ZK Developer's Reference

For ZK 6.0.0

Contents

Articles

References

ZK Developer's Reference

If you are new to ZK, you might want to take a look at the Tutorial and ZK Essentials sections first.

Documentation:Books/ZK_Developer's_Reference

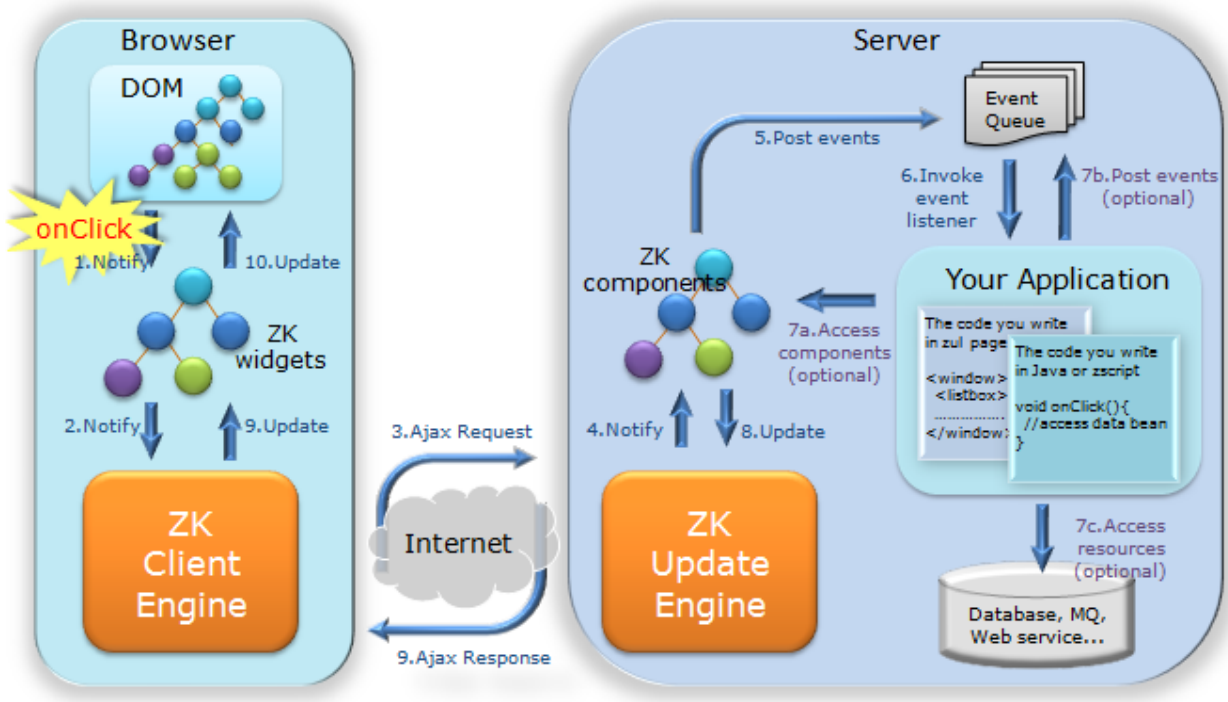
If you have any feedback regarding this book, please leave it here.

<comment>http://books.zkoss.org/wiki/ZK_Developer's_Reference</comment>

Overture

The ZK Developer's Reference is a reference of general features and advanced topics. If you are new to ZK, you might want to start with the Tutorial and ZK Essentials sections first. For information on individual components, please refer to ZK Component Reference. For information on ZUML, please refer to the ZUML Reference.

Architecture Overview



From the Application Developer's Perspective

The ZK application runs on the server. It can access to backend resources, assemble UI with components, listen to user's activity, and then manipulate components to update UI. All of the above activities can be accomplished on 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 on the server, the application can access to full Java technology stacks. User activities, such as Ajax and Server Push, are abstracted to event objects. UI are composed by POJO-like components. It 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 can enhance the interactivity by adding optional client-side functionality, such as client-side event handling, visual effect customizing or even UI composing without server-side coding. ZK enables seamless fusions ranging from pure server-centric to pure client-centric. You can have the best of two worlds: productivity and flexibility.

From the Component Developer's Perspective

Each UI object in ZK consists of a component and a widget. A component is a Java object running on the server representing a UI object which can be manipulated by a Java application. A component has all the behavior of a UI object except that it does not have a visual part. A widget is a JavaScript object[1] running at the client. This object represents the UI object which interacts with the user. Therefore, a widget usually has a visual appearance and it handles events happening at the client.

The relationship between a component and a widget is one-to-one. However, if a component is not attached to a page, there will not be a corresponding widget at the client. On the other hand, the application is allowed to instantiate widgets at the client directly without a corresponding component.

How state synchronization and load distribution might occur depends really on the component. The ZK Client Engine and the Update Engine will work together to provide an elegant and robust channel to simplify the implementation.

For example, assuming that we want to implement a component that allows a user to click on a DOM element to show some detailed information of a component and there are at least two approaches to implement it. Firstly, we could load the detailed information to the client when the widget is instantiated, and then show the details with pure client code. Alternatively, we may choose not to load the detailed information at the very beginning before sending a request back to the server for fulfilling the details when the user clicks on it.

Obviously, the first approach consumes more bandwidth at the initial request but at the same time it also provides faster responses when users click on the DOM element. This is generally more transparent to the application developers, and the implementation of a component can be enhanced later as the project progresses.

[1] It depends on the client. For Ajax-enabled browsers, it is a JavaScript object. For ZK Reach for Android (http://code.google.com/p/zkreach/), it is a Java object running on an Android device.

Execution Flow of Loading a Page

1. When a user types a URL or clicks a hyperlink in the browser, a request is sent to the Web server. If the requested URL matches with the ZK's configured URL pattern[1] , a ZK loader will be invoked to serve this request.
2. The ZK loader loads a specified page and interprets that page to create proper components accordingly[2] .
3. After interpreting the whole page, the ZK loader will render the result to an HTML page. The HTML page is then sent back to the browser accompanied with the ZK Client Engine[3] .
4. The ZK Client Engine renders the widgets into DOM elements and then inserts the DOM elements into the browser's DOM tree to make them visible to users.
5. After that, the ZK Client Engine will sit at the browser to serve requests made by the users, widgets or the applications. If it goes to another page, the execution flow will start over again. If it is going to send an Ajax request back, another execution flow will start as described in the following section.

[1] For more information, please refer to ZK Configuration Reference

[2] If URL is mapped to a richlet, the richlet will be called to handle all UI composition. For more information, please refer to Richlet.

[3] ZK Client Engine is written in JavaScript. Browsers will cache ZK Client engine, so ZK Client engine is usually sent only once at the first visit.

Execution Flow of Serving an Ajax Request

1. The execution flow starts from a widget or the application. This is usually caused by the user's activity (or the application's requirement) and is done by posting a client-side event (Event (http://www.zkoss.org/javadoc/latest/jsdoc/zk/Event.html#)) to a widget (Widget.fire(zk.Event,int) (http://www.zkoss.org/javadoc/latest/jsdoc/zk/Widget.html#fire(zk.Event,int))).
2. The event is then bubbled up to the widget's parent, parent's parent, and finally the ZK Client Engine[1] . The ZK Client Engine then decides whether and when to send the event back to the server in an Ajax request.
3. If necessarily, the ZK Client Engine will send an Ajax request to the ZK Update Engine on the server[2] .
4. Upon receiving Ajax requests, the ZK Update engine will invoke ComponentCtrl.service(org.zkoss.zk.au.AuRequest,boolean) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/sys/ComponentCtrl.html#service(org.zkoss.zk.au.AuRequest,boolean)) for handling an AU request.
5. How the AU request can be handled is really up to a component. But, the component that handles the request usually updates the states, if necessarily, and then notifies the application by posting events to the current execution (Events.postEvent(org.zkoss.zk.ui.event.Event) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Events.html#postEvent(org.zkoss.zk.ui.event.Event))).
6. If any event is posted, the ZK Update Engine will process them one-by-one by invoking the event listeners.
7. The event listener, provided by an application, may choose either to update the backend resources or the components or to post other events.
8. Finally, the ZK Update Engine collects all updates of components, including states change, attachment and detachment for optimization and then send a collection of commands back to the client.
9. The ZK Client Engine evaluates each of these commands to update the widgets accordingly. Then the widgets will update the browser's DOM tree to make them available to the user.

[1] A widget could choose to stop this bubble-up propagation by use of Event.stop(_global_.Map) (http://www.zkoss.org/javadoc/latest/jsdoc/zk/Event.html#stop(_global_.Map))

[2] . From the server's viewpoint, an Ajax request is another type of HTTP request.

When to Send an Ajax Request

When the ZK Client Engine receives a bubbled-up client-side event (Event (http://www.zkoss.org/javadoc/latest/jsdoc/zk/Event.html#)), it will decide whether and when to send the event back to the server for further processing:

1. If there is a non-deferrable event listener registered on the server, the Ajax request will be sent immediately.
2. If there is a deferrable event listener registered on the server, the request will be queued at the client and it will be sent when another event is triggered and a non-deferrable event listener registered for it.
3. If the widget declares that the event is important (ComponentCtrl.CE_IMPORTANT (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/sys/ComponentCtrl.html#CE_IMPORTANT)), the event will be queued for later transmission too.
4. If none of the above case or the widget has no corresponding component on the server, the event will be dropped.

