusion Architecture



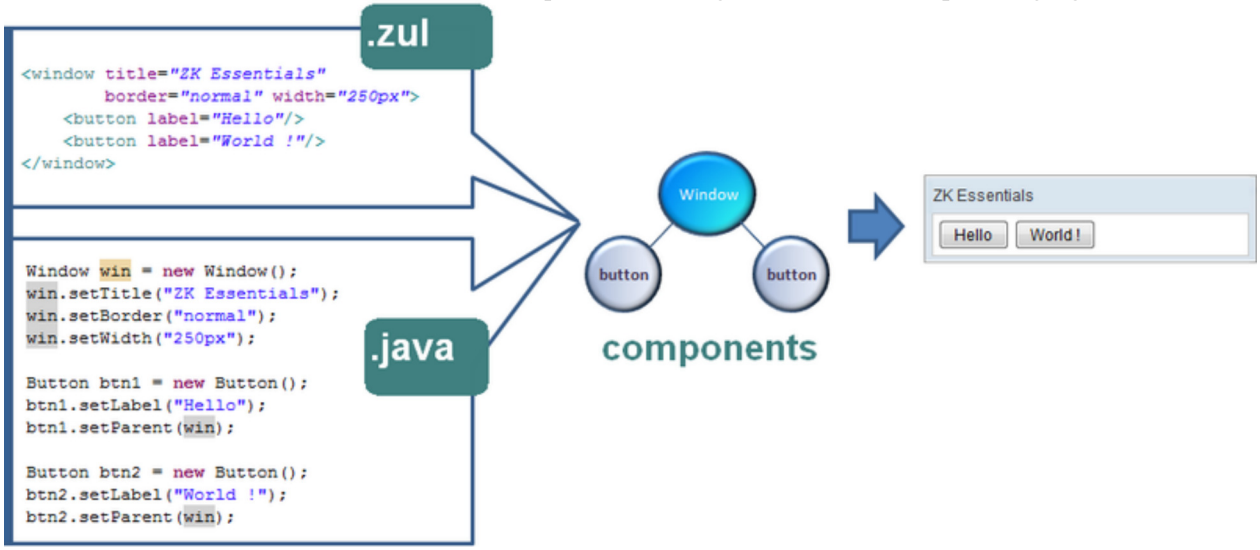
When a ZK application runs on the server, it can have access to the backend resources, assemble UI with components, listen to user's activity, and then manipulate components to update UI. All are done at the server. The synchronization of the states of the components between the browser and the server is done automatically by ZK and transparently to the application.

When running at the server, the application can access full Java technology stack. User activities, including Ajax and Server Push, are abstracted to event objects. UI are composed of POJO-like components. ZK is the most productive approach to develop a modern Web application.

With ZK's Server+client Fusion architecture, your application will never stop running on the server. The application could enhance the interactivity by adding optional client-side functionality, such as client-side event handling, visual effect customizing or event even UI composing without server-side coding. ZK enables seamless fusion from pure server-centric to pure client-centric. You can have the best of two worlds: productivity and flexibility.

Component Based UI

In ZK, we can declare UI components using either markup language or Java.



For example, here we declared a Window [1] component, setting the border to normal and resizing its width to a definite 250 pixels. Enclosed in the Window [1] are two Button [2] components.

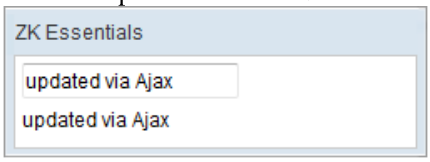
Where We Declare the Components

The components are declared in files with the extension ".zul". A ZUL page is interpreted dynamically on the server. We could think of it as a JSP empowered with Ajax capabilities. For instance, here we create a new ZUL file, **ajax.zul**, and we implement a sample, in which users' input in Textbox [3] is reflected in the label below instantly when the text box loses focus:

ajax.zul

```
<window title="ZK Essentials" border="normal" width="250px">
	<vlayout>
	<textbox id="txtbx" onChange="lbl.value = txtbx.value"/>
	<label id="lbl"/>
	</vlayout>
</window>
```

The markup declaration above renders a sample program, see below:



ZK components can also be easily declared in Java source codes and are also compatible with Java. Please see the following.

```
package org.zkoss.zkdemo;

import org.zkoss.zk.ui.Page;
import org.zkoss.zk.ui.GenericRichlet;
import org.zkoss.zul.*;

public class TestRichlet extends GenericRichlet {
    //Richlet//
    public void service(Page page) {

        final Window win = new Window("ZK Essentials", "normal",
false);
        win.setWidth("250px");

        Vlayout vl = new Vlayout();
        vl.setParent(win);

        final Textbox txtbx = new Textbox();
          txtbx.setParent(vl);

        final Label lbl = new Label();
          lbl.setParent(vl);

        txtbx.addEventListener("onChange", new EventListener(){
                @Override
                public void onEvent(Event event) throws Exception {
                      lbl.setValue(txtbx.getValue());
                }
           });


        win.setPage(page);
    }
}
```

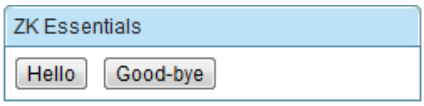
Please refer to Developer's Reference: Richlet for more details on programming with Richlets.

What the Components Declarations Become

Components declared using ZUML in a ZUL file are parsed by a ZK enhanced XML parser. The components declared are created as Plain Old Java Objects (POJO) in the JVM on the server. Suppose we have a ZUL page that contains nested components as shown below:

```
<window title="ZK Essentials" border="normal" width="250px">
      <button label="Hello"/>
      <button label="Good-bye "/>
</window>
```

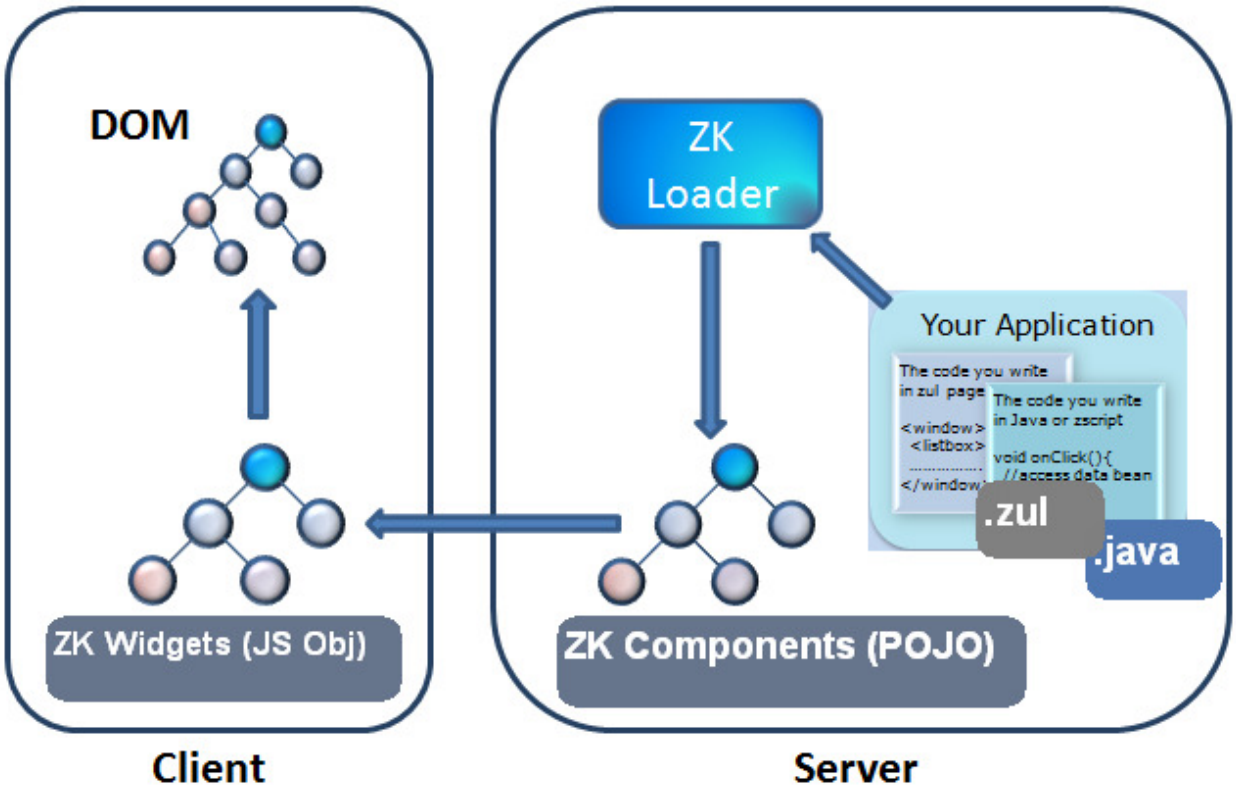
which would render a window containing two buttons as shown in the image below:



The markup in ZUL is equivalent to the following POJO declarations in Java:

```
Window win = new Window();
      win.setTitle("ZK Essentials");
      win.setBorder("normal");
      win.setWidth("250px");

Button helloBtn = new Button();
      helloBtn.setLabel("Hello");
      helloBtn.setParent(win);

Button byeBtn = new Button();
      byeBtn.setLabel("Good-bye");
      byeBtn.setParent(win);
```

Components on the server are translated into instructions(in JSON) for widget (JavaScript objects) creation and then sent to the client.



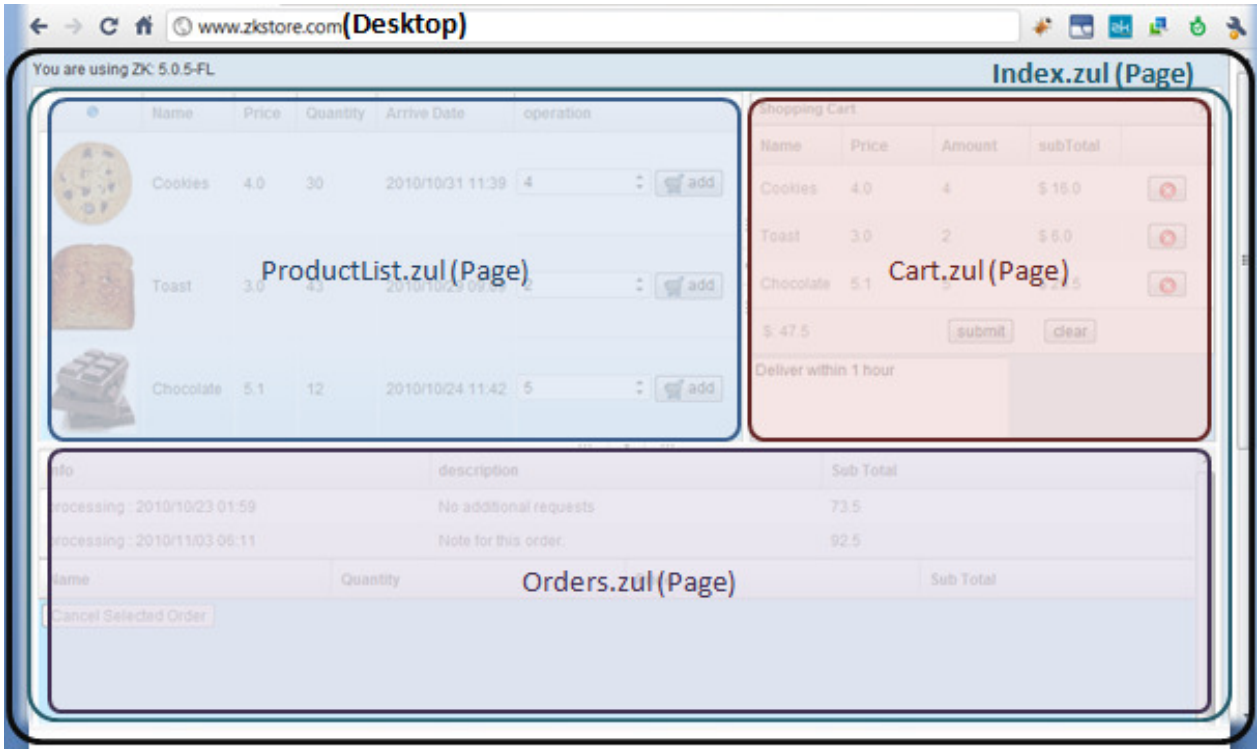
Where the Components Belong

The Page

Imagine components are actors in a play, then Page [4] is the stage where components play out their roles. A page is a space holder in a browser window where ZK components can be attached and detached. A Page [4] is not a component and it does not implement a Component [5] interface. A page is automatically created for a ZUL page when user makes the request for it.

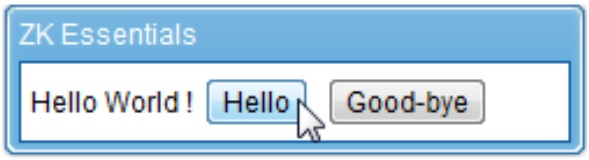
The Desktop

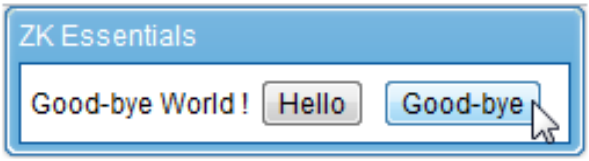
Suppose we have a shopping cart application deployed on **www.zkstore.com**. When a user enters this URL in a browser, by default, the page **index.zul** is requested. A Desktop [6] is created automatically when a Page [4] is created. A desktop may contain one or more pages, serving requests for the same URL. The concept is illustrated below:



How to Access a Component

With the components nested and stacked up together to give us our application UI, we need a way to identify the necessary ones for manipulation. For example, we might need to dynamically append a component to an existing component, or program one component's behaviour to depend on that of another.
The sample below illustrates the case:





The markup source is:

```
<window title="ZK Essentials" mode="overlapped" border="normal" width="250px">
      <label id="lbl"/>World !
      <button label="Hello " onClick="lbl.value = self.label"/>
```

```
      <button label="Good-bye " onClick="lbl.value = self.label"/>
</window>
```

The value of the label with ID "lbl" depends on which button the user clicks: "Hello", or "Good-bye". When a button is clicked, the value of the button's label is assigned to the value of the label. Note that the string "World !" is automatically converted to a label. ZK assigns each component with a UUID (Universal Unique Identifier) to keep track of all the components internally. This UUID is over-ridden when the developer provides it with a more legible ID.

Finding a Component Programmatically in Java

Suppose we have a POJO declaration for a simple UI that looks something like this:

```
Window outerWin = new Window();

Button outerBtn = new Button();
btn.setParent(outerWin);

Window innerWin = new Window();
innerWin.setParent(outerWin);

Button innerBtn = new Button();
innerBtn.setParent(innerWin);
```

For better readability, the equivalent declaration above using ZUML in ZUL file is given here:

```
<window id="outerWin">
       <button id="outerBtn">
        <window id="innerWin">
             <button id="innerBtn"/>
        </window>
</window>
```

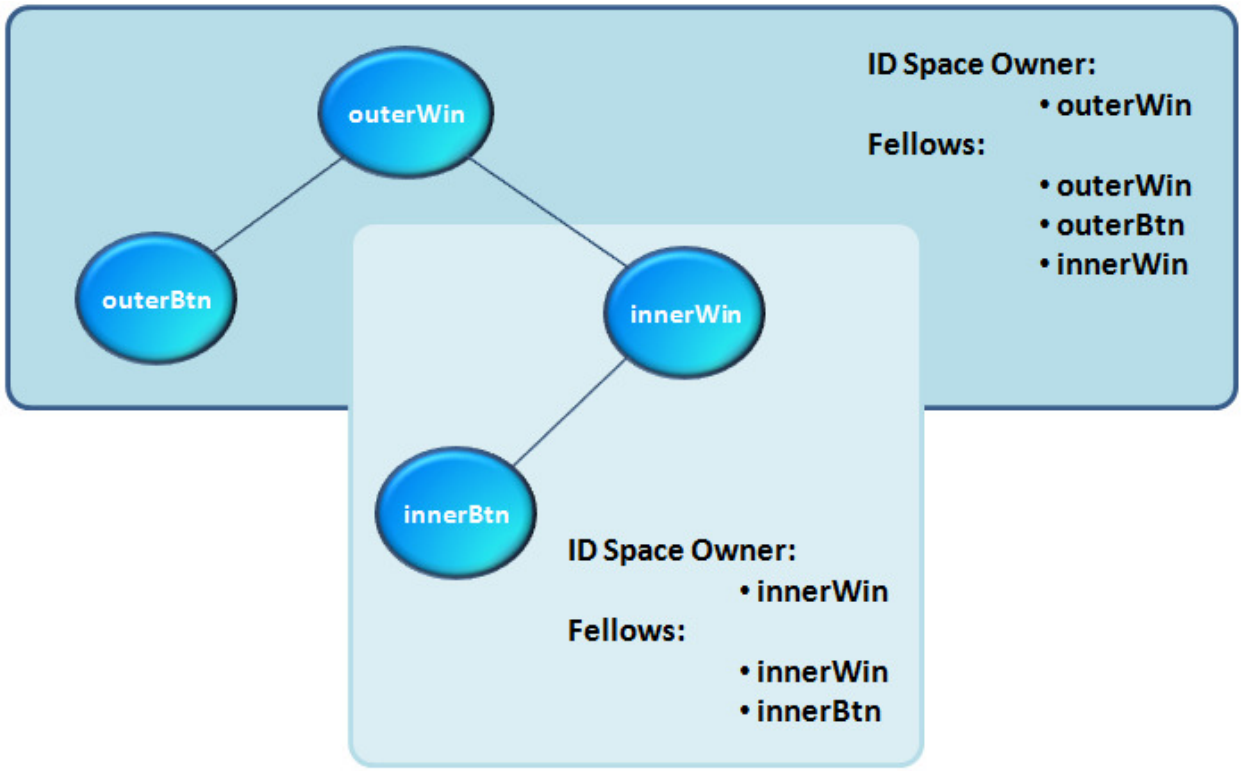
Now suppose we have a controller class where we want to programmatically access and modify the children components (outerBtn, innerWin, innerBtn); how should we accomplish this if we only have a reference to the Window component?

One of the approaches is to call the component's Component.getFellow() [7] method. For example, if we wish to access the inner Window, we could do the following:

```
Window innerWin = (Window)outerWin.getFellow("innerWin");
```

```
We can call the getFellow method on any component to access another component in the same ID Space.
```

An ID Space is a way to group components into a more manageable collection in ZK so we don't risk getting references to components in a large component tree mixed up. This concept is illustrated below:



The Window [1] component and Page [8] are **Space Owners** by default. We can make any component a space owner by allowing the component to implement the IdSpace [9] interface. To identify a component's space owner, we cam call its **getSpaceOwner** method.