A non-deferred event listener is an event listener (EventListener (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/EventListener.html#)) that does not implement Deferrable (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Deferrable.html#). In other words, to minimize the traffic from the client, you might want to implement an event listener with Deferrable (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Deferrable.html#) if applicable.

Version History

Version	Date	Content

Technology Guidelines

ZK provides end-to-end solutions from UI design, development, testing to production. Here is the technology guidelines to help developers to make choices along the way.

If you are new to ZK and prefer to have some prior knowledge of ZK first, you could skip this section and come back later when you understand more about ZK.

MVC vs. MVVM vs. ZScript

They serve different purposes and could work together. However, some developers get confused about these technologies.

When to use MVC and/or MVVM

MVC (Model-View-Control; aka., Model-View-Presenter) and MVVM (Model-View-ViewModel; aka., Presentation Model) are both design patterns that isolates the dependency among the domain data, the domain logic and the user interface. They are both supported by ZK, and They are praised for its separation of concerns [1] and thus easy to to collaborate, develop and maintain. For a production system, it is strongly suggested to take either MVC or MVVM approach.

MVC separates the design into into three roles: model, view and controller. The controller is the *middle-man* gluing the view and the model (aka., data).

On the other hand, MVVM has three 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. Unlike MVC, ViewModel introduces additional abstraction, so it can be written without any detailed knowledge of the view. In other words, the change of the view will have much less impact to the ViewModel. However, the extra abstraction requires extra design thinking.

In most cases, MVVM is suggested for better separation of concerns. On the other hand, MVC is good for small user interface, such as implementing a macro or composite component, because it is quite straightforward.

When to use zscript

Zscript allows you to embed Java code in ZUML pages. It speeds up the design cycle, so this can be a good approach for prototyping, POC and testing. Zscript is also good for exploiting ZK features and reporting bugs to ZK. However, like any other interpreters, the performance is not very good as it tends to be error-prone. For this reason, it is *not* suggested to use zscript for production systems.

MVC Extractor

ZK Studio provides a tool called MVC Extractor that can convert zscript into MVC automatically. It simplifies the process of transferring the code from prototyping to production.

Documentation links

MVC:
- ZK Developer's Reference: MVC
- ZK Developer's Reference: MVVM
- ZK Developer's Reference: Performance Tips
- Composer [2] and SelectorComposer [3]

ZSCRIPT:
- ZK Developer's Reference: Scripts in ZUML
- ZK Studio Essentials: MVC Extractor

Data Binding

When to use

Data Binding automates the data-copy plumbing code (CRUD) between UI components and the data source. It is strongly suggested to use Data Binding whenever applicable because it can help boost programmers' productivity and the code is easy to read and maintain.

When not to use

Barely. However, as Data Binding requires more time and effort to learn than EL expressions, EL expressions provides an alternative for people who not familiar with ZK, especially during the UI design phase.

Documentation links

- ZK Developer's Reference: Data Binding

ZUML vs. Richlet vs. JSP

When to use ZUML

ZUML is an XML-based approach to declare UI. It does not require any programming knowledge and it works well with MVC, Data Binding and others. ZUML is strongly suggested for usage unless you have different preferences (such as pure Java and JSP).

However, if most parts of a page are in HTML scripts (such as header and footer) and the UI designer is not familiar with ZUML, you could still use JSP to define the page and then include ZUML page(s) for the part that requires ZK components.

Notice that using ZUML does not prevent you from creating components dynamically in Java. In fact, it is a common practice to use ZUML to layout the theme of a Web application, and then use pure Java to manipulate it dynamically.

When to use Richlet

A richlet is a small Java program that composes a user interface in Java for serving a user's request. You could try to use it if you prefer to compose UI in pure Java (like Swing).

When to use JSP

If you would like to use ZK in legacy JSP pages, you could try one of following approaches:

1. Include <jsp:include> in a ZUML page.
2. Apply ZK JSP Tags [4] to a JSP page directly.

As described above, if most of a page consist pure HTML code and the UI designer is not familiar with ZUML, you could use JSP to design the page and then include it in ZUML pages if necessarily.

Notice that ZUML supports the use of HTML tags well (without JSP). For more information, please refer to the ZK Developer's Reference: HTML Tags.

Documentation links

ZUML:	• ZK Developer's Reference: ZUML • ZK Developer's Reference: HTML Tags
Richlet:	• ZK Developer's Reference: Richlet
JSP:	• ZK Developer's Reference: Use ZK in JSP and ZK JSP Tags

Bookmarks vs. Multiple Pages

A traditional page-based Web framework forces developers to split an application into pages. On the other hand, Ajax (ZK) allows developers to group a set of functionality into a single desktop-like page that enables a more friendly user experience.

Grouping is much better based on the functionality, unless it is a small application. For example, it might not be a good idea to group administration with, let's say, data entry. Here are some guidelines:

- If a set of functionality is a logical unit to use and/or to develop, you might want to group it into a single page.
- If SEO (i.e., able to be indexed by search engine) is important, it is better to split UI into multiple pages (and turn on the crawlable option).

It does not matter whether the UI shares the same template (such as header and footer) or not because it will be easy anyway to create similar multiple pages (by the use of inclusion, templating and composite).

When to use bookmarks (in single page)

After grouping a set of functionality into a single page, users can still click on the BACK and the FORWARD button to switch among the states of the single page and even bookmark on a particular state, as if there are multiple pages. This can be done by using Browser History Management (aka., bookmarks). You might consider this as a technique to simulate multiple pages (for a single page with multiple states).

When to use multiple pages

If a set of functionality is logically independent of one another, you could make them as separated pages. To jump from one page to another, you could use the so-called send-redirect technique.

Documentation links

Bookmarks:
- ZK Developer's Reference: Browser History Management
- ZK Developer's Reference: Browser Information and Control

Multiple Pages:
- ZK Developer's Reference: Forward and Redirect
- ZK Developer's Reference: Include, Templating and Composite for consistent UI across multiple pages.

Native Namespace vs. XHTML Components

ZK provides several ways to use XHTML tags in a ZUML document. Here we will discuss native namespace vs. XHTML components. In a ZUML document, they basically mean the same thing except for the XML namespace. Therefore it should be easy to switch between them.

When to use native namespace

With the use of an XML namespace called the native namespace, you could declare any tags in ZUML as long as they are valid to the client (i.e., any HTML tags for a browser). It is suggested to use this technology if the HTML tags are static. For example, you will not able to change the content dynamically with Ajax. The header, sidebar, footer and the layout elements are typical examples. It saves the memory on the server.

When to use XHTML components

ZK also provides a set of components to represent each XHTML tag on the server. Unlike the native namespace, these are the 'real' ZK components.

It is suggested that you may change their content dynamically because they behave the same as other ZK components. However, since it is a component, it consumes the server's memory.

Documentation links

- ZK Developer's Reference: HTML Tags
- ZK Developer's Reference: Performance Tips|Native vs. XHTML
- ZK Developer's Reference: Performance Tips: Stubonly

Include, Macro, Composite and Templating

They allow developers to modularize the UI such that it becomes easier to develop, maintain and reuse.

When to use include

Include allows you to include a ZUML page, a static page, a JSP page or the result of a servlet. This is the most suitable for usage if you would like to:

1. Include a non-ZUML page
2. Use a page (Page [5]) to encapsulate a ZUML page[6] [7]

The limitation of Include is that you can not encapsulate its behavior in a Java class (like macro or composite components do).

[1] http://en.wikipedia.org/wiki/Separation_of_concerns
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Composer.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/SelectorComposer.html#
[4] http://www.zkoss.org/product/zkjsp.dsp
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Page.html#
[6] You have to specify mode="defer" to create a Page (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Page.html#) instance.
[7] Whether a page is required really depends on developer's preference. Introducing a page is more complicated but logically more loosely-coupled.

When to use macro components

Macro components allow developers to define a new component with a ZUML page. So if you would like to reuse a ZUML page across different pages, you can because

1. Though optional, you could encapsulate the behavior in a Java class
2. It is easier to map a macro component to another URI, if necessary
3. There is no difference between the use of a macro component and other components

When to use composite components

Composite component is another way to define a new component. With this approach, you could extend a new component from any existent components. However, you must implement a Java class to represent the component[1] . Unlike macro components, you have to handle the component creation from a ZUML page by yourself[2] .

Feel free to use composite components if you want to inherit the behavior from an existent component, such as Window (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Window.html#) and Cell (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Cell.html#), and enhance it to have child components defined in a ZUML document.

---

[1] Here is an example of composite components in ZK Demo (http://www.zkoss.org/zkdemo/composite/composite_component)
[2] There is a utility called ZK Composite (https://github.com/zanyking/ZK-Composite). It allows to define a composite component with Java annotations. Please refer to Small Talks: Define Composite Component using Java Annotation in ZK6 for the details.