There are other methods to accomplish this, we summarize them in a table for clarity:

Component	method	Note
outerBtn	= (Button)outerWin.getFellow("outerBtn");	The components **outerWin**, **outerBtn**, and **innerWin** form an **ID Space**; with **outerWin** being the **Space Owner**. Components in the same ID Space can call each other by ID using the **getFellow** method.
innerWin	= (Window)outerWin.getFellow("innerWin");	**innerWin** belongs to the same ID Space as **outerWin**, hence, it could be called using the **getFellow** method.
innerBtn	= (Button)outerWin.getFellow("innerWin").getFellow("innerBtn");	**innerWin** and **innerBtn** belong to an **ID Space** of their own, with **innerWin** being the **Space Owner**. **innerWin** is also a member of the **ID Space** which **outerWin** is the **Space Owner**. Hence, we can call **getFellow** on **outerWin** to get **innerWin**, then call **getFellow** on **innerWin** to get **innerBtn**.
outerBtn	= (Button)Path.getComponent("/outerBtn");	The Path [10] provides the utility method **getComponent** which takes the relative **path** of the component as its argument. **/outerBtn** is equivalent to **outerWin/outerBtn**
innerWin	= (Window)Path.getComponent("/innerWin");	**innerWin** and **outerBtn** both have **outerWin** as an ID Space Owner.
innerBtn	= (Button)Path.getComponent("/innerWin/innerBtn");	**innerBtn** has **innerWin** as its ID Space Owner, **innerWin** in turn has **outerWin** as its **Space Owner**. Hence, we write **/innerWin/innerBtn**, which is equivalent to **outerWin/innerWin/innerBtn**

outerBtn	= (Button)outerWin.getFirstChild();	The **getFirstChild** method returns the first child component of the caller. The advantage of using this method is that you don't even need to know the component ID to fetch the component.
innerWin	= (Window)outerWin.getFirstChild().getNextSibling();	The **getFirstChild** method gets the **outerBtn** since it's outerWin's first child component. We then call the **getNextSibling** method to find the **innerWin**.
innerBtn	= (Button)outerWin.getFirstChild().getNextSibling().getFirstChild();	We compound another **getFirstChild** method to get the first, and only, child component of **innerWin**.

Notes on XML Syntax

The language we use to declare the components is ZUML, an abbreviation for ZK UI Markup Language. ZUML follows the syntax of XML. Here are a couple of basic quick notes if you're not familiar with XML[11] .

- **Elements must be well formed**
  - *close declaration with an end tag:*

```
<window></window>
```

  - *close declaration without an end tag (equivalent to the above statement):*

```
<window/>
```

- **Elements must be properly nested:**
  - *Correct:*

```
<window>
  <groupbox>
        Hello World!
  </groupbox>
</window>
```

  - *Wrong:*

```
<window>
  <groupbox>
        Hello World!
  </window>
 </groupbox>
```

- **Only a single "root" component is allowed:**
  - *one root - legal*

```
<button />
```

  - *two roots - illegal*

```
<button/>
<button/>
```

  - *one root containing all other components - legal*

```
<window>
  <button/>
```

```
        <button/>
    </window>
```

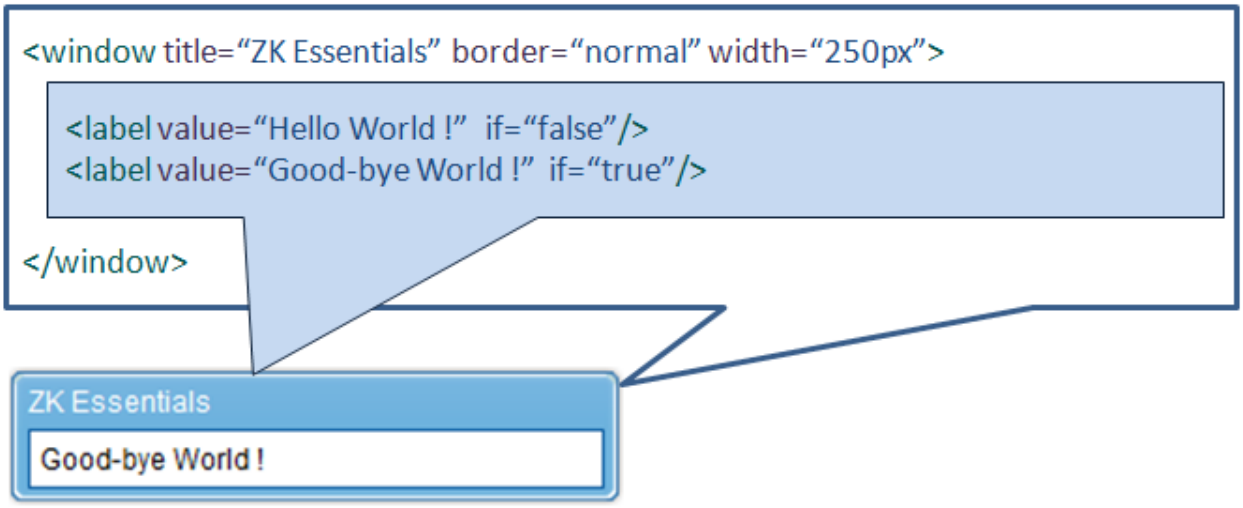
- **Attribute value must be quoted**
  - *Correct:*

```
    <window width="600px"/>
```

  - *Incorrect:*

```
    <window width=600px/>
```

Using XML tags, we declare a component and set a component's attributes; as an alternative to coding in Java files, we could set a component's attributes to initialize values, evaluate conditions/expressions, and handle events. The figure belows shows an example of how we could easily dictate whether a component is to be displayed on a page when the "if" condition is declared as its attribute. The label "Hellow World !" is not displayed since its "if" condition is declared to be false.



[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Window.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Button.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Textbox.html#
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/ui/Page.html#
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#
[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/ui/Desktop.html#
[7] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#getFellow()
[8] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Page.html#
[9] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/IdSpace.html#
[10] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Path.html#
[11] Please refer to resources on Internet, such as (http://www.w3schools.com/xml/xml_whatis.asp) and (http://www.xml.com/pub/a/98/10/guide0.html) should you need to get comfortable with its syntax and conventions

Event Driven Programming

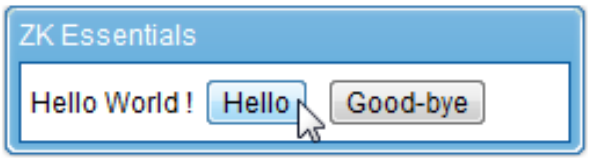
In the previous section, we saw how components are declared in ZK User Interface Markup Language (ZUML) to compose an application's user interface. Now we'll look into ZK's event driven programming model to make all the components in an application work together to meet our cause.

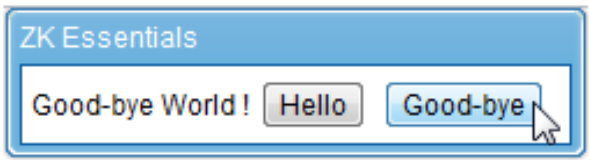
How Event Driven Programming Works with Request/Response

Event driven programming is widely adopted for responsive GUI experience. Each ZK component can accept one or more event listeners registered to itself. A component registered with an event listener will listen to actions from users or events registered by the ZK framework itself. The event driven programming model in ZK works perfectly with the Request-Response model of HTTP. Take the sample we saw in the previous section:

```
<window title="ZK Essentials" mode="overlapped" border="normal" width="250px">
      <label id="lbl"/>World !
      <button label="Hello " onClick="lbl.value = self.label"/>
      <button label="Good-bye " onClick="lbl.value = self.label"/>
</window>
```

The functionality for this sample code displays the strings of text the user has selected and the text is displayed in a label.





Breaking down how the event driven model works

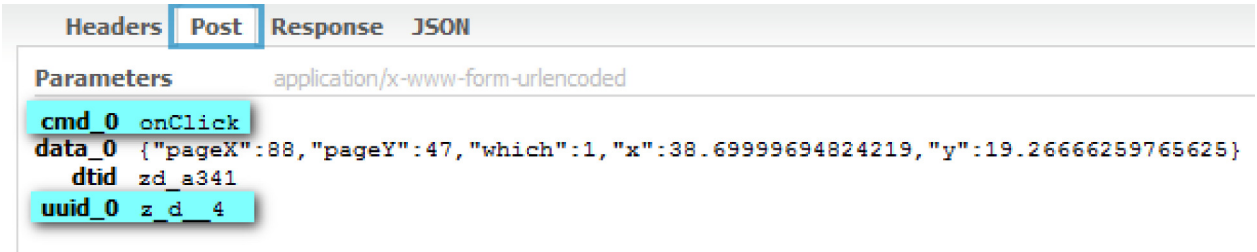
Register Event Listeners

- At line 3 and 4, we register the onClick events on the buttons so we'll know when and which button is clicked.

```
      <button label="Hello " onClick="lbl.value = self.label"/>
      <button label="Good-bye " onClick="lbl.value = self.label"/>
```

Event Sent with Request

- When the user clicks one of the buttons, its respective onClick event is sent to the server for processing by default (optionally, we could handle events from the client directly). As shown highlighted in blue, the "onClick" event and the button's UUID (Universal Unique Identifier) is sent by the Ajax request.



Event Handling Instructions

- From the request, the ZK upload engine knows **which component** fired the event and **what event** is fired. It also knows how to handle this request because we've specified the event handling code in ZUL using EL (Expression Language): "lbl.value = self.label", which translates into the following Java code:

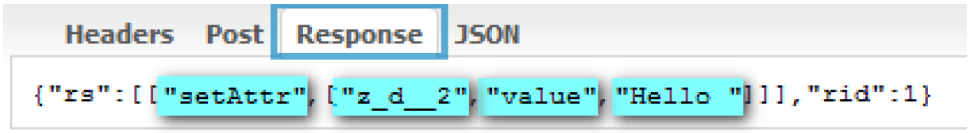
```
lbl.setValue(self.getLabel);
```

- Where lbl is the ID we assigned to the Label component and self refers to the Button component itself (akin to **this** in Java)

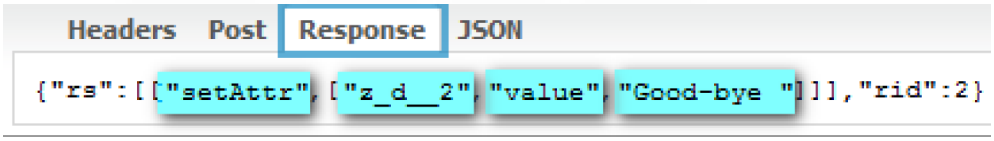
Update Instruction Sent Back in Response

- The server then sends the instructions back to the client to reflect this change in the label's value:

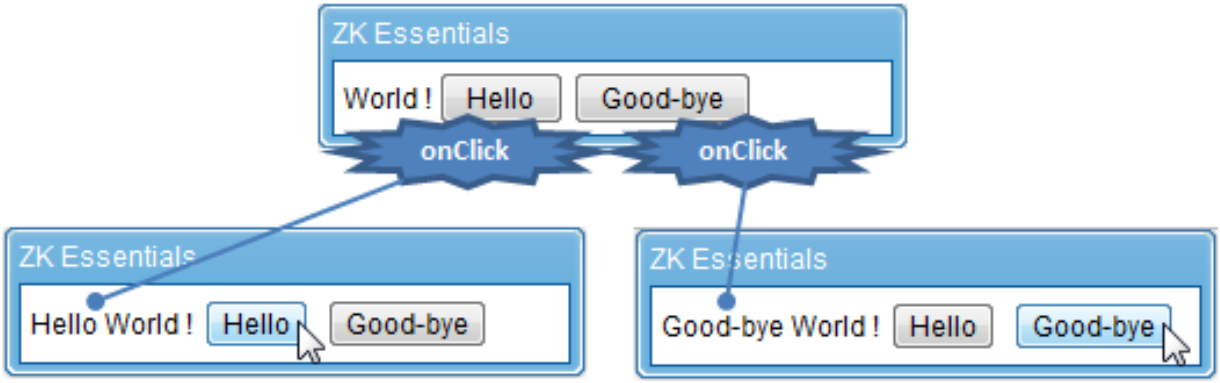
Hello



Good-bye



- Where the instructions tell the client engine to set the attribute "value" to "Hello" for the component with the UUID "z_d__2", which is the label component.



How to Register Event Listeners

Various ways are provided in ZK to register and handle events, we'll look into the most common usages here.

Event Handling in ZUL page

In a ZUL file, an event handler is registered to a component as the component's attribute.
For example, suppose we have a button labeled "Hello", and we want to create another button labeled "World !" dynamically when the "Hello" button is clicked.

We could program this behavior entirely in a ZUL file:

```
<window id="win" title="ZK Essentials" border="normal" width="250px">
      <button label="Hello">
            <attribute name="onClick">
            <![CDATA[
                  Button btn = new Button();
                  btn.setLabel("World !");
                  btn.setParent(win);
            ]]>
            </attribute>
```

```
        </button>
</window>
```

Inside the button declaration, we declare the "onClick" event as the button's attribute and code its event handling instructions directly in ZUL. This is possible because the Java code is interpreted by BeanShell [1] dynamically. In the Java code, we first created a new Button, set its label to "World !", and then set the Window component as its parent. Despite its convenience, registering event directly from ZUL is not recommended if your application is performance sensitive. As all the Java code needs to be interpreted dynamically.

Event Handling In a Controller

Following the ZK MVC pattern, we create a controller class where we can wire variables and listen to events in our UI.
So here we make the necessary changes:

- At line 1, we declared: apply=*controller class name* so that the events fired within the window is all forwarded to the controller for event handling.

```
<window id="win" title="ZK Essentials" border="normal" width="250px" apply="demo.zkoss.Sa
        <button label="Hello"/>
</window>
```

We then implement our controller class:

- At line 9, we extend the SelectorComposer [2] so we can wire variables and listen to events using annotations whose parameters are CSS-like selectors.
- At lines 11 and 12, we declare a Window component and wire it with the window in our ZUL file using annotation.
- At line 14, we use a selector to wire the Button whose label is "Hello" and listen to its onClick event.
- At line 15, the method createWorld contains the same Java code we did previously to dynamically create a button.

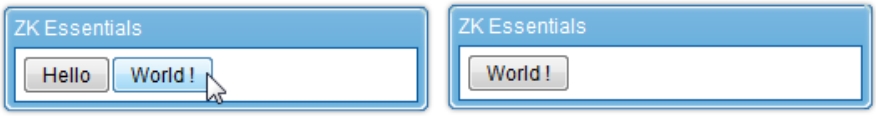
```
package demo.zkoss;

import org.zkoss.zk.ui.select.SelectorComposer;
import org.zkoss.zk.ui.select.annotation.Listen;
import org.zkoss.zk.ui.select.annotation.Wire;
import org.zkoss.zul.Button;
import org.zkoss.zul.Window;

public class SampleCtrl extends SelectorComposer {

        @Wire("window")
        Window win;

        @Listen("onClick = button[label='Hello']")
        public void createWorld(){
                Button btn = new Button();
                btn.setLabel("World !");
                btn.setParent(win);
        }
}
```

Event Handling of Dynamically Generated Components

Suppose we take our previous example a step further, we would like to clear off the "Hello" button in the window when we click the dynamically created "World !" button. We'll need to register an event listener to the dynamically created "World !" button.

- At line 12, we add a new event listener to the newly created button. The **addEventListener** method takes an event name (String) and an EventListener [3] as its arguments.
- Within the anonymous **EventListener** class, we implement the **onEvent** method to have the Window component fetch its first child component, which is the button labelled "Hello", and call its **detach** method to clear it off the Window component.

```
public class SampleCtrl extends SelectorComposer {

	@Wire("window")
	Window win;

	@Listen("onClick = button[label='Hello']")
	public void createWorld(){
		Button btn = new Button();
		btn.setLabel("World !");
		btn.setParent(win);

		btn.addEventListener("onClick", new EventListener(){
			public void onEvent(Event event) throws Exception {
				Button helloBtn;
				win.getFirstChild().detach();
			};
		});

	}
}
```



References

[1] http://www.beanshell.org/
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/ui/select/SelectorComposer.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/EventListener.html#

Working with the Sample Applications

In this section, we'll go over how to set up the sample applications on a ZK development environment.

The Resources

This book comes with two examples:

- **Store** - which uses an in memory model defined by Java statements
- **Store with database** - which uses a persistent model driven by Hibernate. This is used at the last chapter of the book.

If you require Maven & Git, please download and install them first:

1. Maven [1]
2. Git [2]

Store

The store is available on github, you can retrieve it using the following commands:

```
git clone git://github.com/zkbooks/ZK-Essentials.git zkessentials
git checkout withoutdb
```

If you prefer to set up the project in eclipse please refer to here.

Store with Database

The store with database is also available on our svn. You can checkout the project using:

```
git clone git://github.com/zkbooks/ZK-Essentials.git zkessentials
git checkout withdb
```

If you prefer to set up the project in eclipse please refer to here.

Running the Sample Applications Using Maven

To run the sample applications using Maven, change directory to zkessentials root and issue the command:

```
mvn jetty:run
```

and direct your browser to http://localhost:8080/zkessentials.

References

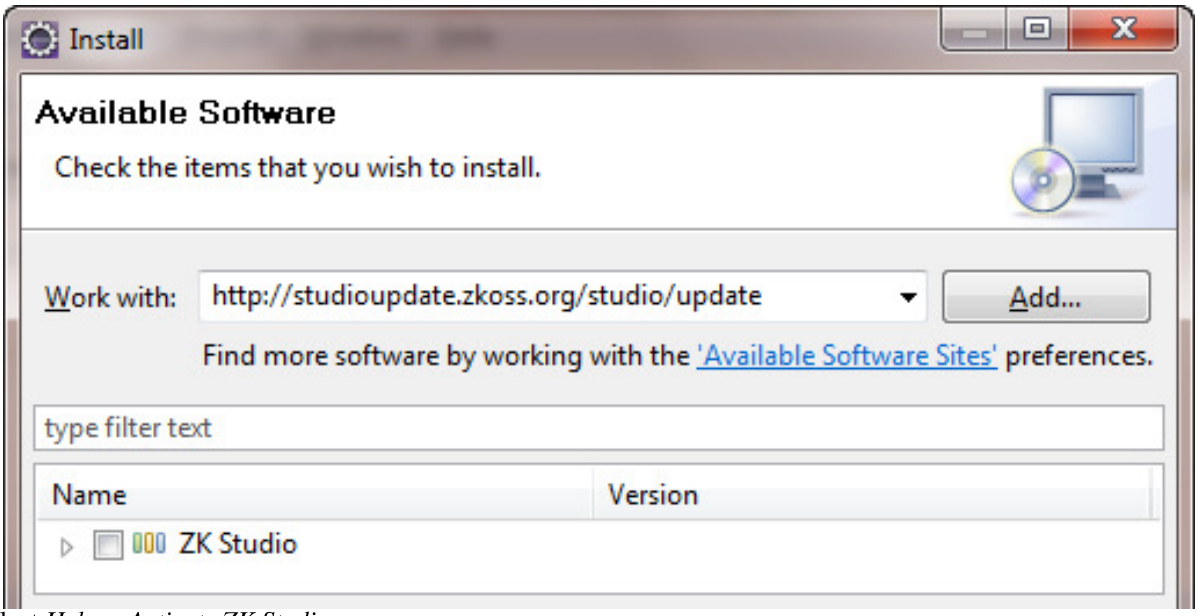
[1] http://maven.apache.org/download.html
[2] http://git-scm.com/download

Setting Up the Applications Using Eclipse

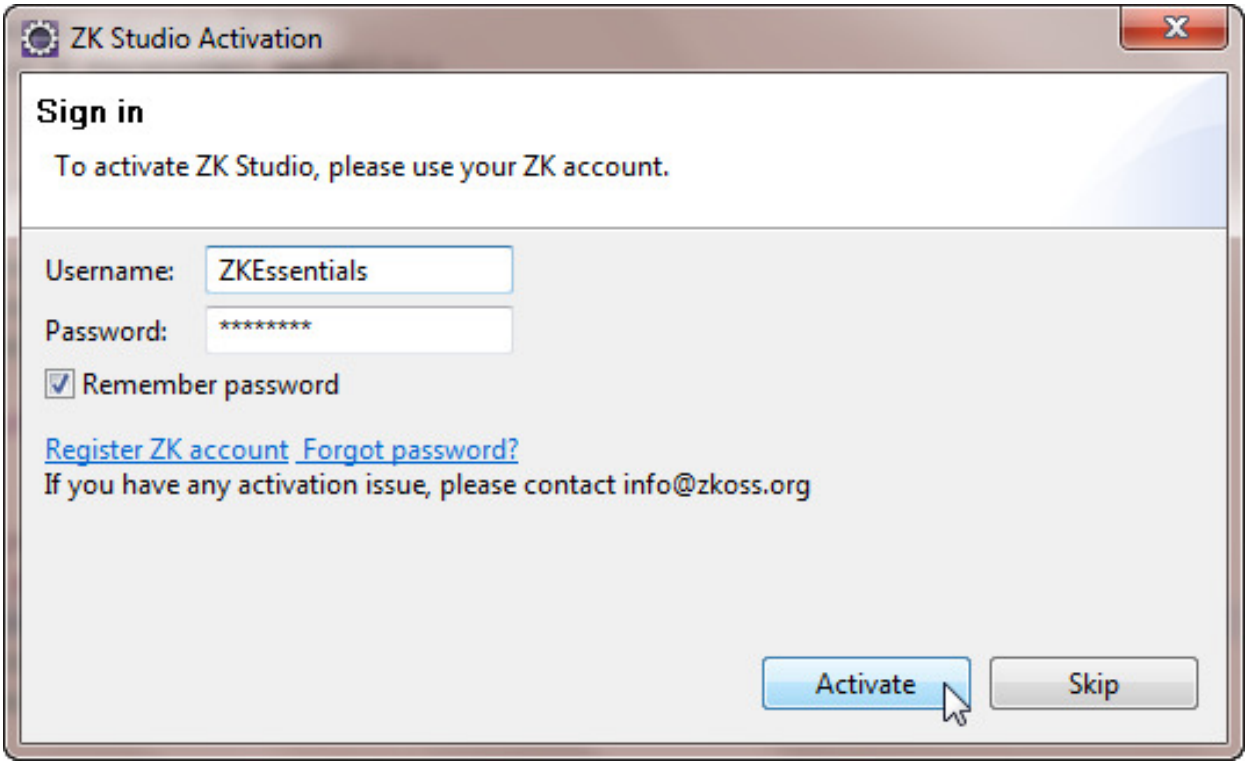
If you prefer to install it manually or use other IDEs, please refer to Create and Run Your First ZK Application Manually.

Set-up ZK Studio

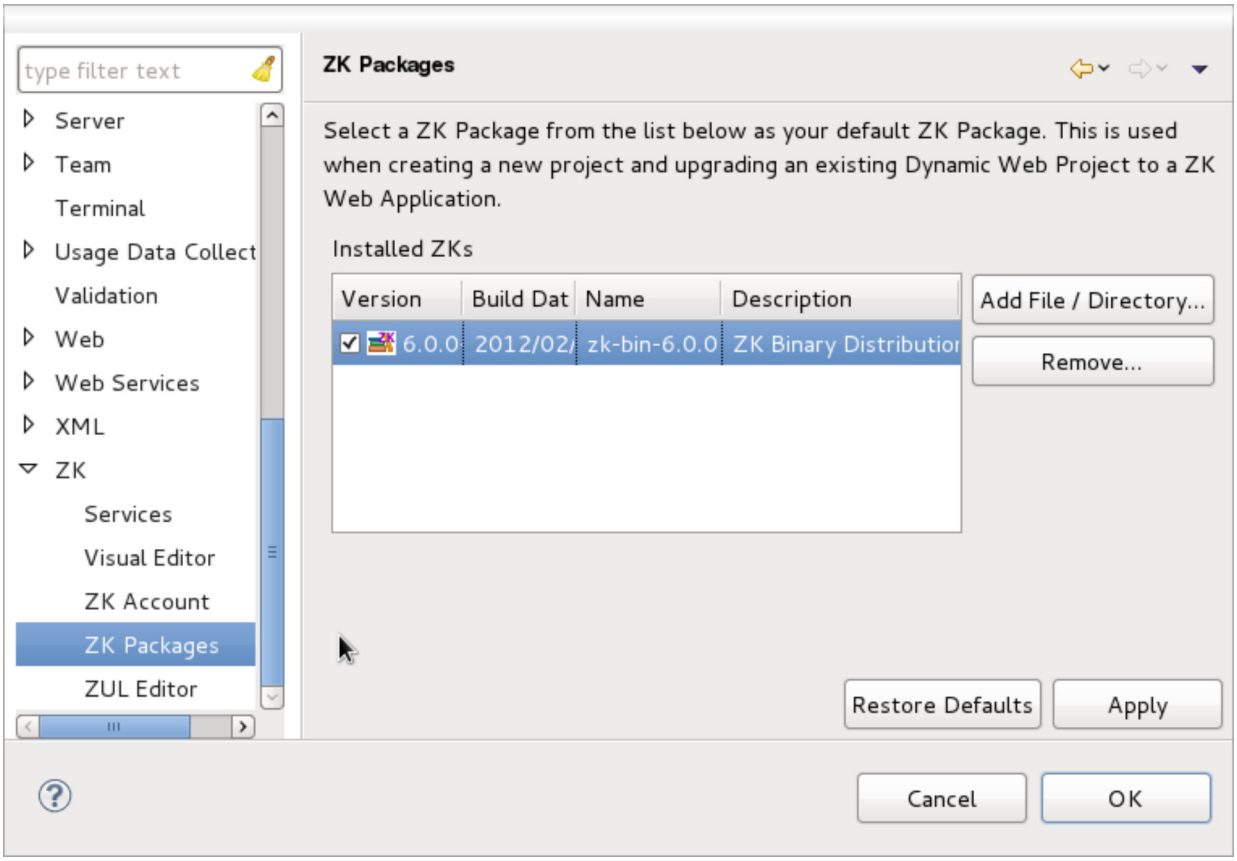
1. Open your Eclipse IDE
2. On the main menu bar, select *Help* > *Install New Software...*
3. Copy and paste the ZK Studio plugin's update URL : http://studioupdate.zkoss.org/studio/update/eclipse_3_5 For Eclipse 3.5, or http://studioupdate.zkoss.org/studio/update/eclipse_3_6 for Eclipse 3.6 into the input box as shown below:



4. Select *Help* > *Activate ZK Studio*



5. Once the download is complete, go to *Window* > *Preferences* > *ZK* > *ZK Packages*, click *Add File/Directory* to add the ZK package downloaded
6. Check-mark the package and click "Ok"



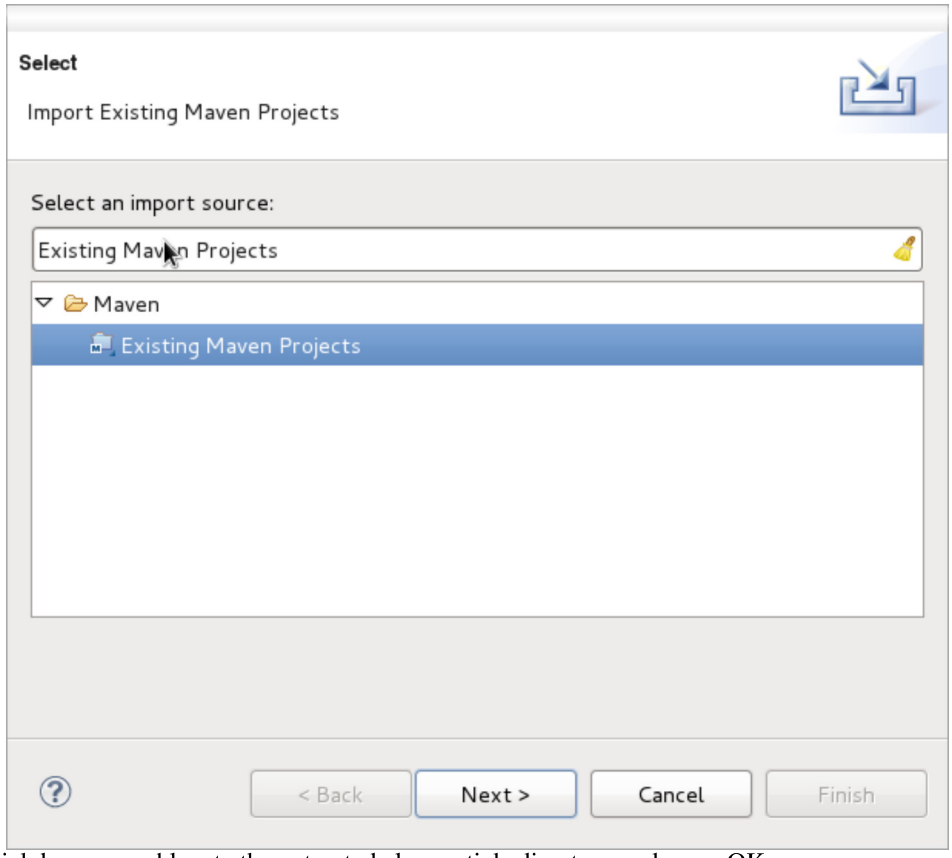
Store

Prerequisites

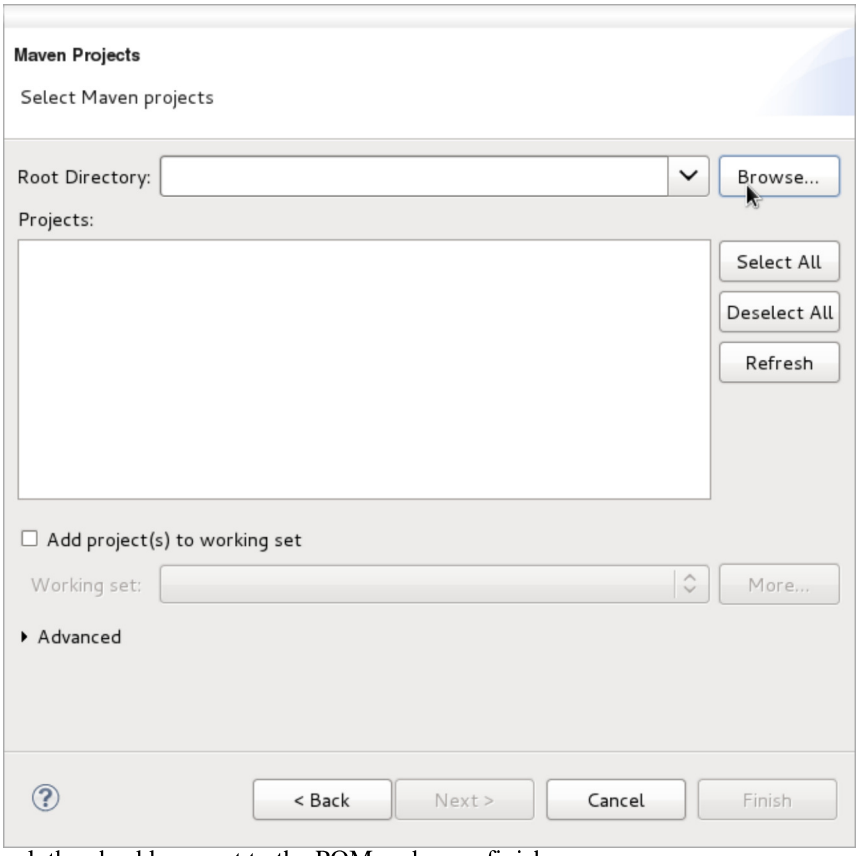
- You require Maven for ZK Studio/Eclipse, if you do now have it you can follow this tutorial on installing it [1]

Set-up the ZK Essentials sample

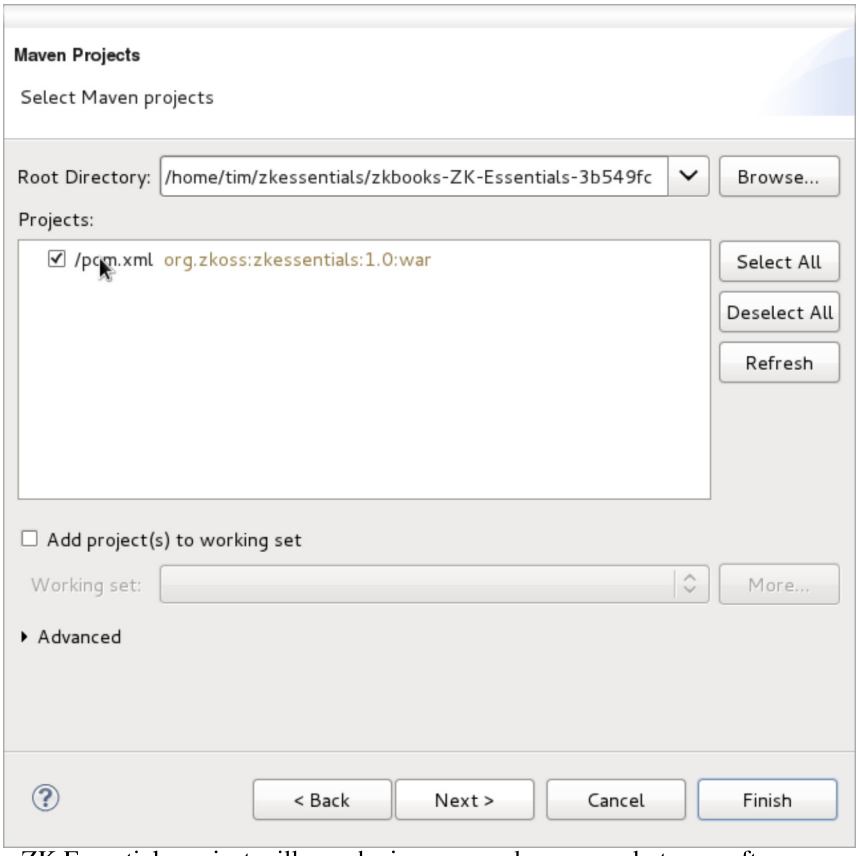
1. For this section we will assume you download the application first rather than do a git checkout from Maven
2. Therefore click here [2] to download the store without a database, and unzip to any folder you want
3. In ZK Studio go to New -> Import, and type "Existing Maven Projects", highlight the option and click next



4. Click browse and locate the extracted zkessentials directory and press OK



5. Check the checkbox next to the POM and press finish



6. The ZK Essentials project will now be in your workspace ready to run after maven downloads all the appropriate files

References

[1] http://m2eclipse.sonatype.org/installing-m2eclipse.html
[2] https://github.com/zkbooks/ZK-Essentials/zipball/withoutdb

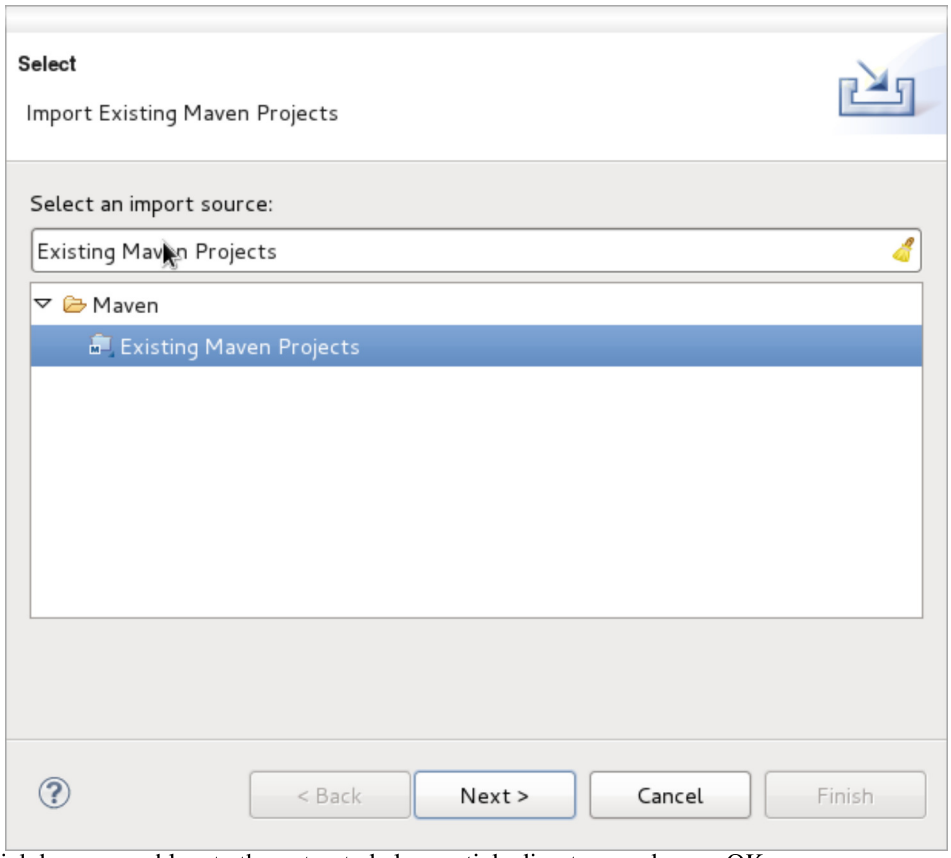
Store with a Database

Prerequisites

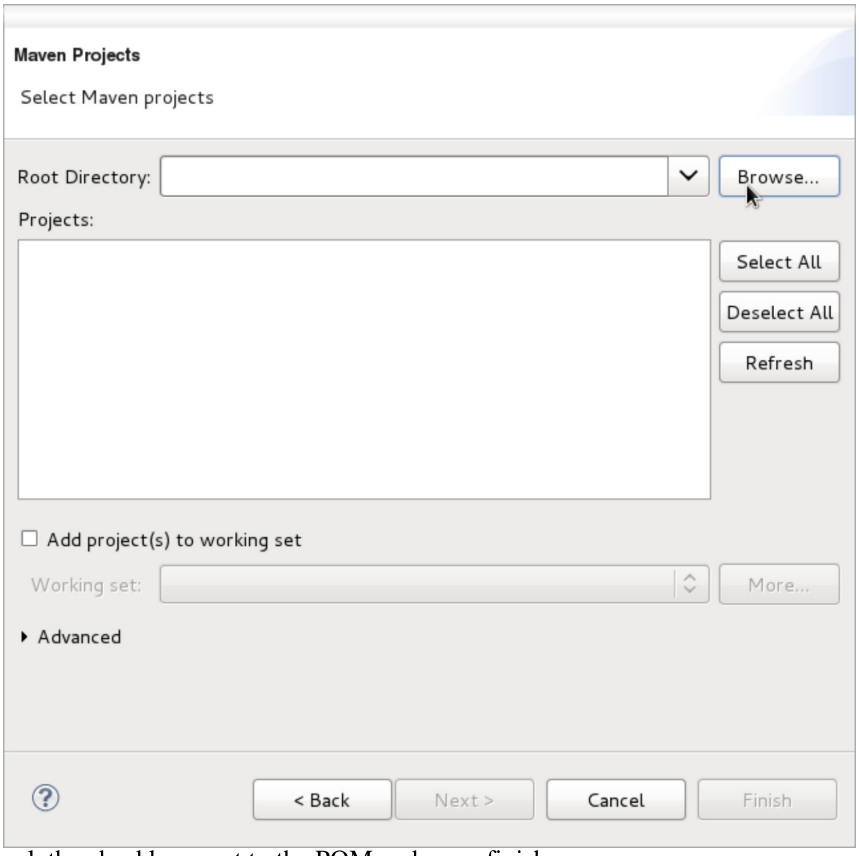
- You require Maven for ZK Studio/Eclipse, if you do now have it you can follow this tutorial on installing it [1]