When to use templating

Templating allows developers to define UI fragments and define how to assemble them into a complete UI at runtime. Its use is very different from other approaches. Feel free to use templating if you would like the overall layout to be decided at runtime based on, let's say, users' roles or preferences.

Performance and Security

For production systems, it is strongly recommended to take a look at the Performance Tips and Security Tips sections first.

JSF

When to use

JSF is a page-based framework. Because it is too complicated to use, we strongly recommend you to deploy ZK. ZK can do whatever JSF can do or even better. However, if you have to use ZK with legacy JSF, please refer to the Embed ZK Component in Foreign Framework section[1] .

[1] Notice that ZK JSF Components is no longer supported.

Version History

Version	Date	Content

Extensions

Here is an overview of the extensions of ZK. They are optional. If you are new to ZK and prefer have some knowledge of ZK first, you could skip this section and come back later after you understand more about ZK.

There are hundreds of projects which extend ZK's functionality by boosting programmer productivity, providing sample code and many others. For more projects, you could search Google Code [1], Sourceforge.net [2], GitHub [3], ZK Forge [4] etc.

IDE and Tools

ZK Studio [5]

ZK Studio is a visual integrated development environment for developing ZK applications with Eclipse IDE [6].

REM [7]

REM is a NetBeans [8] module for ZK. It simplifies the development of ZK with NetBeans IDE [8].

ZTL [9]

ZTL is a testing toll to automate ZK tests including unit testing and screen-image comparison.

ZK CDT [10]

ZK CDT is a component development tool which provides wizards to simplify the creation of ZK components.

ZK Jet

ZK Jet is a browser extension that works with Firefox and Google Chrome. This provides users with a ZK sandbox environment.

run-jetty-run [11]

Use this plugin's embedded Jetty distribution to run web applications in Eclipse. This helps to speed up the ZK development by minimizing the deployment time. The project is maintained by Tony Wang [12], a member of the ZK Team.

Libraries and Integrations

ZK Spring [13]

ZK Spring integrates ZK and Spring framework [14]. It supports Spring, Spring Security [15], and Spring Web Flow [16].

ZK JSP Tags [4]

ZK JSP Tags is a collection of JSP tags built upon ZK components, such as that developers could use ZK components and other JSP tags in the same JSP page.

ZKGrails [17]

ZKGrails is a ZK plugin for the next generation rapid Web development framework, Grails [18].

ZK addon for Spring ROO [19]

ZK addon for Spring ROO enables rapid development of ZK / Spring / JPA projects using Spring ROO [20].

ZK UI Plugin for Grails [21]

The ZK UI plugin, similar to ZKGrails [17], can be seamlessly integrated ZK with Grails [18]. It uses the Grails' infrastructures, such as gsp and controllers.

ZEST [22]

ZEST is a lightweight MVC and REST framework which provides an additional page-level MVC pattern to isolate the request's URI, controller and view (such as ZUML document).

ZK Web Flow [23]

ZK Web Flow provides Ajax-based web flow mechanisms for rich applications.

ZK CDI [24]

ZK CDI is integrated with ZK and JBoss Weld CDI RI [25].

ZK Seam [26]

ZK Seam is integrated with ZK and Seam [27].

ZK Mobile [28]

ZK Mobile runs as a native application on mobile devices that support, let's say, Java Mobile and Google Android. It does not require a modern Web browser.

ZK JSF Components [29]

ZK JSF Components are a collection of JSF Components built upon highly interactive ZK Ajax components.

Components and Themes

ZK Themes [30]

ZK Themes is a collection of various themes, including breeze, silvertail and sapphire.

ZK Spreadsheet [31]

ZK Spreadsheet is a ZK component delivering functionalities found in Microsoft Excel to ZK applications.

ZK Pivottable [32]

ZK Pivottable is a ZK component for data summarization that sorts and sums up the original data layout.

ZK Calendar [33]

ZK Calendar is a ZK component enabling rich and intuitive scheduling functionality to ZK applications.

Canvas4Z [34]

Canvas4Z is an experimental component that leverages the power of HTML5 Canvas to draw arbitrary shapes and objects in the browsers.

ZUSS

ZUSS (ZK User-interface Style Sheet) is an extension to CSS. It is compatible with CSS, while allows the dynamic content, such as variables, mixins, nested rules, expressions, and Java methods with existing CSS syntax.

ZK Incubator Widgets [35]

ZK Incubator Widgets [35] hosts a collection of incubator widgets, tools and add-ons.

References

[1] http://code.google.com/query/#q=zk
[2] http://sourceforge.net/search/?q=zk
[3] http://github.com/search?langOverride=&q=zk&repo=&start_value=1&type=Repositories
[4] http://sourceforge.net/projects/zkforge/
[5] http://www.zkoss.org/product/zkstudio.dsp
[6] http://www.eclipse.org
[7] http://rem1.sourceforge.net/
[8] http://www.netbeans.org/
[9] http://code.google.com/p/zk-ztl/
[10] http://code.google.com/a/eclipselabs.org/p/zk-cdt/
[11] http://code.google.com/p/run-jetty-run/
[12] https://github.com/tony1223
[13] http://www.zkoss.org/product/zkspring.dsp
[14] http://www.springsource.org/
[15] http://static.springsource.org/spring-security/site/
[16] http://www.springsource.org/webflow
[17] http://code.google.com/p/zkgrails/
[18] http://www.grails.org
[19] http://code.google.com/p/zk-roo/
[20] http://www.springsource.org/spring-roo
[21] http://www.grails.org/plugin/zkui
[22] http://code.google.com/p/zest/
[23] http://code.google.com/p/zkwebflow/
[24] http://code.google.com/p/zkcdi/
[25] http://seamframework.org/Weld
[26] http://code.google.com/p/zkseam2/
[27] http://seamframework.org/
[28] http://www.zkoss.org/product/zkmobile
[29] http://www.zkoss.org/product/zkjsf
[30] http://code.google.com/p/zkthemes/
[31] http://www.zkoss.org/product/zkspreadsheet.dsp
[32] http://www.zkoss.org/product/zkpivottable.dsp
[33] http://www.zkoss.org/product/zkcalendar.dsp
[34] http://code.google.com/p/canvas4z/
[35] http://code.google.com/p/zk-widgets/

UI Composing

Each UI object is represented by a component (Component [1]). In this section we will discuss how to declare UI, including XML-based approach and pure-Java approach.

This section describes more general and overall concepts of UI composing. For more detailed and specific topics, please refer to the UI Patterns section. For detailed information on each individual component, please refer to the ZK Component Reference.

References

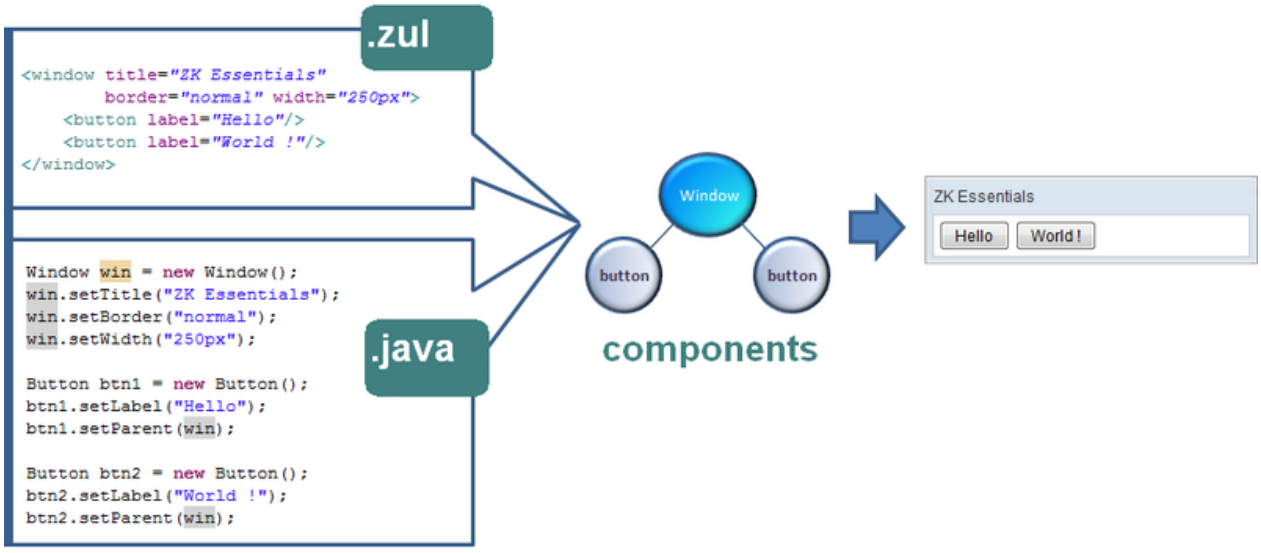
[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#

Component-based UI

Overview

Each UI object is represented by a component (Component [1]). Thus, composing an UI object is like assembling components. To alter UI one has to modify the states and relationships of components.

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



As shown above, there are two ways to declare UI: XML-based approach and pure-Java approach. You can mix them if you like.