Set-up the ZK Essentials sample

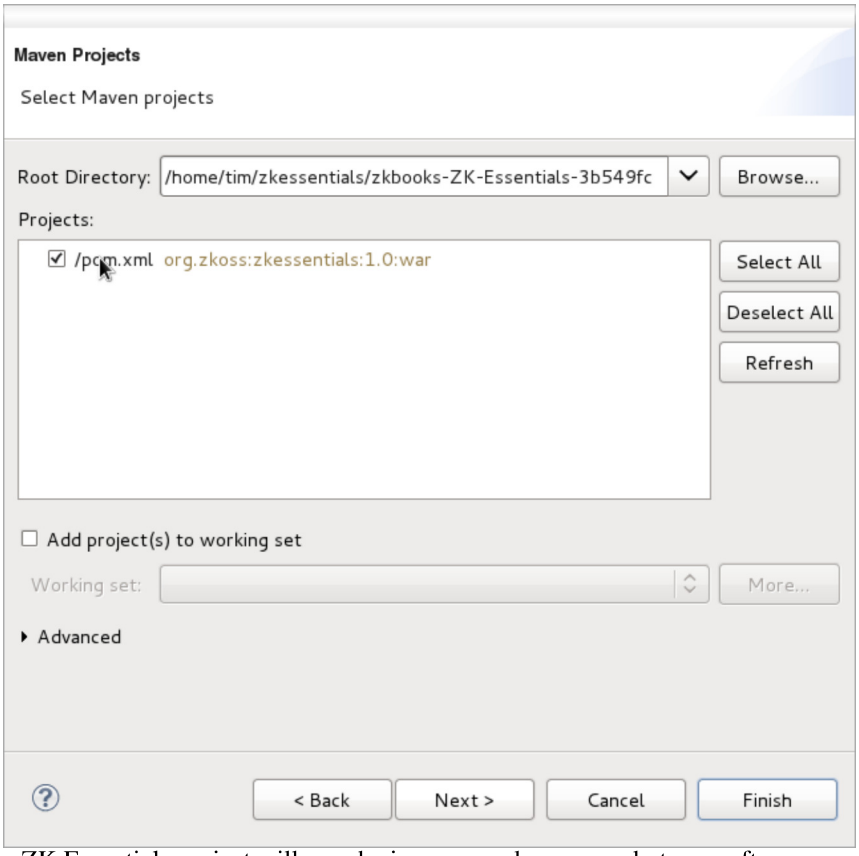
1. For this section we will assume you download the application first rather than do a git checkout from Maven
2. Therefore click here [1] to download the store without a database, and unzip to any folder you want
3. In ZK Studio go to New -> Import, and type "Existing Maven Projects", highlight the option and click next



4. Click browse and locate the extracted zkessentials directory and press OK



5. Check the checkbox next to the POM and press finish



6. The ZK Essentials project will now be in your workspace ready to run after maven downloads all the appropriate files

References

[1] https://github.com/zkbooks/ZK-Essentials/zipball/withdb

Laying out Your ZK Components

In the first section, we learned about the foundations of the ZK framework: components and events. Now we're ready to embark our effort into creating a shopping cart application using ZK. We will begin by learning how to lay out the look of a ZK application first.

How to Layout Components

Taking Web Pages to a Higher Dimension with Ajax

The content in a static web page is bounded by its page size and its layout; for example, introducing new information to a static web page means redirecting to a new page. With Ajax, new information and presentation could be dynamically loaded onto a page by replacing pieces of the existing DOM elements, making an Ajax enabled web page analogous to have a higher order of dimension than a traditional HTML page. A clear illustration of that is the tabs (Tabbox [1]) widget, which allows document content to be stacked up in an area with fixed size on a single page. In the table below, we summarize how the Ajax elements enable us to manage the page layout freely:

Ajax Element	Usage	Demo
Overlay	Stack contents up in a fixed area on the same page	Tabbox [2]
Folding	Allow "show and hide" for contents dependent on other contents	Hierarchical Grid [3], Master Detail [4],
Pop-up	Allow contents contained in an area to be evoked on top of its parent page with its position adjustable	Modal Window [5], Pop-ups [6],
Collapsible / Closable	Allow contents to be hidden so page space is disclosed	Portal Layout [7], Border Layout [8]
Splittable	Allow the contents area size to be adjustable	Splitter [9], Border Layout [8]

A Rundown on ZK's Layout Components

ZK offers a variety of layouts that incorporates these elements to bring flexible layout to meet the demands of building diverse web applications.
Please see the following references for these ZK components:

- Border Layout
  - Component Reference
  - Demo [8]
- Portal Layout
  - Component Reference
  - Demo [7]
- Column Layout
  - Component Reference

    - Demo [10]
- Table Layout
    - Component Reference
    - Demo [11]
- Tab Box
    - Component Reference
    - Demo [2]
- Group Box
    - Component Reference
    - Demo [12]
- Vlayout
    - Component Reference
    - Demo [13]
- Hlayout
    - Component Reference
    - Demo [13]

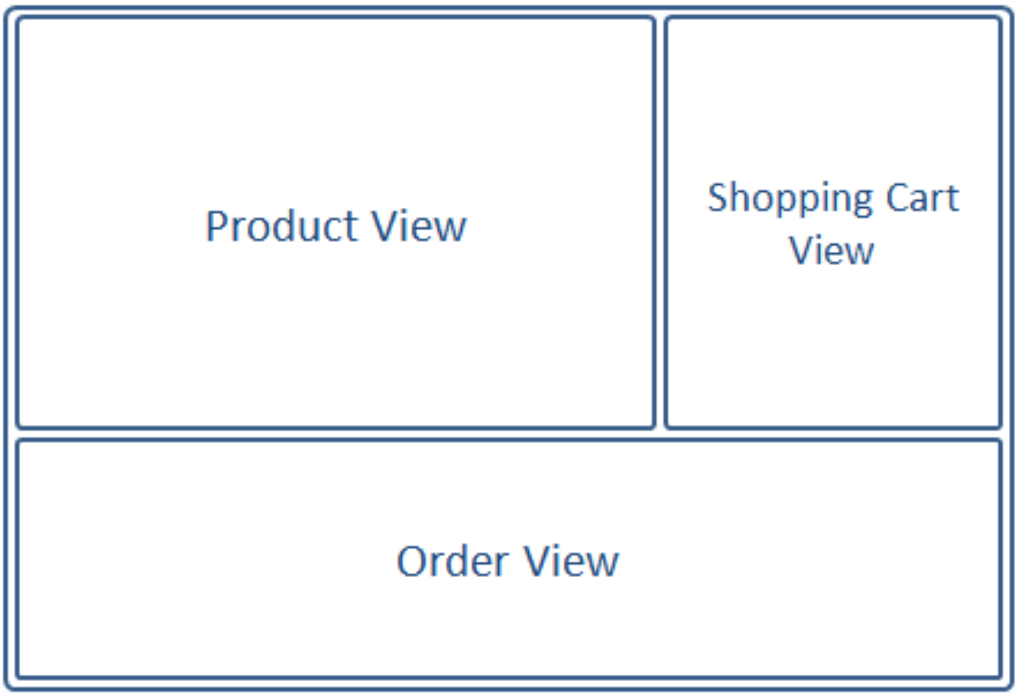
References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Tabbox.html#
[2] http://www.zkoss.org/zkdemo/tabbox
[3] http://www.zkoss.org/zkdemo/grid/hierarchy
[4] http://www.zkoss.org/zkdemo/grid/master_detail
[5] http://www.zkoss.org/zkdemo/window/modal_dialog
[6] http://www.zkoss.org/zkdemo/menu/pop-ups
[7] http://www.zkoss.org/zkdemo/layout/business_portal
[8] http://www.zkoss.org/zkdemo/layout/border_layout
[9] http://www.zkoss.org/zkdemo/layout/splitter
[10] http://www.zkoss.org/zkdemo/layout/column_layout
[11] http://www.zkoss.org/zkdemo/layout/table_layout
[12] http://www.zkoss.org/zkdemo/layout/group_box
[13] http://www.zkoss.org/zkdemo/layout/light_boxes

Using ZK Borderlayout

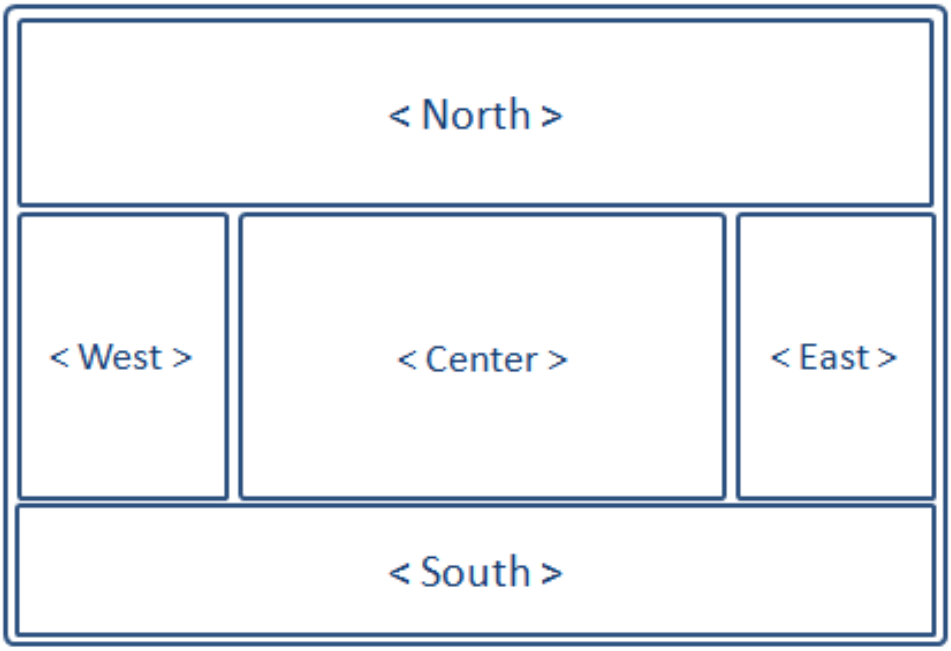
Suppose we have the following requirements in our shopping cart application:

- Product View - A table displaying the available products to purchase
- Shopping Cart View - A list of items placed in the shopping cart
- Order View - A record of all the orders placed



Introducing Border Layout and Its Attributes

The borderlayout divides a page into five sections: North, East, Center, West, South.



The attributes involved in the configuration of border layout is listed in the table below:

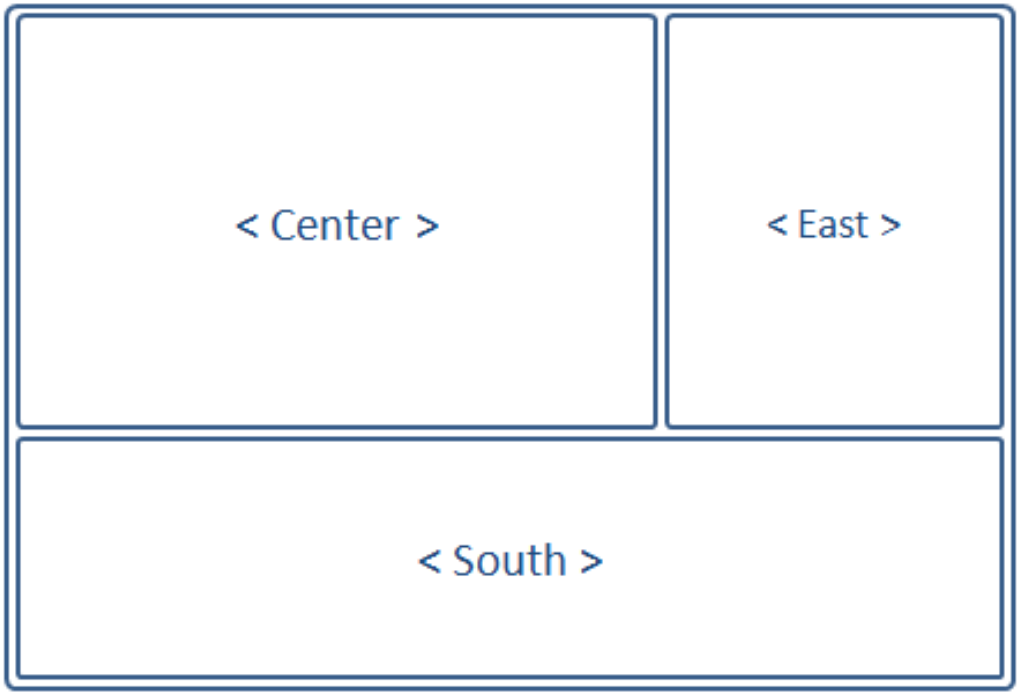
Attributes	Sample	Description
size	<north size="20%">...</north> <south size="180px">...</south>	set size in pixels or a percentage relative to its parent component
border	<east border="normal">...</east>	show/hide border; "normal" shows border, while "none" hides border
collapsible	<west collapsible="true">...</west>	allow the whole division to show/hide
maxsize/minsize	<south splitter="true" maxsize="500px" minisize="200px">...</south>	set the maximum and minimum allowable size for a component;
splittable	<north splittable="true">...</north>	allow the contents area size to be adjustable

Why Use Borderlayout

Border Layout stands out as the obvious choice because it supports:

- splittable [9], so we could adjust the dimensions of the views by dragging the splitters.
- collapsible [8], so views could be collapsed to make room for other views.

With the Borderlayout [1], we could outline the shopping cart application like the following:



The borders can be made adjustable (splittable="true"), and the east and south components can be collapsed to give room to the center component displaying the Product View.

Implementing the Views

We start implementing the Views for our shopping cart sample application by first making the divisions of the Views using borderlayout:

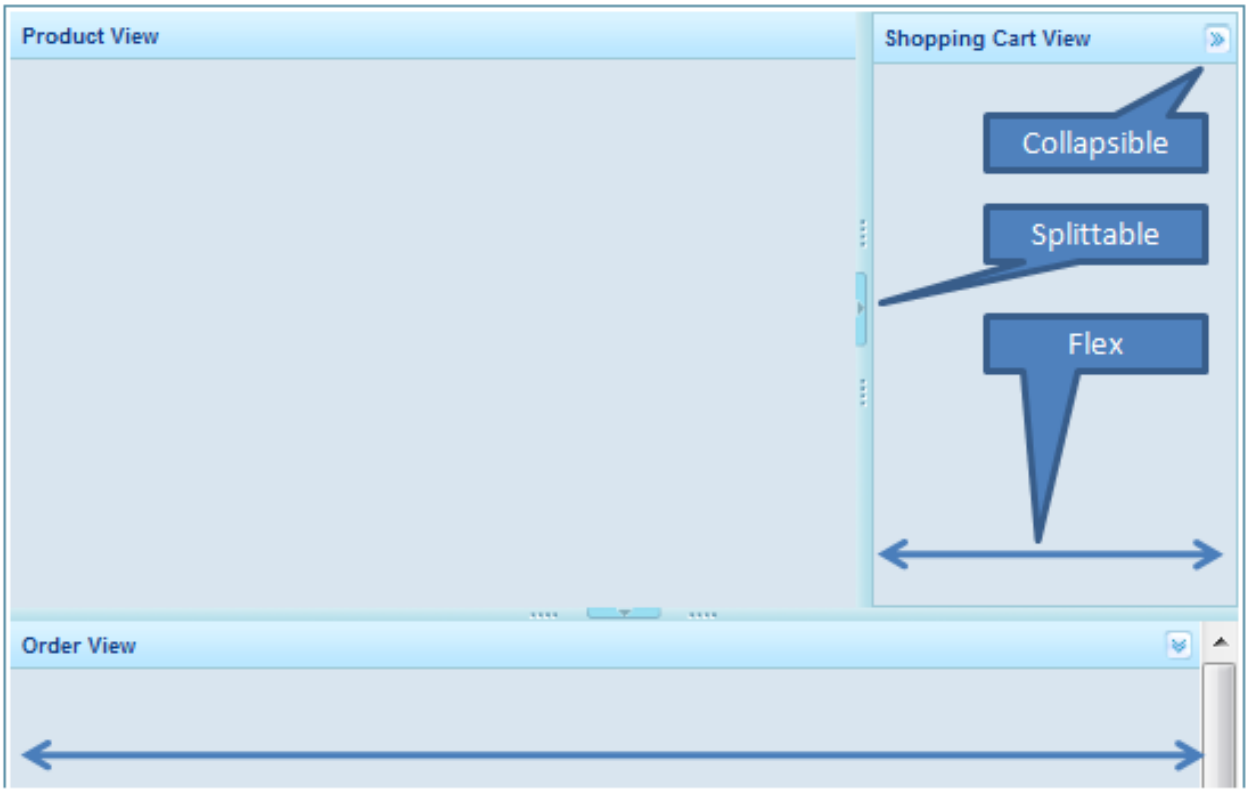
```
<window border="normal" width="100%" height="900px">

   <borderlayout>

      <center title="Product View" border="0">
            <div id="PrdoDiv">
```

```
                //Product View Implementation
            </div>
        </center>

        <east  title="Shopping Cart View" size="30%" flex="true" splittable="true" collapsi
            <div style="background:#D9E5EF; height:100%" apply="demo.web.ui.ctrl.Shopping
                 //Shopping Cart View Implementation
            </div>
        </east>

        <south  title="Order View" size="250px" flex="true" border="0" splittable="true" co
            <div>
                //Order View Implementation
            </div>
        </south>

    </borderlayout>
```

Here we note a few key implementations:

- Since the size of the borderlayout is not specified, its dimensions will assume those of its parent component Window [1].
- The borderlayout component itself does not have the "border" attribute, only its children (north, west, etc..) do.
- north, east, center, west, and south, are allowed to have only one child component; therefore, you should use container components such as Div [2] or Window [1] to wrap multiple components when placing them in borderlayout's children components.
- The center component does not take any size attributes; its height is dependent on the dimensions of north/south, whereas its width is dependent on the dimensions of east/west.
- The splittable and collapsible attributes are set to "true" to allow the dimensions of east and south to be adjusted dynamically by the user.
- The apply attribute is used here to assign a controller to the container component so components within it are auto-wired with data objects and events are automatically forwarded to event handlers.