Forest of Trees of Components

Like in a tree structure, a component has at most one parent, while it might have multiple children.

Some components accept only certain types of components as children. Some do not allow to have any children at all. For example, Grid [3] in XUL accepts Columns [4] and Rows [5] as children only.

A component without any parents is called a **root component**. Each page is allowed to have multiple root components, even though this does not happen very often.

Notice that if you are using ZUML, there is an XML limitation, which means that only one document root is allowed. To specify multiple roots, you have to enclose the root components with the `zk` tag. `zk` tag a special tag that does not create components. For example,

```
<zk>
    <window/> <!-- the first root component -->
    <div/> <!-- the second root component -->
</zk>
```

getChildren()

Most of the collections returned by a component, such as Component.getChildren() [6], are live structures. It means that you can add, remove or clear a child by manipulating the returned list directly.. For example, to detach all children, you could do it in one statement:

```
comp.getChildren().clear();
```

It is equivalent to

```
for (Iterator it = comp.getChildren().iterator(); it.hasNext();) {
    it.next();
    it.remove();
}
```

Note that the following code will never work because it would cause ConcurrentModificationException.

```
for (Iterator it = comp.getChildren().iterator(); it.hasNext();)
    ((Component)it.next()).detach();
```

Sorting the children

The following statement will fail for sure because the list is live and a component will be detached first before we move it to different location.

```
Collections.sort(comp.getChildren());
```

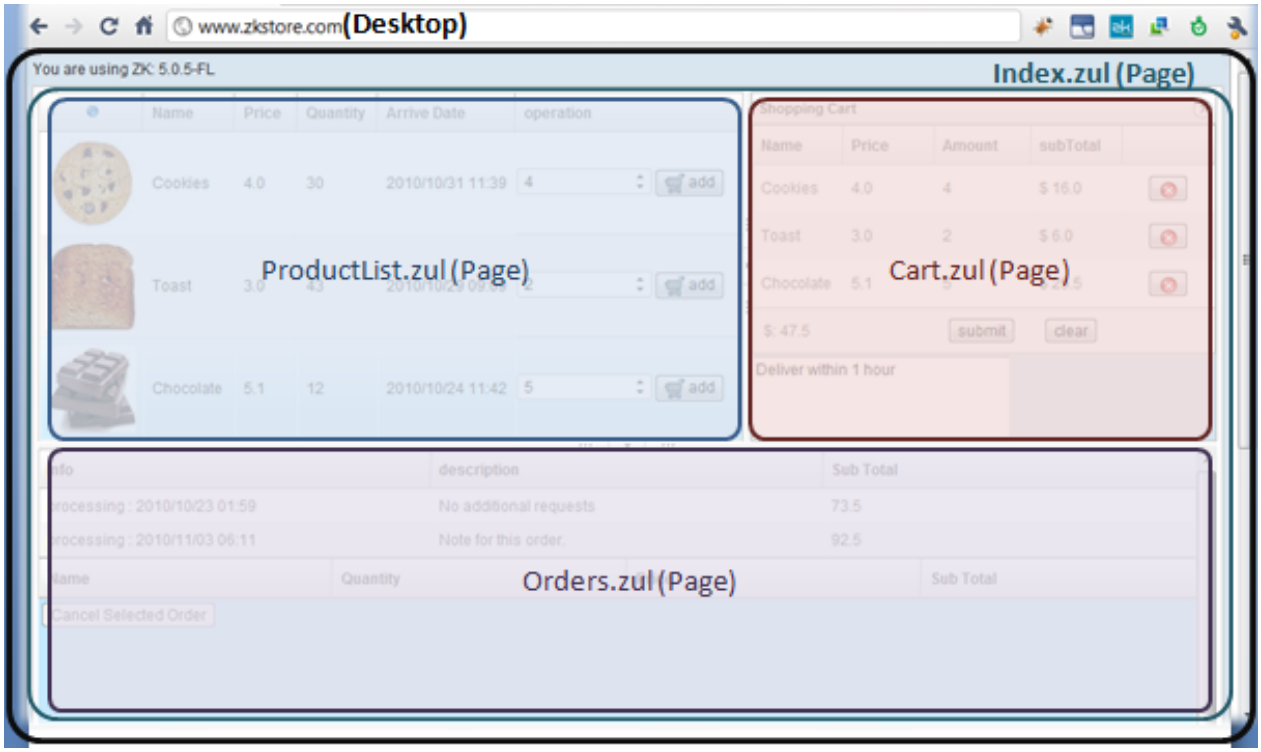
More precisely, a component has at most one parent and it has only one spot in the list of children. It means, the list is actually a set (no duplicate elements allowed). On the other hand, Collections.sort() cannot handle a set correctly.

Thus, we have to copy the list to another list or array and then sort it. java.util.Comparator) Components.sort(java.util.List, java.util.Comparator) [7] is a utility to simplify the job.

Desktop, Page and Component

A page (Page [8]) is a collection of components. It represents a portion of the browser window. Only components attached to a page are available at the client. They are removed when they are detached from a page.

A desktop (Desktop [9]) is a collection of pages. It represents a browser window (a tab or a frame of the browser)[10] . You might image a desktop representing an independent HTTP request.



A desktop is also a logic scope that an application can access in a request. Each time a request is sent from the client, it is associated with the desktop it belongs. The request is passed to boolean) DesktopCtrl.service(org.zkoss.zk.au.AuRequest, boolean) [11] and then forwarded to boolean) ComponentCtrl.service(org.zkoss.zk.au.AuRequest, boolean) [12]. This also means that the application can not access components in multiple desktops at the same time.

Both a desktop and a page can be created automatically when ZK Loader loads a ZUML page or calls a richlet (Richlet.service(org.zkoss.zk.ui.Page) [13]). The second page is created when the Include [14] component includes another page with the defer mode. For example, two pages will be created if the following is visited:

```
<!-- the main page -->
<window>
  <include src="another.zul" mode="defer"/> <!-- creates another page -->
</window>
```

Notice that if the mode is not specified (i.e., the instant mode), Include [14] will not be able to create a new page. Rather, it will append all components created by another.zul as its own child components. For example,

```
<window>
  <include src="another.zul"/> <!-- default: instant mode -->
</window>
```

is equivalent to the following (except div is not a space owner, see below)

```
<window>
  <div>
```

```
    <zscript>
      execution.createComponents("another.zul", self, null);
    </zscript>
  </div>
</window>
```

[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/Grid.html#
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Columns.html#
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Rows.html#
[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#getChildren()
[7] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Components.html#sort(java.util.List,
[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/Desktop.html#
[10] Under portal environment, there might be multiple desktops in one browser window. However, it is really important in the developer's viewpoint.
[11] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/sys/DesktopCtrl.html#service(org.zkoss.zk.au.AuRequest,
[12] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/sys/ComponentCtrl.html#service(org.zkoss.zk.au.AuRequest,
[13] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Richlet.html#service(org.zkoss.zk.ui.Page)
[14] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Include.html#

Attach a Component to a Page

A component is available at the client only if it is attached to a page. For example, the window created below will not be available at the client.

```
Window win = new Window();
win.appendChild(new Label("foo"));
```

A component is a POJO object. If you do not have any reference to it, it will be recycled when JVM starts garbage collection (http://en.wikipedia.org/wiki/Garbage_collection_(computer_science)).

There are two ways to attach a component to a page:

1. Append it as a child of another component that is already attached to a page (Component.appendChild(org.zkoss.zk.ui.Component) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#appendChild(org.zkoss.zk.ui.Component)), org.zkoss.zk.ui.Component) Component.insertBefore(org.zkoss.zk.ui.Component, org.zkoss.zk.ui.Component) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#insertBefore(org.zkoss.zk.ui.Component,), or Component.setParent(org.zkoss.zk.ui.Component) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#setParent(org.zkoss.zk.ui.Component))).
2. Invoke Component.setPage(org.zkoss.zk.ui.Page) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#setPage(org.zkoss.zk.ui.Page)) to attach it to a page directly. It is also another way to make a component become a root component.

Since a component can have at most one parent and be attached at most one page, it will be detached automatically from the previous parent or page when it is attached to another component or page. For example, b will be a child of win2 and win1 has no child at the end.

```
Window win1 = new Window;
Button b = new Button();
win1.appendChild(b);
win2.appendChild(b); //implies detach b from win1
```

Detach a Component from a Page

To detach a Component from the page, you can either invoke comp.setParent(null) if it is not a root component or comp.setPage(null) if it is a root component. Component.detach() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/zk/ui/Component.html#detach()) is a shortcut to detach a component without knowing if it is a root component.

Invalidate a Component

When a component is attached to a page, the component and all of its descendants will be rendered. On the other hand, when a state of a attached component is changed, only the changed state is sent to client for update (for better performance). Very rare, you might need to invoke Component.invalidate() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#invalidate()) to force the component and its descendants to be rerendered[1] .

There are only a few reasons to invalidate a component, but it is still worthwhile to note them down:

1. If you add more than 20 child components, you could invalidate the parent to improve the performance. Though the result Ajax response might be longer, the browser will be more effective to replace a DOM tree rather than adding DOM elements.
2. If a component has a bug that does not update the DOM tree correctly, you could invalidate its parent to resolve the problem[2] .