The Finalized Layout



References

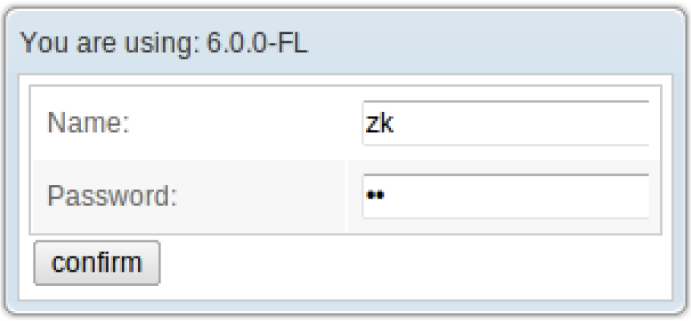
[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Borderlayout.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Div.html#

Handling the Login Process using ZK MVC and Sessions

Having established a layout for the application we move onto dealing with user authentication. In this section we will take a look at how to build your login using ZUL files, ZK MVC pattern of development and Sessions.

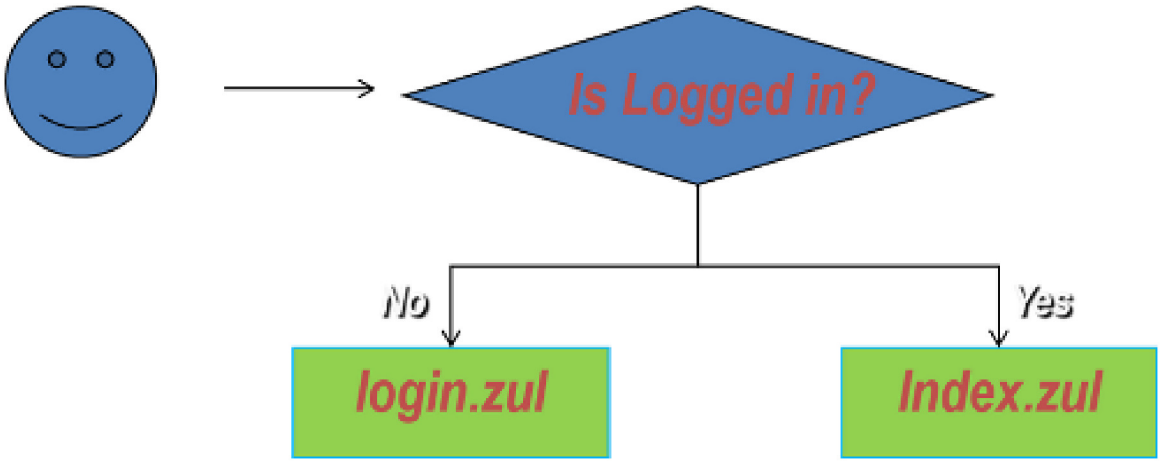
The Basic Login

Here we'd like a simple login page that prompts user for their name and password:



Login Behavior

Before starting to implement the login, first let us define the behavior that is required in the use case. The use case below represents the flow we would like to deliver to users upon entry or login to the system.



Building the Login UI

The first thing we need to do is to create the log-in interface using a ZUL file. The basic ZUL markup representing our login form is shown below.

<syntax lang="xml"> <?page title="ZK Store - Login"?> <window id="loginWin" border="normal" width="300px" title="You are using: ${desktop.webApp.version}"> <grid> <rows> <row> Name: <textbox id="nameTxb"/></row> <row> Password:<textbox id="passwordTxb" type="password"/></row> </rows> </grid> <button id="confirmBtn" label="confirm"/> <label id="mesgLbl"/> </window> </syntax>

The ZUL representation is very simple and consists of a Grid which aids the layout of the Textbox along with a Button to perform the confirm action and a Label to notify the user of any issues with the login process. At the moment the ZUL file represents a dumb login page as the button is not mapped to an action.

The recommended way to add functionality to your UI is to follow the MVC pattern of development. The following section details how to implement ZK using the MVC pattern.

Implementing ZK MVC

The MVC is developers' favorite pattern as it neatly separates various application layers in a clear manner. ZK fosters this process by allowing UI declaration to be done using an XML declarative language. It should be noted, however, that we can create Swing programmatic UIs using Richlets.

ZK MVC revolves around two key items, the SelectorComposer [1] and the apply attribute. The SelectorComposer [1] is a utility class which adds auto-wire functionality to provide access to UI objects from Java code without any effort. Let's see the demonstration of how to add this functionality into the login page.

Creating and Applying a Controller (SelectorComposer)

First, we need to create a controller class named LoginViewCtrl which extends from the SelectorComposer [1], this is a trivial matter in Java.

```
public class LoginViewCtrl extends SelectorComposer<Window>
```

SelectorComposer requires a control type, in this case we use a Window as the root component we will apply the composer to will be a Window. For ease of use developers can use Component as SelectorComposer's type allowing the controller to work with any component as the root.

To create this class, we need to link it to our ZUL file. This has to be done by indicating the apply attribute.

```
<?page title="ZK Store - Login"?>
<window id="loginWin" border="normal" width="300px"
      title="You are using: ${desktop.webApp.version}"
      apply="demo.web.ui.ctrl.LoginViewCtrl" mode="overlapped"
      position="center,center">
      <grid>
            <rows>
                  <row>
                        Name:
                        <textbox id="nameTxb" />
                  </row>
                  <row>
                        Password:
                        <textbox id="passwordTxb" type="password" />
                  </row>
            </rows>
      </grid>
      <button id="confirmBtn" label="confirm" />
      <label id="mesgLbl" />
</window>
```

As demonstrated in the code above, we set the apply attributes value to a full class name with path. For the above this class would be <mp>demo.web.ui.ctrl.LoginViewCtrl</mp>.

Auto-Wiring Components and Events

When the SelectorComposer [1] is applied, we can declare the components within the class as private variables which will be auto wired with the equivalent ZUL components when the annotation @Wire is present.

<syntax lang="java" high="3,4,6,7"> public class LoginViewCtrl extends SelectorComposer<Window> { @Wire private Textbox nameTxb, passwordTxb; @Wire private Label mesgLbl; } </syntax>

You can now access the UI components on the server-side and interact with them. All the wiring and communication is taken care of by ZK automatically. There are many more advanced @Wire options for more infomation on those please click here.

The next step is to enable the capturing of events. In this specific case we need to capture the <mp>onClick</mp> event of the button <mp>confirmBtn</mp>. To do so, we need to create a method with a specific annotation. In this case, we can use the event name followed by a an = then a # to dictate that the component ID will come next then the component ID. The method is as follows:

<syntax lang="java"> @Listen("onClick=#confirmBtn") public void confirm() { doLogin(); } </syntax>

Notice that the method does not take any parameters, however, if necessary, the method can take an Event [2] object as a parameter or the subsequent subclass which is applicable to the context. The @Listen annotation takes a selector string, for more information on how to construct selectors please refer to here.

The doAfterCompose Method

Lastly, we need to consider that people coming to the login.zul page may already be logged in just as our use case indicates. Therefore, we need to set up a mechanism that checks whether that is the case before loading the UI. This is done by overriding the SelectorComposer [1] method SelectorComposer.doAfterCompose(org.zkoss.zk.ui.Component) [3] which would be called when the events and components are wired. Thus, if you override this method, please make sure to call the super method. At this stage, your doAfterCompose(org.zkoss.zk.ui.Component) [3] method should look like this:

<syntax lang="java"> public void doAfterCompose(Component comp) throws Exception { super.doAfterCompose(comp);

//to be implemented, let's check for a login } </syntax>

Now we have the insights of what advantages ZK provides us with by wiring the components, events, and data automatically. We need to move on to deal with our goal of implementing a user management system to check credentials and redirect it according to our use case. The next section walks you through how user credentials is implemented using a session.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/SelectorComposer.html#

[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Event.html#

[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/SelectorComposer.html#doAfterCompose(org.zkoss.zk.ui.Component)

Managing credentials using ZK Sessions

One of the paradigms used within the essentials guide is the singleton managers which live within the session. The basic premise for storing singletons in the session is the fact that they are available anywhere at any time and are also specific to user credentials. When the session expires, the login credentials are conveniently cleared.

The Credentials Manager for User Authentication

For managing credentials we create a singleton named <mp>UserCredentialManager</mp> which wraps a <mp>UserDAO</mp>. Additionally, the <mp>UserCredentialManager</mp> will unveil two <mp>getInstance</mp> methods, one takes a Session [1] and the other one does not. The basic concept is that when the instance is retrieved, it checks the Session [1] for an existing credential manager. If there is no present, it creates a new one.

The code below demonstrates the two <mp>getInstance</mp> methods along with the creation of the <mp>userDAO</mp> in the constructor.

<syntax lang="java" high="11,18"> public class UserCredentialManager {

private static final String KEY_USER_MODEL = UserCredentialManager.class.getName()+"_MODEL"; private UserDAO userDAO; private User user; private UserCredentialManager(){ userDAO = new UserDAO(); } public static UserCredentialManager getIntance(){ return getIntance(Sessions.getCurrent()); } /** * * @return */ public static UserCredentialManager getIntance(Session zkSession){ HttpSession httpSession = (HttpSession) zkSession.getNativeSession(); // Session session = Executions.getCurrent().getDesktop().getSession(); // Session session = Executions.getCurrent().getSession(); Session session = Sessions.getCurrent(); synchronized(zkSession){ UserCredentialManager userModel = (UserCredentialManager) zkSession.getAttribute(KEY_USER_MODEL); if(userModel==null){ zkSession.setAttribute(KEY_USER_MODEL, userModel = new UserCredentialManager()); } return userModel; } } } </syntax> The manager is very standard exposing a login method which if successful sets the <mp>User</mp> object and a <mp>isAuthenticated</mp> method which checks to see whether the user is null and returns accordingly. Having put this into place we can now make use of it in our controller to change the page flow of the application.

Redirecting the User Depending on the UserCredentialManager

If we think back to our login page and the use case scenarios, we have two situations we need to check for a valid user: one is when the users navigate to the page and the other is when the users press the confirm button. If the user is authenticated, we need a way to redirect to the index page. This should be dealt with the Execution [2] class which provides information about the current execution, such as the request parameters.

A SelectorComposer contains an Execution object, execution, which is accessible. If we need to access it outside of a SelectorComposer [1] we can retrieve the current execution by calling the getCurrent() [3] method in the Executions [4] class as it is static.

The Executions.sendRedirect(java.lang.String) [5] class has a method named Executions [4], which redirects the user to a page you indicate. In this case, it is “index.zul.”

<syntax lang="java" high="2"> if(UserCredentialManager.getIntance(session).isAuthenticated()){ execution.sendRedirect("index.zul"); } </syntax>

This concludes the login topic, and the next session we will see how to display information to users using a Grid [6] and Listbox [7].

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Session.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Execution.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#getCurrent()
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#sendRedirect(java.lang.String)
[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Grid.html#
[7] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Listbox.html#

Displaying Information Using Grid, MVC and composite components

In the last section, we learned how to handle the login process using the MVC. This section builds on the MVC knowledge and also introduces the conept of composite components which can be used with MVC and MVVM patterns.

Displaying the Data

Data presentation in ZK involves preparing the Model and Controller in association with the View (ZK components). Once an association is established among these three parts, the data presented in ZK components are automatically updated whenever the data model is changed. In the previous section, we declared the basic UI skeleton, now we'll look into how to prepare a data model and use Template component to populate the Grid component in our Product View.

Associate View and Model in Controller

To associate our Product View (Grid component) with a Model, we'll implement a controller class which extends the ZK utility class SelectorComposer [1]. In its SelectorComposer.doAfterCompose(org.zkoss.zk.ui.Component) [3] method, the ZK components declared in mark up are wired with the component instances declared in the controller for our manipulation, while the events fired are automatically forwarded to this controller for event handling. Therefore, in our controller class, we override and call SelectorComposer.doAfterCompose(org.zkoss.zk.ui.Component) [3] method of its super class and implement the code that assigns a model and renderer to our Product View Grid.

For more information on the intracacies of this relationship please refer to link needed here.

```
public class ProductViewCtrl extends SelectorComposer<Div> {

	@Wire
	private Grid prodGrid;


	@Override
	public void doAfterCompose(Div comp) throws Exception {
		super.doAfterCompose(comp);

		List<Product> prods = DAOs.getProductDAO().findAllAvailable();

		ListModelList<Product> prodModel = new ListModelList<Product>(prods);
		prodGrid.setModel(prodModel);
	}

	...

}
```

Notice that the Grid reference "prodGrid" in **ProductViewCtrl.java** is auto-wired with the Grid declared in mark up whose ID is "prodGrid".

```
<div id="PrdoDiv" apply="demo.web.ui.ctrl.ProductViewCtrl">
	<grid id="prodGrid">
		<columns sizable="true">
			<column image="/image/Bullet-10x10.png"
					align="center" width="100px" />
			<column label="Name" width="100px" />
			<column label="Price" width="50px" />
			<column label="Quantity" width="50px" />
			<column label="Arrive Date" width="110px" />
			<column label="operation" />
		</columns>
		    <template name="model">
		    ...
		    </template>
	</grid>
```

```
</div>
```

Wrapping Data Object with ListModel

The implementations of ZK's ListModel [1] interface provide a wrapper for Java Collections to work with ZK's data modeling mechanisms. For Grid [6] and Listbox [7] components, we use the ListModelList [2] implementation; adding or removing entries in the ListModelList [2] would cause the associated Listbox [7] to update its content accordingly.

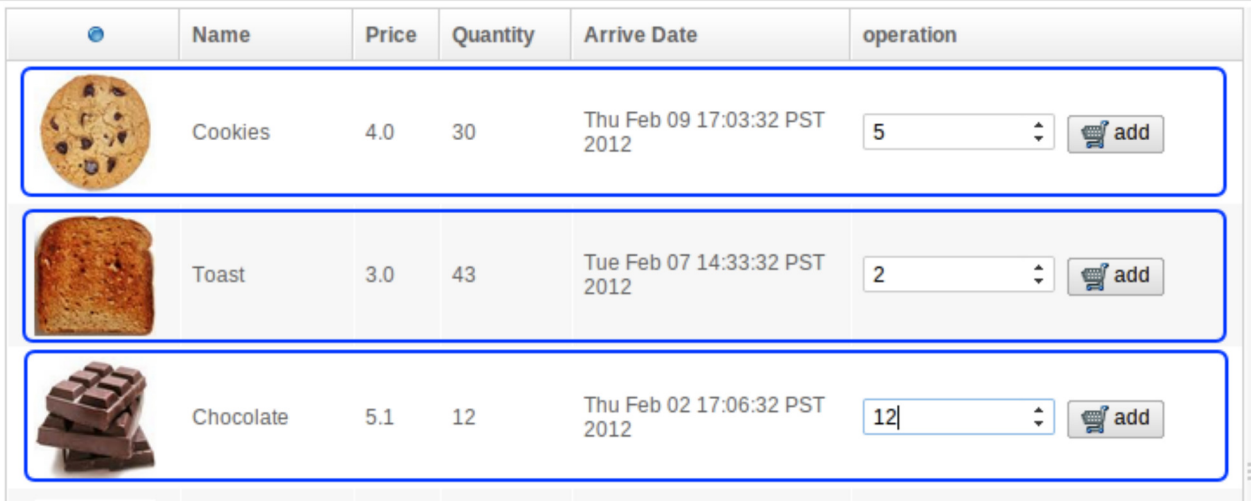
In the snippet below, the ListModelList takes in a product list fetched from a DAO object as its argument. At line 6, we assign the ListModelList object as the model for the Product View Grid by using Grid's setModel method.

ProductViewCtrl.java

```
public void doAfterCompose(Component comp) throws Exception {

            List<Product> prods = DAOs.getProductDAO().findAllAvailable();

            ListModelList<Product> prodModel = new ListModelList<Product>(prods);
            prodGrid.setModel(prodModel);

        ...
            });
    }
```

Using The Template Component to Populate Rows in Grid

Now that the model is prepared, we'll make use of the Template component which affords developers the ability to define ZUML fragments which will be used when rendering each row.



In this case our ZUML fragment needs to contain a row and then a control for each column. Here is the final template to recreate the interface.

```
<grid id="prodGrid">
    <columns sizable="true">
        ...
    </columns>
    <template name="model">
        <row value="${each}">
            <image height="70px" width="70px"
                src="${each.imgPath}" />
```

```
                    <label value="${each.name}" />
                    <label value="${each.price}" />
                    <label value="${each.quantity}" />
                    <label value="${each.createDate}" />
                    <productOrder maximumQuantity="${each.quantity}" product="${each}" />
                </row>
            </template>
</grid>
```

A template is a ZUML fragment that defines how to render data model in its children components and can contain any ZUML element you would like including other templates. It is generally used within a Listbox, Grid, Tree and Combobox to describe how repeating items are displayed. In this case the template defines the layout of each row. The row value is assigned as "each" where each refers to a bean provided by the model.

The syntax of the template looks incredibly similar to what one observed in the databinding chapter except using a $ instead of an @. This is because the markup below makes use of EL. EL expressions differ from databinding in the fact that they are only interpreted once, whereas databidning will be re-interpreted whenever it is notified a value has changed. Seeing as the product list will remain relatively constant, we'll use EL here.

For more information on EL please visit link needed here

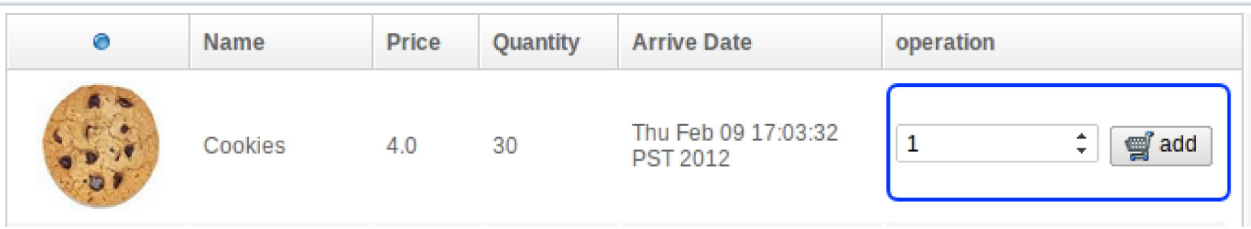
In this case, the Template component's output is relatively simple, however, one interesting item in the Template is the productOrder component. This is in fact a composite control which is created by developers which we'll look into in the next section.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModel.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModelList.html#

Composite Components in the MVC

In this section, we'll see how to use ZK's Composite Component pattern to implement a product order component that consists of a label, a spinner, and a button in the Product View.



Declaring a Composite Component

A composite component is an advanced template that affords developers the ability to extend from any component. A composite component consists of a Java class which extends from a component, in this case the ProductOrder component extends from Cell as it will be placed into a grid row. The Cell will therefore contain multiple components.

ProductOrder also implements IdSpace and AfterCompose; these two interfaces enable the component to take on additional behaviours. By implementing AfterCompose, when the composite component is created it enbables developers to perform actions such as wiring of variables or registering event listeners. In the case of the ProductOrder the snippet below shows the afterCompose method:

```
public class ProductOrder extends Cell implements IdSpace, AfterCompose
 {

      ...

      public void afterCompose() {
            // 1. Render the ZUML fragment

Executions.createComponents("/WEB-INF/composite/productorder.zul",
                        this, new HashMap<String, Object>() {
                              private static final long
serialVersionUID = 7141348964577773718L;

                              {
                                    put("maximumQuantity",
getMaximumQuantity());
                              }
                        });

            // 2. Wire variables, components and event listeners
(optional)
            Selectors.wireVariables(this, this, null);
            Selectors.wireComponents(this, this, false);
            Selectors.wireEventListeners(this, this);
      }
```

```
        ...

}
```

Rendering the Composite Component

The snippet above shows two things happening, firstly the composite component renders a ZUML fragment which will be contained within itself. Taking a look at the Executions.createComponents function call on line 7, the first parameter is the location of the ZUML fragment, the second parameter is the component which will become the root, in this case itself. The third component is a HashMap containing the parameters to send to the fragment, in this case the maximumQuantity, which is just a simple bean.

The fragment is shown below:

```
<zk>
      <spinner id="spnQuantity" constraint="min 1 max ${arg.maximumQuantity}" value="1"/>
      <button id="btnAdd" label="add" image="/image/ShoppingCart-16x16.png" />
      <label id="lblError" />
</zk>
```

In the above fragment one can see how the maximum quantity, which was passed to the fragment on creation, is accessed and used using EL expressions.

Wiring Variables and Event Listeners

Finally, within the composite component, variables, components and event listeners need to be wired. This is easy to do using the following method calls in the afterCompose function.

```
Selectors.wireVariables(this, this, null);
Selectors.wireComponents(this, this, false);
Selectors.wireEventListeners(this, this);
```

Using the Composite Component

Having built the component it now needs to be registered with ZK, this is easily done by putting a directive at the top of the ZUL file you want to use it in. In the case of this application it is present in index.zul.

```
<?component name="productOrder" class="demo.web.ui.ctrl.ProductOrder" ?>
```

The directive specifies a name for the component which can be referenced in the ZUL file and the appropriate class which drives the functionality. The component can now be used as follows:

```
<grid id="prodGrid">
      <columns sizable="true">
            ...
      </columns>
      <template name="model">
            <row value="${each}">
                  <image height="70px" width="70px"
                        src="${each.imgPath}" />
                  <label value="${each.name}" />
                  <label value="${each.price}" />
                  <label value="${each.quantity}" />
```

```
                <label value="${each.createDate}" />
                <productOrder maximumQuantity="${each.quantity}" product="${each}" />
            </row>
        </template>
</grid>
```

For more information on the composite component please consult the Developer's reference

This concludes this section of the document, the data is displayed correctly. The adding of products is discussed in the last chapter as it deals with communication between MVC and MVVM thus is a topic best introduced after MVVM. If you are interested you can take a look now.

Displaying information using the Listbox and MVVM

The last section, discussed using the MVC and composite components. This section introduces the new ZK 6 pattern MVVM which is the recommended way of implementing ZK applications.

What is the MVVM

Introduction of MVVM

MVVM[1] is an abbreviation of a design pattern named **Model-View-ViewModel** which originated from Microsoft.[2] It is a specialization of Presentation Model[3] introduced by Martin Fowler, a variant of the famous MVC pattern. This pattern has 3 roles: View, Model, and ViewModel. The View and Model plays the same roles as they do in MVC.

The ViewModel in MVVM acts like *a special Controller* for the View which is responsible for exposing data from the Model to the View and for providing required action and logic for user requests from the View. The ViewModel is type of *View abstraction*, which contains a View's state and behavior. But *ViewModel should contain no reference to UI components* and knows nothing about View's visual elements. Hence there is a clear separation between View and ViewModel, this is also the key characteristics of MVVM pattern. From another angle, it can also be said that the View layer is like an UI projection of the ViewModel.

Strength of MVVM

Separation of data and logic from presentation

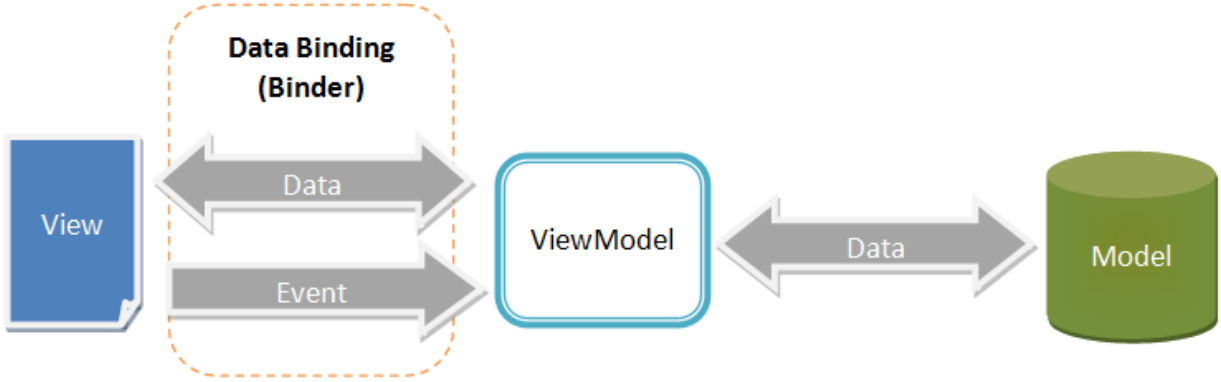
The key feature is that ViewModel knows nothing about View's visual elements guarantees the one way dependency from View to the ViewModel thus avoiding mutual programming ripple effects between UI and the ViewModel. Consequently, it brings the following advantages:

- It's suitable for **design-by-contract** programming. [4] As long as the contract is made (what data to show and what actions to perform), the UI design and coding of ViewModel can proceed in parallel and independently. Either side will not block the other's way.
- **Loose coupling with View**. UI design can be easily changed from time to time without modifying the ViewModel as long as the contract does not change.

- **Better reusability**. It will be easier to design different views for different devices with a common ViewModel. For a desktop browser with a bigger screen, more information can be shown on one page; while for a smart phone with limited display space, designing a wizard-based step-by-step operation UI can be done without the need to change (much of) the ViewModel.
- **Better testability**. Since ViewModel does not "see" the presentation layer, developers can unit-test the ViewModel class easily without UI elements.

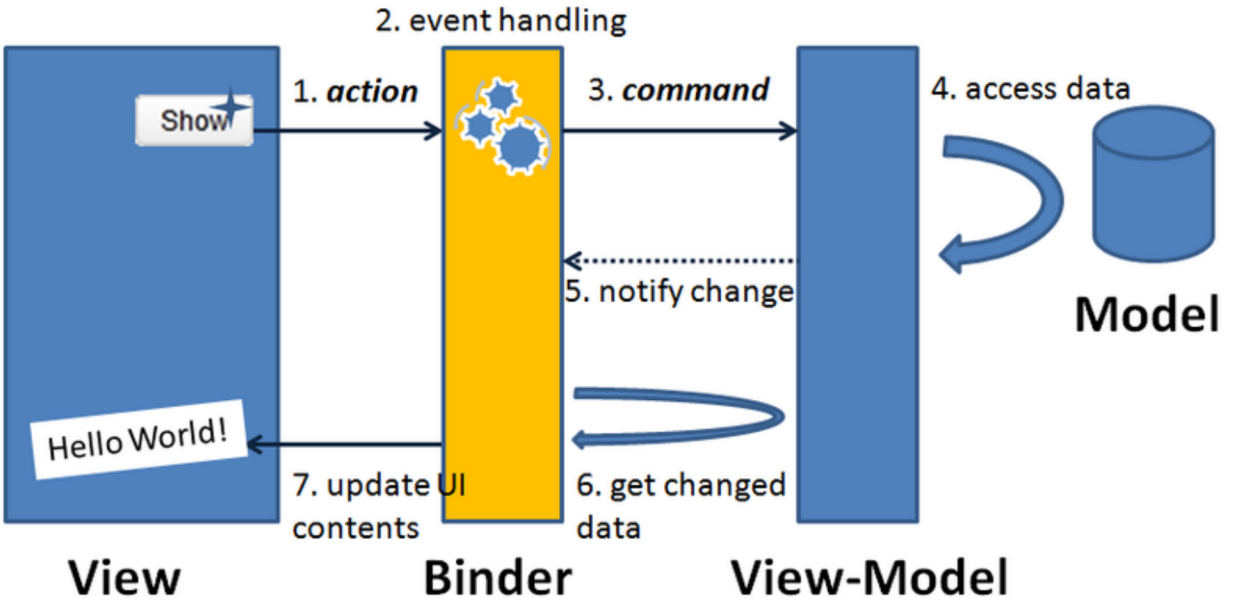
MVVM & ZK Bind

Since the ViewModel contains no reference to UI components, it needs a mechanism to synchronize data between the View and ViewModel. Additionally, this mechanism has to accept the user request from the View layer and bridge the request to the action and logic provided by the ViewModel layer. This mechanism, the kernel part of the MVVM design pattern, is either data synchronising codes written by the application developers themselves or a data binding system provided by the framework. ZK 6 provides a data binding mechanism called **ZK Bind** to be the infrastructure of the MVVM design pattern and the **binder** [5] plays the key role to operate the whole mechanism. The binder is like a broker and responsible for communication between View and ViewModel.



Detail Operation Flow

Here we use a simple scenario to explain how MVVM works in ZK. Assume that a user click a button then "Hello World" text appears. The flow is as follows:



As stated in the paragraph earlier, the binder synchronizes data between UI and ViewModel.

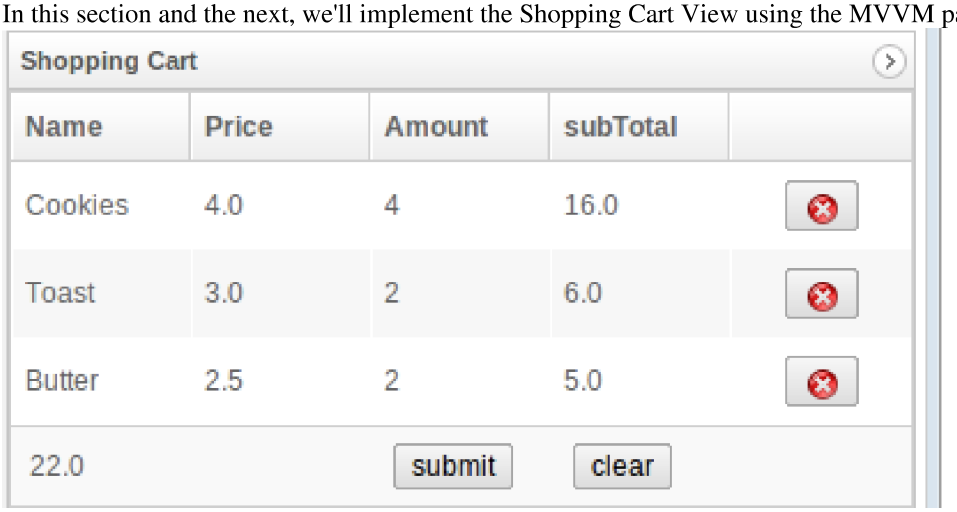
1. A user presses a button on the screen (perform an action).
2. A corresponding event is fired to the binder.
3. The binder finds the corresponding action logic (It is **Command**) in the ViewModel and executes it.
4. The action logic accesses data from Model layer and updates ViewModel's corresponding properties.
5. ViewModel notify the binder that some properties have been changed.
6. Per what properties have been changed, the binder loads data from the ViewModel.
7. Binder then updates the corresponding UI components to provide visual feedback to the user.

References

[1] WPF Apps With The Model-View-ViewModel Design Pattern http://msdn.microsoft.com/en-us/magazine/dd419663.aspx
[2] Introduction to Model/View/ViewModel pattern for building WPF apps http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx
[3] Presentation Model http://martinfowler.com/eaaDev/PresentationModel.html
[4] Design by contract http://en.wikipedia.org/wiki/Design_by_contract
[5] binder ZK Developer's Reference/MVVM/DataBinding/Binder

Displaying Information from the View Model

In this section and the next, we'll implement the Shopping Cart View using the MVVM pattern.



To start with the author will now give an overview of what the final ZUL fragment for the MVVM will look like. This section will then teach developers how to create this step by step.

```
<div apply="org.zkoss.bind.BindComposer"
      viewModel="@id('vm')
@init('demo.web.ui.ctrl.ShoppingCartViewModel')">

      <listbox id="shoppingCartListbox"
            model="@load(vm.cartItems)"
            selectedItem="@bind(vm.selectedItem)">
            <listhead sizable="true">
                  <listheader label="Name" />
                  <listheader label="Price" />
                  <listheader label="Amount" />
                  <listheader label="subTotal" />
                  <listheader align="center" />
```

```
            </listhead>
            <template name="model" var="cartItem">
                  <listitem>
                        <listcell label="@load(cartItem.product.name)" />
                        <listcell label="@load(cartItem.product.price)" />
                        <listcell label="@load(cartItem.amount)" />
                        <listcell label="@load(cartItem.product.price * cartItem.amount)"
                        <listcell>
                              <button image="/image/DeleteCross-16x16.png" onClick="@comm
                        </listcell>
                  </listitem>
            </template>
            ...
      </listbox>
        ...
</div>
```

Initializing the BindComposer and ViewModel

The first thing that needs to be done is the initialization of the BindComposer which will manage binding for it's child components. This can be done simply using the familiar apply attribute to the parent control of your choice. In the case of this example the div which contains the listbox was chosen.

```
<div apply="org.zkoss.bind.BindComposer"
      viewModel="@id('vm')
@init('demo.web.ui.ctrl.ShoppingCartViewModel')">
```

Line 2 in the above snippet assigns a value to a *viewModel* property. This property is used for two things

1. Tell the binder which ViewModel class to instantiate
2. Assign a name to the instantiated class

In this case @id is used to set the name of the instantiated view model to 'vm' while @init is used to specify the class to instantiate. Having specified a view model class one now needs to create the said class.

Implementing The View Model

In the MVVM pattern a view model is an abstraction of Model. It extracts the necessary data to be displayed on the View from one or more Model classes. Those data are exposed through getter and setter method like JavaBean's property. To create a view model is easy, there is no need to extend from nor implement any other class. A view model is a POJO and exposes its data using getter and setter methods.

```
public class ShoppingCartViewModel {
      ...

      private CartItem selectedItem;

      public CartItem getSelectedItem() {
            return selectedItem;
      }

      public void setSelectedItem(CartItem selectedItem) {
```

```
            this.selectedItem = selectedItem;
      }

      public List<CartItem> getCartItems() {
            return UserUtils.getShoppingCart().getItems();
      }

      public ShoppingCart getShoppingCart() {
            return UserUtils.getShoppingCart();
      }
        ...
}
```

Binding Data to View Model

For loading & saving data ZK's MVVM pattern provides two general commands, @load and @bind. @load command will only load data and any change to the data from the UI will not be saved to the bean. @bind on the other hand provides both loading and saving of the bean so any changes to the values driven by the UI will be reflected server-side.

With this in mind it is time to start loading the data into the shopping cart. First thing to be done is load the model and selected item. This is easily done by accessing the getters and setters we implemented for the View Model:

```
<div apply="org.zkoss.bind.BindComposer"
      viewModel="@id('vm')
@init('demo.web.ui.ctrl.ShoppingCartViewModel')">

      <listbox id="shoppingCartListbox"
            model="@load(vm.cartItems)"
            selectedItem="@bind(vm.selectedItem)">

                  ...

      </listbox>
</div>
```

It is now possible to implement a template as outlined in previous sections to create each Listitem. Using databinding the resulting template is extremely simple.

```
      <listbox id="shoppingCartListbox"
            model="@load(vm.cartItems)"
            selectedItem="@bind(vm.selectedItem)">
            <listhead sizable="true">
                  ...
            </listhead>
            <template name="model" var="cartItem">
                  <listitem>
                        <listcell label="@load(cartItem.product.name)" />
                        <listcell label="@load(cartItem.product.price)" />
                        <listcell label="@load(cartItem.amount)" />
```

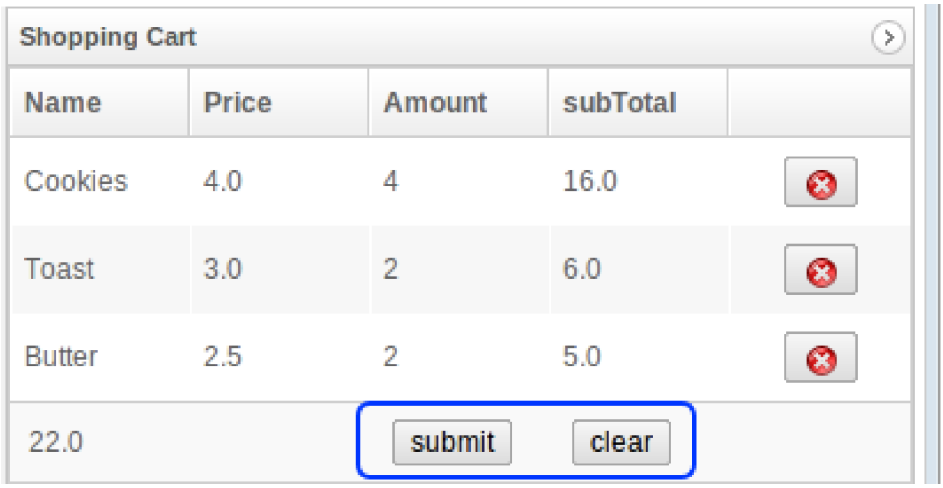
```
                        <listcell label="@load(cartItem.product.price * cartItem.amount)"
                        <listcell>
                              <button image="/image/DeleteCross-16x16.png" onClick="@comm
                        </listcell>
                    </listitem>
                </template>
                    ...
        </listbox>
```

Line 6 brings about an intersting use case as instead of just specifying a bean's property it specifies an equation. The binder is capable of handling expressions. There are many more expressions available for more information please click here link needed.