---

[1] ZK Update Engine will queue the *update* and *invalidate* commands, and then optimize them before sending them back to the client (for better performance)

[2] Of course, remember to let us know and we will fix it in the upcoming version.

Don't Cache Components Attached to a Page in Static Fields

As described above, a desktop is a logical scope which can be accessed by the application when serving a request. In other words, the application cannot detach a component from one desktop to another desktop. This typically happens when you cache a component accidentally.

For example, the following code will cause an exception if it is loaded multiple times.

```
<window apply="foo.Foo"/> <!-- cause foo.Foo to be instantiated and executed -->
```

and foo.Foo is defined as follows[1] .

```
package foo;
import org.zkoss.zk.ui.*;
import org.zkoss.zul.*;

public class Foo implements org.zkoss.zk.ui.util.Composer {
   private static Window main; //WRONG! don't cache it in a static field
   public void doAfterCompose(Component comp) {
        if (main == null)
            main = new Window();
        comp.appendChild(main);
   }
}
```

The exception is similar to the following:

```
org.zkoss.zk.ui.UiException: The parent and child must be in the same
desktop: <Window u1EP0>
```

```
org.zkoss.zk.ui.AbstractComponent.checkParentChild(AbstractComponent.java:1057)

org.zkoss.zk.ui.AbstractComponent.insertBefore(AbstractComponent.java:1074)
      org.zkoss.zul.Window.insertBefore(Window.java:833)

org.zkoss.zk.ui.AbstractComponent.appendChild(AbstractComponent.java:1232)
      foo.Foo.doAfterCompose(Foo.java:10)
```

[1] A composer (Composer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Composer.html#)) is a controller that can be associated with a component for handling the UI in Java. For the information, please refer to the Composer section.

Component Cloning

All components are cloneable (java.lang.Cloneable). It is simple to replicate components by invoking Component.clone() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#clone()).

```
main.appendChild(listbox.clone());
```

Notice

- It is a *deep clone*. That is, all children and descendants are cloned too.
- The component returned by Component.clone() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#clone()) does not belong to any pages. It doesn't have a parent either. You have to attach it manually if necessary.
- ID, if any, is preserved. Thus, you *cannot* attach the returned component to the same ID space without modifying ID if there is any.

Similarly, all components are serializable (java.io.Serializable). Like cloning, all children and descendants are serialized. If you serialize a component and then de-serialize it back, the result will be the same as invoking Component.clone() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#clone())[1] .

[1] Of course, the performance of Component.clone() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#clone()) is much better.

Version History

Version	Date	Content

ID Space

ID Space

It is common to decompose a visual presentation into several subsets or ZUML pages. For example, you may use a page to display a purchase order, and a modal dialog to enter the payment term. If all components are uniquely identifiable in the same desktop, developers have to maintain the uniqueness of all identifiers for all pages that might create in the same desktop. This step can be tedious, if not impossible, for a sophisticated application.

The concept of ID space is hence introduced to resolve this issue. An ID space is a subset of components of a desktop. The uniqueness is guaranteed only in the scope of an ID space. Thus, developers could maintain the subset of components separately without the need to worry if there is any conflicts with other subsets.

Window (Window [1]) is a typical component that is an ID space. All descendant components of a window (including the window itself) form an independent ID space. Thus, you could use a window as the topmost component to group components. This way developers only need to maintain the uniqueness of each subset separately.

By and large, every component can form an ID space as long as it implements IdSpace [1]. This type of component is called the space owner of the ID space after the component is formed. Components in the same ID space are called "fellows".

When a page implements IdSpace [1], it becomes a space owner. In additions, the macro component and the include component (Include [14]) can also be space owners.

Another example is idspace (Idspace [2]). It derives from div, and is the simplest component implementing IdSpace [1]. If you don't need any feature of window, you could use ispace instead.

You could make a standard component as a space owner by extending it to implement IdSpace [1]. For example,

```
public class IdGrid extends Grid implements IdSpace {
   //no method implementation required
}
```

Tree of ID Space

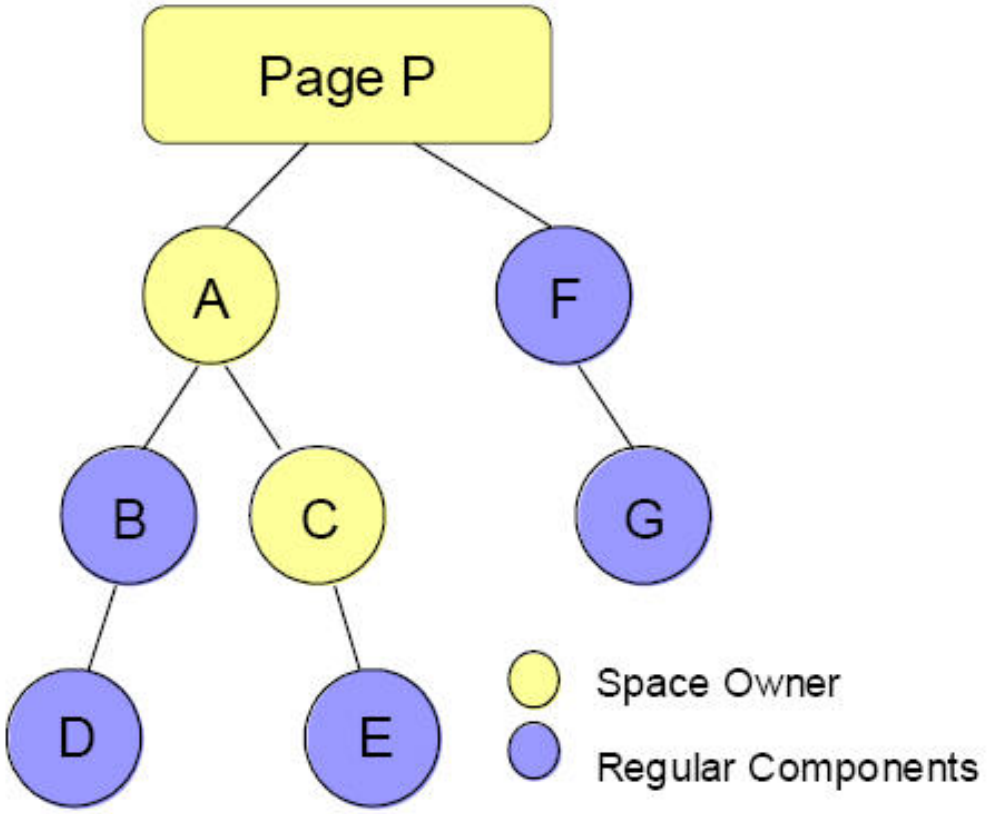
If an ID space has a child ID space, the components of the child space are not part of the parent ID space. But the space owner of the child ID space will be an exception in this case. For example, if an ID space, let's say X, is a descendant of another ID space, let's say Y, then space X's owner is part of space Y. However, the descendants of X is not a part of space Y.

For example, see the following ZUML page

```
<?page id="P"?>
<zk>
      <window id="A">
            <hbox id="B">
                  <button id="D" />
            </hbox>
            <window id="C">
                  <button id="E" />
            </window>
      </window>
      <hbox id="F">
```

```
            <button id="G" />
        </hbox>
</zk>
```

will form ID spaces as follows:



As depicted in the figure, there are three spaces: P, A and C. Space P includes P, A, F and G. Space A includes A, B, C and D. Space C includes C and E.

Components in the same ID spaces are called fellows. For example, A, B, C and D are fellows of the same ID space.

getFellow and getSpaceOwner

The owner of an ID space could be retrieved by Component.getSpaceOwner() [3] and any components in an ID space could be retrieved by Component.getFellow(java.lang.String) [4], if it is assigned with an ID (Component.setId(java.lang.String) [5]).

Notice that the `getFellow` method can be invoked against any components in the same ID space, not just the space owner. Similarly, the `getSpaceOwner` method returns the same object for any components in the same ID space, no matter if it is the space owner or not. In the example above, if C calls `getSpaceOwner` it will get C itself, if C calls `getSpaceOwnerOfParent` it will get A.

Composer and Fellow Auto-wiring

With ZK Developer's Reference/MVC, you generally don't need to look up fellows manually. Rather, they could be *wired* automatically by using the auto-wiring feature of a composer. For example,

```
public class MyComposer extends SelectorComposer {
    @Wire
    private Textbox input; //will be wired automatically if there is a
fellow named input

    public void onOK() {
      Messsagebox.show("You entered " + input.getValue());
    }
    public void onCancel() {
      input.setValue("");
    }
}
```

Then, you could associate this composer to a component by specifying the apply attribute as shown below.

```
<window apply="MyComposer">
    <textbox id="input"/>
</window>
```

Once the ZUML document above is rendered, an instance of MyComposer will be instantiated and the input member will also be initialized with the fellow named input. This process is called "auto-wiring". For more information, please refer to the Wire Components section.