This takes care of basic loading and saving functionality, but what about actions such as clicking buttons? The next section introduces how to give commands based on component actions.

Performing Actions on Events

The previous section outlined how to push and pull data to and from the View Model. This section discusses how to catch events such as a button click in our Shopping Cart View under the MVVM pattern.



The @command Annotation

The method for invoking a command is rather simple and just requires two steps.

1. Register the command on the event in the ZUL file
2. Write a function in the View Model which will be fired when the command arrives

Both steps are exceptionally easy providing greater productivity to developers. The first step is demonstrated below:

```
<button id="submitOrderBtn" label="submit" onClick="@command('submitOrder')" />
```

In the above snippet a command is assigned to the button's onClick method. This command is named "submitOrder". On clicking the button the event will be fired and the command "submitOrder" will be passed to the binder. The binder will then attempt to execute the respective command from the View Model. Therefore one needs to exist. There are two ways to implement the command in the View Model:

1. Make the method name match the commmand name and annotate the method with @command
2. Annotate the function with @command("submitOrder")

In this case the author chose to match the method name with the command name "submitOrder"

```
@Command
@NotifyChange({"cartItems", "shoppingCart", "orderNote"})
public void submitOrder() {
	DAOs.getOrderDAO().createOrder(UserUtils.getCurrentUserId(),
getCartItems(), getOrderNote());

	...

	clearOrders();
}
```

The method in the View Model will therefore be called when the button is clicked. A keen observer may have noticed the @NotifyChange annotation on the method, the following section discusses this and the uses of it.

The @NotifyChange Annotation

The @NotifyChange annotation's purpose is funamental, if a method which is called by the binder changes a property that is bound then the developer needs to specify that. In the example below the code will change the property cartItems, shoppingCart and orderNote therefore these are included in the @NotifyChange annotation.

```
@Command
@NotifyChange({"cartItems", "shoppingCart", "orderNote"})
public void submitOrder() {
	DAOs.getOrderDAO().createOrder(UserUtils.getCurrentUserId(),
getCartItems(), getOrderNote());

	...

	clearOrders();
}
```

Notify change can adjust the value of one property, @NotifyChange("property"), more than one property @NotifyChange({"one", "two"}) and all properties @NotifyChange("*").

This now enables developers to write effective commands, though this is not the end of the power available. It is also possible to pass parameters to command methods.

Passing Information to @Command

Sometimes one will want to pass parameters to a command method, this can be done painlessly. Take the orders list for example as an example when pressing the red cross button the item should be deleted, to do this the easiest way is to pass the item to the command ready for it to be deleted at the backend. The item can be passed to the command using the following syntax:

```
<template name="model" var="cartItem">
	<listitem>
		<listcell label="@load(cartItem.product.name)" />
		<listcell label="@load(cartItem.product.price)" />
		<listcell label="@load(cartItem.amount)" />
		<listcell
```

```
                  label="@load(cartItem.product.price *
cartItem.amount)"
                  style="word-wrap: word-break" />
            <listcell>
                  <button
                        image="/image/DeleteCross-16x16.png"
                        onClick="@command('deleteOrder',
cartItem=cartItem)" />
            </listcell>
      </listitem>
</template>
```

Line 12 demonstrates the syntax, @command is used as normal but there is an extra attribute added, in this case it is cartItem=cartItem, the basic syntax is as follows:

```
nameThatYouAssign=variable
```

In the above case the template is defining the variable name to access the active bean as cartItem. This value can then be retrieved in the View Model using the command function and a parameter prepended with the annotation @BindingParam. Below shows this method at work.

```
@Command
@NotifyChange({"cartItems", "shoppingCart"})
public void deleteOrder(@BindingParam("cartItem") CartItem cartItem) {
      getShoppingCart().remove(cartItem.getProduct().getId());
}
```

The @BindingParam annotation specifies the name as "cartItem" this corresponds to the name as defined in the UI. After doing this the variable is accessible in the function with no further effort needed.

This concludes the teaching of knowledge on how to implement both the MVC and MVVM patterns for the product view, shopping cart and orders list. The last question is how do these 3 sections communicate. The product view uses MVC but needs to tell the shopping cart (which uses MVVM) that a product has been added to the order and the shopping cart needs to tell the orders list that there is a new order even though they both use different View Models.

The next session introduces how to do this.

Communicating between MVC and MVVM patterns and between View Models

Communication between the MVC, MVVM and view models sounds to be the most difficult task to perform, however, it is in fact very easy to do and extremely powerful. The first item of interest is dealing with communication between the MVC and MVVM.

Posting a command from the MVC to an MVVM view model

To offer communication there needs to be a way for the MVC pattern bound section of the application to pass a command to the binder which will in turn talk to all the view models and tell them to run a particular command. This is achieved by posting a global command and works in a much similar way to the normal command system.

The example below shows a function which was taken from our SelectorComposer (MVC pattern) and shows the issuing of a global command. In the product view upon clicking of the add button will result in the execution of said function.

```
@Listen("onAddProductOrder=#PrdoDiv #prodGrid row productOrder")
public void addProduct(Event fe) {

	if (!(fe.getTarget() instanceof ProductOrder)) {
		return;
	}

	ProductOrder po = (ProductOrder) fe.getTarget();

	try {
		UserUtils.getShoppingCart()
			.add(po.getProduct(), po.getQuantity());
	} catch (OverQuantityException e) {
		po.setError(e.getMessage());
	}

	BindUtils.postGlobalCommand(null, null, "updateShoppingCart",
null);
}
```

Line 17 shows the function "BindUtils" and its function "postGlobalCommand" is used to post a command to the binder. In this case the first two strings take the name of the binder and the scope of said binder, in this case it is set to null to use the default. In most cases one would want to set this to null. Then the string name for the command is given along with a map of arguments to pass, in this case null as there are no arguments.

This globalCommand is then executed by the binder on any view model that has registered for it. In the case of this application the ShoppingCartViewModel needs to refresh the cart items when a product is added to a cart. Therefore a function is created in the ShoppingCartViewModel and registered as a global command, this function has the ability to do some processing and then notify the binder that the cartItems in that view have changed. The snippet below shows this in action.

```
public class ShoppingCartViewModel {

	...

	@GlobalCommand
	@NotifyChange("cartItems")
	public void updateShoppingCart() {
		//no post processing to be done
	}
}
```

This takes care of passing commmands from the MVC composer to MVVM view model, the next section discusses message passing between MVVM view models.

Communication between MVVM view models

Communication between view models can be done in the same way as communication between MVC and MVVM by using the BindUtils to publish a global command from Java code and having a function registered as a GlobalCommand in the other view model. However, it is also possible to add a global command into the zul file. For example for the shopping cart view when submitting the order it needs to tell the order view to update, in the order view one registers the following function to capture a global command:

```
@GlobalCommand
@NotifyChange("orders")
public void updateOrders() {
	//no post processing needed
}
```

This function will updated the orders and it has the option to do post processing if necessary. The main question is how to invoke this, it is possible to do so using the postGlobalCommand function of the BindUtils class, the call to do this would be as follows:

```
BindUtils.postGlobalCommand(null, null, "updateOrders", null);
```

However, an easier way is to place this command into the zul file, so the submit order button structure looks like this:

```
<button id="submitOrderBtn" label="submit"
	onClick="@command('submitOrder') @global-command('updateOrders')"
	disabled="@load(empty vm.cartItems)" />
```

Here one notices that as well as having an @command assigned to the onClick attribute, it now also has an @global-command, which is equivalent to the java function above this snippet. This is the easiest way of providing the command.

This is all one needs to know on sending messages between the MVC and view models. The next section deals with adding Hibernate and a database into the mix.

Working with ZK and databases

This section will guide us through the process of re-implementing sections of our application to make good use of persistent storage using Hibernate.

The Theory of Using ZK with Databases

ZK is predominantly a server centric framework thus developers are not required to leverage any special mechanism to transfer data from the server to the client.

Fundamentally, as discussed in the previous sections, data is provided to the View using a model which is a collection type, usually a <mp>List</mp>.

With ZK handling, transferring the data to the client is substantially easier. A ZK developer is able to leverage any application or ORM framework such as Spring and Hibernate without the need to perform any special tasks.

If you are familiar with Hibernate you do not need to read any further as the standard List of objects retrieved from ORM frameworks is compatible with ZK's model concept. Thus from using the knowledge gained in the previous sections you can construct your own application.

This chapter introduces to those who are unfamiliar with ORM framworks to Hibernate and ZK. The following sections will provide a simple tutorial on refactoring our current application to make good use of persistent storage.

Using Hibernate with ZK

As established previously, the applications DAOs access an "in-memory" model provided by Java Lists. It is required to adjust the DAOs so they can access a persistent model powered by HSQLDB using Hibernate.

Hibernate is an ORM framework of choices for many large enterprises as it provides an easy way to access persistent storage. Hibernate can be combined with Spring and EJB3 to make the lifecycle management of applications and session trivial greater flexibility for developers. In this example only Hibernate will be used, however, there are many other examples of using Hibernate along with Spring available on the web.

HSQLDB is a relational database engine written in Java. It provides an easy way to persist data in a file store.

Setting up Hibernate

Hibernate requires configuration to work. Our sample application contains two configuration files placed in the folder included in the classpath:

1. log4j.properties 2. hibernate.cfg.xml

The log4j.properties file is used to inform Log4j of the level we wish to log. In our case we set the lowest level of logging possible which in this case is DEBUG and TRACE for the hibernate classes. Setting this to such a fine grained setting enables us to see exactly what is going on during hibernate.

The second file **hibernate.cfg.xml** contains detailed configuration for Hibernate in XML format. The content is provided and analyzed below.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//E
<hibernate-configuration>
  <session-factory>
```

```
    <property name="hibernate.dialect">org.hibernate.dialect.HSQLDialect</property>
    <property name="hibernate.connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="hibernate.connection.url">jdbc:hsqldb:file:data/store</property>
    <property name="hibernate.connection.username">sa</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.current_session_context_class">thread</property>
    <property name="hibernate.query.factory_class">org.hibernate.hql.classic.ClassicQuery


    <property name="hibernate.hbm2ddl.auto">create-drop</property>

    <mapping class="demo.model.bean.Order"/>
    <mapping class="demo.model.bean.OrderItem"/>
    <mapping class="demo.model.bean.User"/>
    <mapping class="demo.model.bean.Product"/>
  </session-factory>
</hibernate-configuration>
```

- Line 5 let hibernate know that we intend to use HSQLDialect as it is an HSQL database
- Line 6 specifies the jdbc driver class
- Line 7 is the path to the datastore, in this case informing HSQLDB that we will make use of a file named store in the folder data
- Line 8 is the username, in this case a default sa
- Line 9 dictates that we would like Hibernate to print out the SQL
- Line 10 states that the session context is thread based
- Line 11 states that the HQL style we would like to use is classic
  - HQL is Hibernates own query language. For more information please click here [1]
- Lines 14-19 are to do with definition of the entity types within the application and generation of the database. These topics are covered in the following section.

References

[1] http://docs.jboss.org/hibernate/core/3.3/reference/en/html/queryhql.html

Hibernate, Entities and Database Generation

Hibernate persists Objects to the database. To perform this task we need two things, a database schema and a set of mapped classes. In the previous section, we've explored the Hibernate configuration file and highlighted the following lines:

```
<property name="hibernate.hbm2ddl.auto">create-drop</property>

<mapping class="demo.model.bean.Order"/>
<mapping class="demo.model.bean.OrderItem"/>
<mapping class="demo.model.bean.User"/>
<mapping class="demo.model.bean.Product"/>
```

One of the most important properties in this XML file is “hibernate.hbm2ddl.auto” which will be set as “create-drop”. When Hibernate’s SessionFactory is set, it will automatically create an option which validates or exports the schema DDL to the database. This is very useful during the process of developing a data driven application as it eliminates the need to manually update SQL scripts when your schema needs to change. However, this setting is not recommended for production usage.

NOTE: As the database schema is generated and dropped at the beginning and end of every session, this means that the data is not persisted at the end of the application. By changing this to update the data it will be persisted across multiple runs.

The other properties enable the mapping of Java classes to Database tables. In general, this mapping is declared in two ways:

- By providing the class name using the attribute <mp>class</mp>
- By providing a URI to an XML resource which describes the mapping

Both mapping by XML or annotation can achieve the same results but selecting which implementation method to deploy will depend on the developer's preference. Some developers prefer XML while others incline towards annotations. In this case we chose to use annotations.

Using Hibernate Annotations

When instructing Hibernate that the class is an Entity, the Entity should be persisted the annotation and placed above the class declaration. For property declarations annotations can be placed directly above the field or on the getter of the property.

There are varying opinions on which is better and why, however, in this case we have chosen to place the annotation on the field value as it is clearer for demonstration purposes. The following is the <mp>Product</mp> class which has been annotated.

```
@Entity
@Table(name="products")
public class Product {

	@Id
	  @GeneratedValue(strategy=GenerationType.AUTO)
	private long id;

	  @Column(name="productname")
	private String name;
```

```
        @Temporal(TemporalType.TIMESTAMP)
    private Date createDate;
    private float price;
    private int quantity;
    private boolean available;
    private String imgPath;

    ...
}
```

The above code demonstrates how the <mp>Product</mp> class was annotated to allow persistence. Taking the annotations one at a time we will explore what they do.

@Entity

An entity can be considered as a lightweight persistence domain object which represents a table in a relational database. In this case we let Hibernate know that the class Product is an entity. This is required if you would like to introduce database persistence to classes.

@Table(name="products")

The table entity is set at the class level which enables us to define the table, catalog and schema names of the entity mappings. If names are not specified the default values will be used. However, in our case Product is an SQL reserved word therefore we have to change the table name to products.

@Id

We need to let Hibernate know which field of ours is the ID and this can be easily achieved by applying the annotation @Id.

@GeneratedValue(strategy=GenerationType.AUTO)

Along with the annotation, @Id, we also need to inform Hibernate that it is auto-incremented by the database. We let Hibernate know by using the @GeneratedValue annotation and setting the strategy to GenerationType AUTO.

@Column(name="productname")

The next annotation refers to the definition of a column. Under normal circumstances we do not need to provide a name however, in this case without specifying the name of the column it would default to "name." Just as "product", "name" is also an SQL reserved keyword hence we specify a valid column name, "productname".

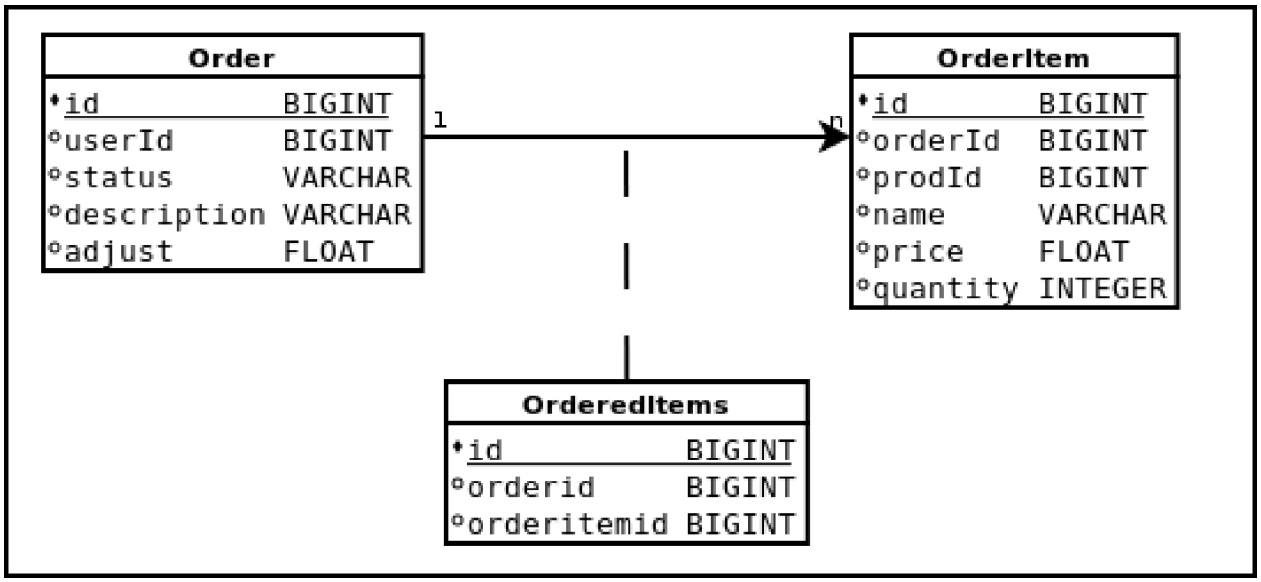
@Temporal(TemporalType.TIMESTAMP)

Java APIs do not define the temporal precision of time. Sometimes we would like to describe the expected precision when placing data into persistent storage. To do this we use the Temporal attribute. Types include DATE, TIME and TIMESTAMP.

The rest of the classes in our example use the same annotation strategies. There is only one exception present in **Order.java** which describes the relationship between an <mp>Order</mp> and its <mp>OrderItems</mp>.

Creating the Relationship - Linking Table

An <mp>Order</mp> has a one-to-many relationships with <mp>OrderItems</mp>. To map this relationship we are required to introduce a linking table named <mp>OrderedItems</mp>. This relationship is shown below:



In order for Hibernate to generate the database schema and map the relationship correctly we need to describe it. To do this we use two annotations:

- @OneToMany
- @JoinTable

Let's explore the relative code.

```
@OneToMany
@JoinTable(
    name="OrderedItems",
    joinColumns=@JoinColumn(name="orderid"),
    inverseJoinColumns=@JoinColumn(name="orderitemid")
)
@LazyCollection(value=LazyCollectionOption.FALSE)
private List<OrderItem> items = new ArrayList<OrderItem>();
```

The "name" attribute tells Hibernate the table name which we would like to use or in our case to create.

The "joinColumns" attribute requires the foreign key columns of the join table which reference the primary table of the entity owning the association, in this case the order id. On the other hand the "inverseJoinColumns" attribute requires the foreign key columns of the join table which reference the primary table of the entity that does not own the association, in this case the OrderItem's id.

We provide both attributes a "@joinColumn". Annotations require **"name"** and **"referencedColumnName"** which are the name of the column in the table OrderedItems and the name of the referenced column within the entity respectively.

The <mp>Order</mp> owns the association (ie. Owns the Order items) so its key is passed to "joinColumns" and <mp>OrderItem</mp> does not own the association so its information is passed to "inverseJoinColumns."

Lazy Loading

There is one last thing to consider; when an <mp>Order</mp> object is loaded we want to load its <mp>OrderItems</mp> at the same time. Hibernate only loads the first object level which can lead to exceptions such as a LazyLoadException when we access the item list which has not been initialized. It means that when the Order object is loaded then the its OrderItems aren't.

In this case it is easy to fix by making use of the @LazyCollection annotation and setting the option to FALSE:

```
@LazyCollection(value=LazyCollectionOption.FALSE)
```

This will ensure that when an <mp>Order</mp> is loaded its <mp>OrderItems</mp> are too. When designing larger systems we have to take into account how much data should be loaded and whether we should follow lazy loading practices or not. For the purposes of this chapter lazy loading is not required.

The final section introduces Hibernate Session management and how we implemented the DAOs.

Hibernate Session Management and Implementing the DAOs

As discussed earlier, we do not use neither an EJB3 container nor Spring. Therefore, we need to manage the session ourselves. This involves building a SessionFactory object when the application loads, closing it when the application ends and manually opening and closing sessions when required.

The SessionFactory and the HibernateUtil

A common pattern in Hibernate is to make use of a <mp>HibernateUtil</mp>. This is a class which provides functionality to build and access our <mp>SessionFactory</mp>. The following code compromises our <mp>HibernateUtil</mp>:

```
public class StoreHibernateUtil {
    private static final SessionFactory sessionFactory;

    static {
        try {

            // Create the SessionFactory from standard
(hibernate.cfg.xml)
            // config file.
            sessionFactory = new
Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Log the exception.
            System.err.println("Initial SessionFactory creation
failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
```

```
        return sessionFactory;
    }

    public static Session openSession() {
        return sessionFactory.openSession();
    }
}
```

In this class, the SessionFactory is built statically upon first accessing the object and then provides static utility methods to access the SessionFactory and then open a new Session.

Please note that there is no need to build the SessionFactory when the application loads or close it when the application terminates. To do so, we implement two ZK interfaces, WebAppInit and WebAppCleanup and specify the implementing class in the zk.xml file:

```
public class HibernateListeners implements WebAppInit, WebAppCleanup {

    public void init(WebApp webapp) throws Exception {
        //initialize Hibernate
        StoreHibernateUtil.getSessionFactory();
    }

    public void cleanup(WebApp webapp) throws Exception {
        //Close Hibernate
        StoreHibernateUtil.getSessionFactory().close();
    }

}
```

```
<zk>
    <listener>
        <listener-class>demo.model.HibernateListeners</listener-class>
   </listener>
</zk>
```

In this case, we named our class <mp>HibernateListeners</mp> and made sure that the <mp>SessionFactory</mp> was built on initialization of the web application and closed in the cleanup method.

This initializes and closes Hibernate now and we need to make use of it in the DAOs.

Re-implementing the DAOs using Hibernate

In this section we are going to explore the general usage of Hibernate in our DAOs for the majority of use cases. We open and close one when performing any database interactions. Hibernate is centered around Session and Transaction concept where a session can have many transactions, commiting and rolling them back from the session as needed. In this example, there will only be one Transaction per session due to the simplicity of the use case. However, it is entirely possible to have a session which span all the transactions of a user. For more information on Sessions and Transactions in Hibernate please click here [1].

Let's take a simple use case for retrieving all the available <mp>Products</mp> found in <mp>ProductDAO.java</mp>:

```
public List<Product> findAll(){
            //return new ArrayList<Product>(dbModel.values());
            Session session = StoreHibernateUtil.openSession();
            Query query = session.createQuery("from products");
            List<Product> products = query.list();

            session.close();
            return products;
}
```

In this code snippet, we utilize our <mp>HibernateUtil</mp> to open a new session at line 3 and then create a query. The query is written in HQL (Hibernate Query Language) retrieving all the rows from the table products. For more information on HQL please click here [1]. Once we create the Query object, we can use its listed method to retrieve the <mp>List</mp> of products, close the session and then return.

At the same time sometimes we need to retrieve all available <mp>Products</mp>. This means we need to retrieve all the Products where available is equal to true. In our example, we do this by creating a Criteria object:

```
public List<Product> findAllAvailable(){
            Session session = StoreHibernateUtil.openSession();
            Transaction t = session.beginTransaction();

            Criteria criteria =
session.createCriteria(Product.class).add(Restrictions.eq("available",
true));
            List<Product> products = criteria.list();
            t.commit();
            session.close();

            return products;
}
```

The snippet follows the same session management strategy, however, this time we create a <mp>Criteria</mp> object to add a restriction that "available" must be true. Then we can use the method <mp>criteria.list()</mp> to retrieve a list of products which are available.

The retrieval methods in each DAO follow a similar strategy so finally let's take a look at the add method of the <mp>OrderDAO</mp> which persists an <mp>Order</mp>.

```
public void add(Order order){

            Session session = StoreHibernateUtil.openSession();
            Transaction t = session.beginTransaction();

            session.persist(order);
            t.commit();

            session.close();
}
```

We see that the Session management strategy is exactly the same, however, now we use the session's persist function to add the <mp>Order</mp> object to the session, then we commit the transaction and close the session. This will then add the Order object to the relative table.

References

[1] http://community.jboss.org/wiki/sessionsandtransactions

Hibernate Summary

As demonstrated by this tutorial Hibernate is highly configurable and very powerful. This chapter covered the very basic annotations and usage of Hibernate, therefore when employing the power of Hibernate for a larger application we should further explore Hibernate's options such as not allowing null values for certain table columns. For a complete list of Hibernate's annotations please click here [1].

References

[1] http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html_single/

Summary and Further Readings

We glimpsed into the fundamental principles behind how ZK works and we employed some of the essential features of ZK to build a sample shopping cart applications in this book. Let's recap on these elements in ZK:

Components

- Components are UI elements that an application developer put them together to build the application UI, just like LEGO bricks. Consult ZK_Component_Reference guide on what children components a particular component can have.
- Components can be declared in ZUML (ZK User Interface Markup Language, in XML syntax), or alternatively, in Java.
- Components (POJO) run on JVM on the server, and they have their counter parts (widgets - JS objects) on the client side.
- User activities on widgets are reflected to the components on the server. Component updates are reflected back to the widgets on the client side.
- A Page [8] is a "stage" where components come on and off to play their parts and fulfill their roles.
- Components are conceptually grouped in ID Spaces; we can call the getFellow() [7] method on one component to get another component as long as they are in the same ID Space.
- By default, components declared in Window [1] form an ID Space (including Window [1] itself) with Window [1] being the ID Space Owner.
- By default, a Page [8] is an ID Space Owner; any component can become the ID Space Owner by implementing IdSpace [9]

Events

- User activities (eg. onClick) and system notifications (eg. server push) are abstracted to event objects.
- Event listeners on components must be registered so that event objects can be created and forwarded.
- By default, all events are sent to the server for processing.
- With ZK's Server+Client Fusion architecture, events could also be handled at the client directly.
- Event handling code can be declared as a component attribute in ZUL.
- Events can be forwarded to a controller class for handling.
- Event listeners can be added dynamically in Java.

ZK MVC

- The ZK MVC approach provides a great separation among data (aka. model), UI and business logic (aka. control).
- Under the ZK MVC pattern, UI components are declared with ZUML in ZUL files. Tasks such as setting a component's data model and renderer, component manipulation are implemented in a Controller class in Java
- The controller class should extend SelectorComposer [1] so the components declared in ZUL can be referenced in Java code. Events are automatically forwarded to the controller class for handling.

Data Display in UI: View-Model-Renderer

- The View-Model-Renderer approach facilitates the display of a data collection in a component such as a Grid [6] or Listbox [7].
- The View-Model-Renderer approach provides a great separation between data and UI.
- The Model is a wrapper for a Java Collection which serves as an interface between the ZK framework and the data collection. For example, ListModelList [2] takes a list as its argument, allowing the framework to access and manipulate data objects within the collection.
- The Renderer provides a set of instructions for associating a bean and bean's properties to UI components.

Annotated Data Binding

- Annotated Data Binding provides a great separation between data and UI.
- Annotated Data Binding allows bean properties to be declared as component attribute values so that View and Model are bound automatically without the need to specify a Renderer or creating a Model wrapper explicitly.
- Under the hood, data binder calls a bean's getter and setter methods to retrieve, or set values in UI components.
- Data binding works with ZK MVVM such as that:
    - ViewModel class provides a getter method to obtain the bean, or Collection.
    - Declare the bean, or Collection, as the value for a UI component in ZUL. The data binder will call the Controller class' getter method to append data to UI.
- By implementing the TypeConverter [2] interface, data type can be converted between UI component and data model when data binding occurs.

Gaining a Deeper Understanding

To strengthen the concepts we learned in this book, please refer to the following links:

- ZK Developer's Reference/UI Composing
- ZK Developer's Reference/Event Handling
- ZK Developer's Reference/Databinding
- ZK Developer's Reference/MVC
- Small Talk: ZK MVC Made Easy

Other ZK Features that Make Things Easier and More Fun

The materials in this book touched bases on the essentials of ZK application development. Other important concepts are crucial to really leveraging the benefits that ZK which brings to Ajax application development. This includes:

- ZK Client-side Reference/General Control/Event Listening
- ZK Developer's Reference/UI Composing/Macro Component

Performance Tips

Please follow these guidelines for building performance critical enterprise applications:

- ZK Developer's Reference/Performance Tips

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zki/ui/util/SelectorComposer.html#

[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/databind/TypeConverter.html#

Article Sources and Contributors

ZK Essentials *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials *Contributors*: Sphota, Tmillsclare

Introduction *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Introduction *Contributors*: Alicelin, Sphota, Tmillsclare

An Introduction to ZK's Server client Fusion Architecture *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Introduction_to_ZK/An_Introduction_to_ZK%27s_Server_client_Fusion_Architecture *Contributors*: Alicelin, PJ li, Sphota, Tmillsclare, Tomyeh

Component Based UI *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Introduction_to_ZK/Component_Based_UI *Contributors*: Alicelin, PJ li, Sphota, Tmillsclare, Tomyeh

Event Driven Programming *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Introduction_to_ZK/Event_Driven_Programming *Contributors*: Alicelin, PJ li, Sphota

Working with the Sample Applications *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Working_with_the_Sample_Applications *Contributors*: Sphota

The Resources *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Working_with_the_Sample_Applications/The_Resources *Contributors*: Alicelin, Sphota, Tmillsclare

Setting Up the Applications Using Eclipse *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Working_with_the_Sample_Applications/Setting_Up_the_Applications_Using_Eclipse *Contributors*: Peterkuo, Sphota, Tomyeh

Store *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Working_with_the_Sample_Applications/Setting_Up_the_Applications_Using_Eclipse/Store *Contributors*: Alicelin, PJ li, Sphota, Tmillsclare

Store with a Database *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Working_with_the_Sample_Applications/Setting_Up_the_Applications_Using_Eclipse/Store_with_a_Database *Contributors*: Alicelin, PJ li, Sphota, Tmillsclare

Laying out Your ZK Components *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Laying_out_Your_ZK_Components *Contributors*: Alicelin, Sphota, Tmillsclare

How to Layout Components *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Laying_out_Your_ZK_Components/How_to_Layout_Components *Contributors*: Alicelin, PJ li, Sphota, Tomyeh

Using ZK Borderlayout *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Laying_out_Your_ZK_Components/Using_ZK_Borderlayout *Contributors*: Alicelin, PJ li, Sphota, Tmillsclare, Tomyeh

Handling the Login Process using ZK MVC and Sessions *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Handling_the_Login_Process_using_ZK_MVC_and_Sessions *Contributors*: Alicelin, PJ li, Sphota, Tmillsclare

The Basic Login *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Handling_the_Login_Process_using_ZK_MVC_and_Sessions/The_Basic_Login *Contributors*: Alicelin, PJ li, Sphota, Tmillsclare

Implementing ZK MVC *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Handling_the_Login_Process_using_ZK_MVC_and_Sessions/Implementing_ZK_MVC *Contributors*: Alicelin, Ashishd, Jeanher, Jimmyshiau, PJ li, Sphota, Tmillsclare, Vincent

Managing credentials using ZK Sessions *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Handling_the_Login_Process_using_ZK_MVC_and_Sessions/Managing_credentials_using_ZK_Sessions *Contributors*: Alicelin, PJ li, Sphota, Tmillsclare, Vincent

Displaying Information Using Grid, MVC and composite components *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Displaying_Information_Using_Grid%2C_MVC_and_composite_components *Contributors*: Tmillsclare

Displaying the Data *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Displaying_Information_Using_Grid%2C_MVC_and_composite_components/Displaying_the_Data *Contributors*: Sphota, Tmillsclare

Composite Components in the MVC *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Displaying_Information_Using_Grid%2C_MVC_and_composite_components/Composite_Components_in_the_MVC *Contributors*: Sphota, Tmillsclare

Displaying information using the Listbox and MVVM *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Displaying_information_using_the_Listbox_and_MVVM *Contributors*: Tmillsclare

What is the MVVM *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Displaying_information_using_the_Listbox_and_MVVM/What_is_the_MVVM *Contributors*: Sphota, Tmillsclare

Displaying Information from the View Model *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Displaying_information_using_the_Listbox_and_MVVM/Displaying_Information_from_the_View_Model *Contributors*: Sphota, Tmillsclare

Performing Actions on Events *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Displaying_information_using_the_Listbox_and_MVVM/Performing_Actions_on_Events *Contributors*: Sphota, Tmillsclare

Communicating between MVC and MVVM patterns and between View Models *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Communicating_between_MVC_and_MVVM_patterns_and_between_View_Models *Contributors*: Tmillsclare

Working with ZK and databases *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Working_with_ZK_and_databases *Contributors*: Alicelin, Tmillsclare

The Theory of Using ZK with Databases *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Working_with_ZK_and_databases/The_Theory_of_Using_ZK_with_Databases *Contributors*: Alicelin, Tmillsclare

Using Hibernate with ZK *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Working_with_ZK_and_databases/Using_Hibernate_with_ZK *Contributors*: Alicelin, Tmillsclare

Hibernate, Entities and Database Generation *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Working_with_ZK_and_databases/Hibernate%2C_Entities_and_Database_Generation *Contributors*: Alicelin, PJ li, Tmillsclare

Hibernate Session Management and Implementing the DAOs *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Working_with_ZK_and_databases/Hibernate_Session_Management_and_Implementing_the_DAOs *Contributors*: Alicelin, PJ li, Tmillsclare

Hibernate Summary *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Working_with_ZK_and_databases/Hibernate_Summary *Contributors*: Tmillsclare

Summary and Further Readings *Source*: http://new.zkoss.org/index.php?title=ZK_Essentials/Summary_and_Further_Readings *Contributors*: Alicelin, Gap77, PJ li, Sphota, Tmillsclare, Tomyeh

Image Sources, Licenses and Contributors

Image:architecture-s.png *Source*: http://new.zkoss.org/index.php?title=File:Architecture-s.png *License*: unknown *Contributors*: Tomyeh

Image:ZKEssentials_Intro_Hello.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_Hello.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Intro_AjaxZUL.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_AjaxZUL.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Intro_ZULSample.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_ZULSample.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Intro_ZULtoPOJO.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_ZULtoPOJO.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Intro_MultiPage.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_MultiPage.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Intro_HelloGoodbye.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_HelloGoodbye.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Intro_IDSpace.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_IDSpace.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Intro_Goodbye.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_Goodbye.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Intro_EventReq.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_EventReq.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Intro_EventRes.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_EventRes.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Intro_EventInRes.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_EventInRes.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Intro_onClick.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_onClick.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Intro_HelloBtn.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_HelloBtn.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Intro_GoodbyeBtn.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_GoodbyeBtn.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Intro_WorldBtnEvent.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_WorldBtnEvent.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Intro_HelloDetached.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_HelloDetached.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Setup_ZKStudio.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Setup_ZKStudio.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Setup_ActivateZKStudio.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Setup_ActivateZKStudio.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Setup_ZKJars.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Setup_ZKJars.png *License*: unknown *Contributors*: Sphota, Tmillsclare

File:ZKEssentials_Existing_Maven_Projects.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Existing_Maven_Projects.png *License*: unknown *Contributors*: Tmillsclare

File:ZKEssentials_Click_Browse.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Click_Browse.png *License*: unknown *Contributors*: Tmillsclare

File:ZKEssentials_check_the_POM.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_check_the_POM.png *License*: unknown *Contributors*: Tmillsclare

Image:ZKEssentials_Layout_Wireframe.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Layout_Wireframe.png *License*: unknown *Contributors*: Sphota

Image: ZKEssentials_Layout_BorderFrames.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Layout_BorderFrames.png *License*: unknown *Contributors*: Alicelin, Matthewcheng, Sphota

Image:ZKEssentials_Layout_Borders.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Layout_Borders.png *License*: unknown *Contributors*: Sphota

Image:ZKEssentials_Layout_FinalizedLayout.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Layout_FinalizedLayout.png *License*: unknown *Contributors*: Sphota

File:ZKEss_StoreLogin.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEss_StoreLogin.png *License*: unknown *Contributors*: Sphota

File:ZKEss_LoginUsecase.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEss_LoginUsecase.png *License*: unknown *Contributors*: Tmillsclare

Image:ZKEssentials_DisplayInGrid_ProductView.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_DisplayInGrid_ProductView.png *License*: unknown *Contributors*: Sphota

Image:ZKEss_TemplateGeneratesRows.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEss_TemplateGeneratesRows.png *License*: unknown *Contributors*: Sphota

Image:ZKEss_ProductOrder.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEss_ProductOrder.png *License*: unknown *Contributors*: Sphota

File:Mvvm-architecture.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-architecture.png *License*: unknown *Contributors*: Hawk

File:SmallTalk_MVVM_HELLO_FLOW.png *Source*: http://new.zkoss.org/index.php?title=File:SmallTalk_MVVM_HELLO_FLOW.png *License*: unknown *Contributors*: Henrichen

Image:ZKEss_ShoppingCart.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEss_ShoppingCart.png *License*: unknown *Contributors*: Sphota

Image:ZKEss_ShoppingCartSubmit.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEss_ShoppingCartSubmit.png *License*: unknown *Contributors*: Sphota

File:ZKEss_linking_table.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEss_linking_table.png *License*: unknown *Contributors*: Tmillsclare