Find Component Manually

There are basically two approaches to look for a component: by use of CSS-like selector and filesystem-like path. The CSS-like selector is more powerful and suggested if you're familiar with CSS selectors, while filesystem-like path is recommended if you're familiar with filesystem's path.

Selector

Component.query(java.lang.String) [6] and Component.queryAll(java.lang.String) [7] are the methods to look for a component by use of CSS selectors. For example,

```
comp.queyr("#ok"); //look for a component whose ID's ok in the same ID
space
comp.query("window #ok"); //look for a window and then look for a
component with ID=ok in the window
comp.queryAll("window button"); //look for a window and then look for
all buttons in the window
```

Component.query(java.lang.String) [6] returns the first matched component, or null if not found. On the other hand, Component.queryAll(java.lang.String) [7] returns a list of all matched components.

Path

ZK provides a utility class called Path [8] to simplify the location of a component among ID spaces. The way of using it is similar to `java.io.File`. For example,

```
//Two different ways to get the same component E
Path.getComponent("/A/C/E");//if call Path.getComponent under the same
page.
new Path("/A/C", "E").getComponent(); //the same as new
Path("/A/C/E").getComponent()
```

Notice that the formal syntax of the path string is "*/[/]<space_owner>/[<space_owner>...]/felow*" and only the last element could fellow because it is not space owner. For example,

```
// B and D are fellows in the Id space of A
Path.getComponent("/A/B");  // get B
Path.getComponent("/A/D");  // get D
```

If a component belongs to another page, we can retrieve it by starting with the page's ID. Notice that double slashes have to be specified in front of the page's ID.

```
Path.getComponent("//P/A/C/E");//for page, you have to use // as prefix
```

Notice that the page's ID can be assigned with the use of the page directive as follows.

```
<?page id="foo"?>
<window/>
```

UUID

A component has another identifier called UUID (Universal Unique ID). It is assigned automatically when the component is attached to a page. UUID of a component is unique in the whole desktop (if it is attached).

Application developers rarely need to access it.

In general, UUID is independent of ID. UUID is assigned automatically by ZK, while ID is assigned by the application. However, if a component implements RawId [9], ID will become UUID if the application assigns one. Currently, only components from the XHTML component set implements RawId [9].

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/IdSpace.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Idspace.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#getSpaceOwner()
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#getFellow(java.lang.String)
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#setId(java.lang.String)
[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#query(java.lang.String)
[7] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#queryAll(java.lang.String)
[8] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Path.html#
[9] http://www.zkoss.org/javadoc/latest/zk/org/zkos/zk/ui/ext/RawId.html#

ZUML

There are two ways to compose UI: XML-based approach and pure-Java approach. Here we will describe XML-based approach. For pure-Java approach, please refer to the next chapter.

The declaration language is called ZK User Interface Markup Language (ZUML). It is based on XML. Each XML element instructs ZK Loader to create a component. Each XML attribute describes what value to be assigned to the created component. Each XML processing instruction describes how to process the whole page, such as the page title. For example,

```
<?page title="Super Application"?>
<window title="Super Hello" border="normal">
    <button label="hi" onClick='alert("hi")'/>
```

where the first line specifies the page title, the second line creates a root component with title and border, and the third line creates a button with label and an event listener.

Auto-completion with Schema

When working with a ZUML document, it is suggested to use ZK Studio [5] since it provides a lot of features to simplify editing, such as *content assist* and *visual editor'*.

If you prefer not to use ZK Studio, you could specify the XML schema in a ZUML document as shown below. Many XML editors works better, such as when with auto-complete, if XML schema is specified correctly.

```
 <window xmlns="http://www.zkoss.org/2005/zul"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.zkoss.org/2005/zul
http://www.zkoss.org/2005/zul/zul.xsd">
```

The ZUL schema can be downloaded from http://www.zkoss.org/2005/zul/zul.xsd [1]. In addition, you can find `zul.xsd` under the `dist/xsd` directory in the ZK binary distribution.

This section is about the general use of ZUML. For a complete reference, please refer to ZUML Reference.

References

[1] http://www.zkoss.org/2005/zul/zul.xsd

XML Background

Overview

This section provides the most basic concepts of XML to work with ZK. If you are familiar with XML, you could skip this section. If you want to learn more, there are a lot of resources on Internet, such as http://www.w3schools.com/xml/xml_whatis.asp [1] and http://www.xml.com/pub/a/98/10/guide0.html [2].

XML is a markup language much like HTML but with stricter and cleaner syntax. It has several characteristics worthwhile to take notes of.

Document

The whole XML content, no matter whether if it is in a file or as a string, is called a XML document.

Character Encoding

It is, though optional, a good idea to specify the encoding in your XML so that the XML parser can interpret it correctly. Note: it must be on the first line of the XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
```

In addition to specifying the correct encoding, you have to make sure your XML editor supports it as well.

Elements

An XML element is everything from (including) the element's start tag to (including) the element's end tag.

An element can contain other elements, let it be simple text or a mixture of both. Elements can also have attributes. For example,

```
<window title="abc">
  <button label="click me"/>
</window>
```

where both window and button are elements, while title is an attribute of the window element. The button element is nested in the window element. We call the window component the parent element of button, while the button component is a child element of the window.

The document root is the topmost element (without any parent element). There is exactly one document root per XML document.

Elements Must Be Well-formed

First, each element must be closed. They are two ways to close an element as depicted below. They are equivalent.

Description	Code
Close by an end tag:	`<window></window>`
Close without an end tag:	`<window/>`

Second, elements must be properly nested.

Result	Code

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

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

XML treats every tag as a node in a tree. A node without a parent node is a root component, and it is the root of a tree. In each zul file, only **ONE** tree is allowed.

For example, for being a whole zul file, the following is allowed, for it must have only one root component.

```
<button/>
```

And for being a whole zul file, the following is not allowed, for it must have more than one root component.

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

You can solve the problem simply by adding a tag to enclose the whole zul file to serve as the parent node, so that the zul file has one single tree again.

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

Special Character Must Be Replaced

XML uses `<element-name>` to denote an element, so you have to use special characters for replacement. For example, you have to use `&lt;` to represent the `<` character.

Special Character	Replaced With	
<	<	
>	>	
&	&	
"	"	
'	'	
\t (TAB)		Required only if use it in a XML attribute's value
\n (Linefeed)
Required only if use it in a XML attribute's value

Alternatively, you could tell XML parser not to interpret a piece of text by using `CDATA`. See the following:

```
<zscript>
<![CDATA[
void myfunc(int a, int b) {
    if (a < 0 && b > 0) {
        //do something
    }
]]>
</zscript>
```

It is suggested to always add `<![CDATA[ ]]>` inside your `<zscript> </zscript>`. Thus you don't have to worry about the escape sequences for special characters like "&", "<". In addition, the code can also become much easier to read and for maintenance.

Attribute Values Must Be Specified and Quoted

Result	Code
Correct:	`width="100%" checked="true"`
Wrong:	`width=100% checked`

Both the single quote (') and the double quote (") can be used, so if the value has double quotes, you could use the single quote to enclose it. For example,

```
<button onClick='alert("Hello, There")'/>
```

Of course, you can always use " to denote a double quote.

Comments

A comment is used to leave a note or to temporarily disable a block of XML code. To add a comment in XML, use <!-- and --> to mark the comment body.

```
<window>
<!-- this is a comment and ignored by ZK -->
</window>
```

Processing Instruction

A processing instruction is used to carry out the instruction to the program that processes the XML document. A processing instruction is enclosed with <? and ?>. For example,

```
<?page title="Foo"?>
```

Processing instructions may occur anywhere in an XML document. However, most ZUML processing instructions must be specified at the topmost level (the same level of the document root).

References

[1] http://www.w3schools.com/xml/xml_whatis.asp
[2] http://www.xml.com/pub/a/98/10/guide0.html

Basic Rules

If you are not familiar with XML, please take a look at XML Background first.

An XML Element Represents a Component

Each XML element represents a component, except for special elements like <zk> and <attribute>. Thus, the following example will cause three components (window, textbox and button) being created when ZK Loader processes it.

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

In addition, the parent-child relationship of the created components will follow the same hierarchical structure of the XML document. In the previous example, window will be the parent of textbox and button, while textbox is the first child and button is the second.

Special XML Elements

There are a few elements dedicated to special functionality rather than a component. For example,

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

The zk element is a special element used to aggregate other components. Unlike a real component (say, `hbox` or `div`), it is not part of the component tree being created. In other words, it does not represent any components. For example,

```
<window>
    <zk if="${whatever}">
        <textbox/>
        <textbox/>
    </zk>
</window>
```

is equivalent to

```
<window>
    <textbox if="${whatever}"/>
    <textbox if="${whatever}"/>
</window>
```

For more information about special elements, please refer to ZUML Reference.

A XML Attribute Assigns a Value to a Component's Property or Event Listener

Each attribute, except for special attributes like if and forEach, represents a value that should be assigned to a property of a component after it is created. The attribute name is the property name, while the attribute value is the value to assign. For example, the following example assigns "Hello" to the window's title property. More precisely, Window.setTitle(java.lang.String) [1] will be called "Hello".

```
<window title="Hello"/>
```

Like JSP, you could use EL for the value of any attributes. The following example assigns the value of the request parameter called name to window's title.

```
<window title="${param.name}"/>
```

For more information about EL expressions, please refer to ZUML Reference.

Assign Event Listener if the Name Starts With on

If the attribute name starts with on and the third letter is uppercase, an event listener is assigned. For example, we can register an event listener to handle the onClick event as follows:

```
<button onClick="do_something_in_Java())"/>
```

The attribute value must be a valid Java code, and it will be interpreted[2] when the event is received. You could specify different languages by prefixing the language name. For example, we could write the event listener in Groovy as follows.

```
<vlayout onClick="groovy:self.appendChild(new Label('New'));">
Click me!
</vlayout>
```

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Window.html#setTitle(java.lang.String)
[2] ZK uses BeanShell (http://www.beanshell.org) to interpret it at run time

Special Attributes

There are a few special attributes dedicated to special functionality rather than assigning properties or handling events. For example, the forEach attribute is used to specify a collection of object such that the XML element it belongs will be evaluated repeatedly for each object of the collection.

```
<listbox>
    <listitem forEach="${customers}" label="${each.name}"/>
</listbox>
```

For more information about special attributes, please refer to the Iterative Evaluation section and the ZUML Reference

A XML Text Represents Label Component or Property's Value

In general, a XML text is interpreted as a label component. For example,

```
<window>
  Begin ${foo.whatever}
</window>
```

is equivalent to

```
<window>
  <label value="Begin ${foo.whatever}"/>
</window>
```

A XML Text as Property's Value

Depending on the component's implementation, the text nested in a XML element can be interpreted as the value of a component's particular property. For example, Html (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Html.html#) is one of this kind of components, and

```
<html>Begin ${foo.whatever}</html>
```

is equivalent to

```
<html content="Begin ${foo.whatever}"/>
```

This is designed to make it easy to specify multiple-line value. This is usually used by a particular component that requires a multiple-lines value. For a complete list of components that interprets the XML text as a property's value, please refer to the ZUML Reference.

A XML Processing Instruction Specifies the Page-wide Information

Each XML processing instruction specifies the instruction on how to process the XML document. It is called directives in ZK. For example, the following specifies the page title and style.

```
<?page title="Grey background" style="background: grey"?>
```

Notice that there should be *no* whitespace between the question mark and the processing instruction's name (i.e., page in the above example).

The other directives include the declaration of components, the class for initializing a page, the variable resolver for EL expressions, and so on. For more information about directives, please refer to ZUML Reference.

Version History

Version	Date	Content

EL Expressions

Overview

EL expressions are designed to make a ZUML document easier to access objects available in the application, such as the application data and parameters.

An EL expressions is an expression enclosed with `${` and `}`, i.e., the syntax `${expr}`. For example,

```
<element attr1="${bean.property}".../>
${map[entry]}
<another-element>${3+counter} is ${empty map}</another-element>
```

When an EL expression is used as an attribute value, it could return any kind of objects as long as the attribute allows. For example, the following expressions will be evaluated to `boolean` and `int` respectively.

```
<window if="${some > 10}"><!-- boolean -->
  <progressmetter value="${progress}"/><!-- integer -->
```

If the class does not match, ZK Loader will try to coerce it to the correct one. If a failure has occurred, an exception is thrown.

Multiple EL expressions could be specified in a single attribute:

```
<window title="${foo.name}: ${foo.version}">
```

Example

EL Expression	Result
${1 > (4/2)}	false
${100.0 == 100}	true
${'a' < 'b'}	true
${'hip' gt 'hit'}	false
${1.2E4 + 1.4}	12001.4
${3 div 4}	0.75
${10 mod 4}	2
${empty param.add}	true if the request parameter named `add` is null or an empty string
${param['mycom.productId']}	The value of the request parameter named `mycom.productId`

- The example is from JSP Tutorial [1].
- For more information please refer to Operators and Literals.

Difference from Java

- A string can be enclosed with either single quotes or double quotes. In other words, 'abc' is equivalent "abc".
- The empty operator is useful for testing null and empty string, list and map, such as ${empty param.add}.
- The . operator can be used to access a property of an object (assuming that there is a get method of the same name) or a value of a map, such as ${foo.value.name}.
- The [ ] operator can be used to access an item of a list or array, a value of a map, and a property of an object (assuming that there is a get method of the same name), such as ${ary[5]} and ${wnd['title']}.
- null is returned if the value is not found and the index is out-of-bound.

For more information please refer to Operators and Literals.

Connecting to Java World

EL expressions are evaluated on the server when the page is rendered. Thus, it is allowed to access:

- Components by using its ID
- Variables defined in zscript
- Implicit objects

```
<window title="EL">
    <textbox id="tb" value="${self.parent.title}"/> <!-- self is an implicit object refer
    ${tb.value} <!-- tb is an ID (of textbox) -->
    <button label="Enter" if="${not empty param.edit}"/>
    <zscript>Date now = new Date();</zscript>
    <datebox value="${now}"/> <!-- now is defined in zscript -->
</window>
```

Furthermore, you could define a variable resolver to associate a name with an object, or map a function to a Java static method as described in the following.

Variable Resolver

If you would like to support many variables, you could implement a variable resolver: a class that implements VariableResolver [2].

```
package foo;
public class CustomerResolver implements org.zkoss.xel.VariableResolver
 {
    public Object resolveVariable(String name) {
        if ("customers".equals(name))
            return Customer.getAll("*");
//      if ("recent".equals(name))
//          return something_else;
        return null; //not a recognized variable
    }
}
```

Then, you could specify it in a variable-resolver directive, such as:

```
<?variable-resolver class="foo.CustomerResolve"?>

<listbox>
    <listitem label="${each.name}" forEach="${customers}"/>
</listbox>
```

System-level Variable Resolver

If you have a variable resolver that will be used on every page, you can register a system-level variable resolver rather than specifying it on every page.

This can be done by specifying a variable resolver you have implemented in WEB-INF/zk.xml as follows. For more information, please refer to ZK Configuration Reference.

```
<listener>
    <listener-class>foo.MyVariableResolver</listener-class>
</listener>
```

Then, when a page is created each time, an instance of the specified class will be instantiated and registered as if it is specified in the variable-resolver element.

Notice that since a new instance of the variable resolver is created on each page, there will not be any concurrency issues.

Associate with a Java Method

The collection object could be retrieved by invoking a static method. For example, suppose that we have a class and a static method as follows:

```
package foo;
public class Customer {
    public static Collection<Customer> getAll(String condition) {
        //...returns a collection of customers
    }
    public String getName() {
```

```
        return _name;
    }
    //...
}
```

Then, we could retrieve them with the xel-method directive:

```
<?xel-method prefix="c" name="getAllCustomers" class="foo.Customer"
   signature="java.util.Collection getAll(java.lang.String)"?><!-- Generics not allowed -
<listbox>
    <listitem label="${each.name}" forEach="${c:getAllCustomers('*')}"/>
</listbox>
```

Associate with Multiple Java Methods

If you have several static methods, you could declare them in a XML file called taglib, such as

```
<taglib>
      <function>
            <name>getAllCustomers</name>
            <function-class>foo.Customer</function-class>
            <function-signature>
      java.util.Collection getAll(java.lang.String)
            </function-signature>
            <description>
      Returns a collection of customers.
            </description>
      </function>
      <!-- any number of functions are allowed -->
</taglib>
```

Then, you could use them by specifying it in a taglib directive.

```
<?taglib uri="/WEB-INF/tld/my.tld" prefix="my"?>
<listbox>
    <listitem label="${each.name}" forEach="${my:getAllCustomers('*')}"/>
</listbox>
```

Version History

Version	Date	Content

References

[1] http://download.oracle.com/javaee/1.4/tutorial/doc/JSPIntro7.html
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/xel/VariableResolver.html#

Scripts in ZUML

Embed Server-side Script Code

To make it easier to create a dynamic web page, the ZUML document allows you to embed the script code. Notice that there are two types of script code: server-side and client-side. How the client-side code can be embedded is discussed at the Client-side UI Composing and Client-side Event Listening sections. Here we will discuss how to embed the server-side script code in a ZUML document.

Depending on the requirement, there are two ways to embed the server-side script code in a ZUML document: the `zscript` element and the event handler. The `zscript` element is used to embed the code that will execute when the page is loaded, while the event handler will execute when the event is received.

Notice that the performance of BeanShell is not good and, like any interpreter, typos can be found only when it is evaluated[1] . However, embedding Java code in a ZUML page is a powerful tool for fast prototyping. For example, business analysis could discuss the *real UI* with UI designers, modify it directly and get back the feeling immediately without going through drawings and even recompiling.

[1] For more information, please refer to the Performance Tips section

zscript

First, you could embed the code inside the zscript element, such that they will be evaluated when the page is rendered[1] . For example,

```
<zscript>
//inside is zscript
//you can declare variable, function, and even Java class here.
void foo(String msg) {
    //...
}
comp.addEventListener("onClick",
    new EventListener() {
        public void onEvent(Event event) {
            //...
        }
    });
</zscript>
```

Notice that, by default, the code inside the `zscript` element is Java but you could also use other languages, such as Groovy. Keep in mind that it is *interpreted* at run time (by Beanshell (http://beanshell.org)), so typo or syntax error will be found only when it is interpreted. In addition, it runs at the server, so it could access any Java libraries. You could even define variables, methods, classes with it, and they are visible to EL expressions of the same page.

CDATA

The code embedded in the zscript element must be a valid XML text. In other words, you must encode the special characters well, such as < must be replaced with <, & with & and so on. In additions to encode individual characters, you could also encode the whole code with XML CDATA as follows.

```
<script><![CDATA[
if (some < another && another < last) //OK since CDATA is used
   doSomething();
]]></script>
```

As depicted CDATA is represented with `<![CDATA[` and `]]>`.

---

[1] The zscript element has an attribute called deferred that could make the evaluation as late as possible

Class Declaration

You could define a class declared in a ZUML document, and the class is accessible only in the page it was defined. For example,

```
<?xml version="1.0" encoding="UTF-8"?>
<zk>
<zscript><![CDATA[
public class FooModel extends AbstractTreeModel {
    public FooModel() {
        super("Root");
    }
    public boolean isLeaf(Object node) {
        return getLevel((String)node) >= 4; //at most 4 levels
    }
    public Object getChild(Object parent, int index) {
        return parent + "." + index;
    }
    public int getChildCount(Object parent) {
        return 5; //each node has 5 children
    }
    public int getIndexOfChild(Object parent, Object child) {
        String data = (String)child;
        int i = data.lastIndexOf('.');
        return Integer.parseInt(data.substring(i + 1));
    }
    private int getLevel(String data) {
        for (int i = -1, level = 0;; ++level)
            if ((i = data.indexOf('.', i + 1)) < 0)
                return level;
    }
};
FooModel model = new FooModel();
]]></zscript>
<tree model="${model}">
    <treecols>
```

```
        <treecol label="Names"/>
    </treecols>
</tree>
</zk>
```

Event Handlers

Second, you could put the code inside an event handler, such that it will execute when the event is received, as depicted below.

```
<button onClick='alert("event handler for onXXX inside ZUML is also zscript")'/>
```

Notice that the the name of the event must starts with `on`, and the third letter must be a **upper** case. Otherwise, it will be considered as a property.

Again, the code is Java interpreted at run time and running on the server. For client-side listening, please refer to the Client-side Event Listening section.

For the sake of discussion, we call it zscript no matter the code is embedded in the `zscript` element or in an event handler.

Attribute

If the code is too complicate, you could specify the event handle in the attribute element. For example,

```
<button label="hi">
    <attribute name="onClick"><![DATA[
    if (anything > best)
        best = anything;
    ]]></attribute>
</button>
```

Distinguish `zscript` from EL

Keep in mind, an EL expression is enclosed by ${ }.

For example, `${self.label}` and `${ok.label}` are both EL expressions in the following example:

```
<window>
      <button label="ok" id="${self.label}"/>
      ${ok.label}
</window>
```

On the other hand, in the following example, `alert(self.label)` is not an EL expression. Rather, it's the zscript code:

```
<window>
      <button label="ok" onClick='alert(self.label)'/>
</window>
```

You cannot mix the use of EL expressions with zscript:

```
<window>
      <!-- It's wrong, for java don't accept syntax as ${}-->
      <button label="ok" onClick='alert(${self.label})'/>
```

```
</window>
```

Also notice that the evaluation of EL expressions is very fast, so EL can be used in a production system. On the other hand, zscript is suggested to use only in prototyping or quick-fix.

Variables Defined in zscript Visible to EL

A variable defined in zscript is visible to EL expression, unless it is a local variable, which will be discussed later.

```
<script>
Date now = new Date();
</script>
${now}
```

Java Interpreter

The default interpreter is based on BeanShell (http://www.beanshell.org). It is a Java Interpreter.

Scope for Each ID Space

The Java interpreter is a *multi-scope* interpreter. It creates a scope for each ID space. Since ID space is hierarchical, so is the scopes. If a variable cannot be found in the current ID space, it will go further to parent's ID space try to resolve the variable.

For example, in the following example, two logical scopes are created for window[1] `A` and `B` respectively. Therefore, `var2` is visible only to window `B`, while `var1` is visible to both window `A` and `B`.

```
<window id="A">
    <zscript>var1 = "abc";</zscript>
    <window id="B">
        <zscript>var2 = "def";</zscript>
    </window>
</window>
```

[1] Built in id space owner includes Window (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Window.html#), Page (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Page.html#) and [[ZK Developer's Reference/UI Composing/Macro Component|macro components.

Declare a Local Variable

If a variable is declared inside a pair of the curly braces, it is visible only to the scope defined by the curly braces. It is called a local variable. For example,

```
<zscript>
void echo() {
   String a_local_variable;
}
</script>
```

Here is another example,

```
<window>
      <zscript>
```

```
      {
          Date now = new Date(); //local variable
          abc ="def"; //global variable since not defined before and
not Class specified
      }
      String first = "first"; //global variable
      </zscript>
      0: ${first}
      1:${abc}
      2:${now}
</window>
```

The result shows: `0:first 1:def 2:` . It is because `now` is the local variable and it is invisible to EL expressions. On the other hand, `first` and `abc` are both global variables that are visible to EL expressions. Notice that `abc` is not declared but assigned directly, and it causes a global variable to be created.

Please refer to the Beanshell Documentation (http://beanshell.org/docs.html) and search "scoping" and "local" for more information.

Use Other Languages

Currently, zscript supports Java, Groovy, Ruby, JavaScript and Python. For example,

```
<?page zscriptLanguage="Groovy"?>
<window border="normal">
      <vbox id="vb">
            <label id="l" value="Hi"/>
            <button label="change label" onClick="l.value='Hi, Groovy';"/>
            <button label="add label" onClick="new Label('New').setParent(vb);"/>
      </vbox>
      <button label="alert" onClick="alert('Hi, Groovy')"/>
</window>
```

In addition, you could add your own interpreter by implementing Interpreter (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/scripting/Interpreter.html#). For more information, please refer to ZUML Reference.

Version History

Version	Date	Content

Conditional Evaluation

If and Unless

The evaluation of an element could be conditional. By specifying the `if`, `unless` attribute or both, developers could control whether to evaluate the associated element. It is also the most straightforward way.

For example, suppose that we want to use label, if readonly, and textbox, otherwise:

```
<label value="${customer.label}" if="${param.readonly == 'true'}"/>
<textbox value="${customer.value}" unless="${param.readonly == 'true'}"/>
```

Here is another example: window is created only if `a` is 1 and `b` is not 2. If an element is ignored, all of its child elements are ignored too.

```
<window if="${a==1}" unless="${b==2}">
    ...
</window>
```

Switch and Case

With the `switch` and `case` attributes of the `zk` element, you can evaluate a section of a ZUML document only if a variable matches a certain value. It is similar to Java's switch statement.

For example,

```
<zk switch="${fruit}">
    <zk case="apple">
    Evaluated only if ${fruit} is apple
    </zk>
    <zk case="${special}">
    Evaluated only if ${fruit} equals ${special}
    </zk>
    <zk>
    Evaluated only if none of the above cases matches.
    </zk>
</zk>
```

ZK Loader will evaluate from the first case to the last case, until it matches the switch condition, which is the value specified in the switch attribute. The evaluation is mutually exclusive conditional. Only the first matched case is evaluated.

The zk element without any case is the default – i.e., it always matches and is evaluated if all the cases above it failed to match.

Multiple Cases

You can specify a list of cases in one case attribute, such that a section of a ZUML document has to be evaluated if one of them matches.

```
<zk switch="${fruit}">
    <zk case="apple, ${special}">
    Evaluated if ${fruit} is either apple or ${special}
    </zk>
</zk>
```

Regular Expressions

Regular expressions are allowed in the `case` attribute too, as shown below.

```
<zk switch="${fruit}">
    <zk case="/ap*.e/">
    Evaluate if the regular expression, ap*.e"., matches the switch
condition.
    </zk>
</zk>
```

Used with forEach

Like any other elements, you can use the the forEach attribute (so are if and unless). The `forEach` attribute is evaluated first, so the following is the same as multiple cases.

```
<zk case="${each}" forEach="apple, orange">
```

is equivalent to

```
<zk case= "apple, orange">
```

Choose and When

The `choose` and `when` attributes of the `zk` element is the third approach of conditional evaluation.

As shown below, it is enclosed with a `zk` element with the `choose` attribute, and the ZK Loader will evaluate its child elements (the `zk` elements with the `when` attribute) one-by-one until the first one matches:

```
<zk choose="">
    <zk when="${fruit == 'apple'}">
    Evaluated if the when condition is true.
    </zk>
    <zk><!-- default -->
    Evaluated if none of above cases matches.
    </zk>
</zk>
```

You don't have to assign any value to the `choose` attribute, which is used only to identify the range of the mutually exclusive conditional evaluation.

Version History

Version	Date	Content

Iterative Evaluation

forEach

By default, ZK instantiates a component for each XML element. If you would like to generate a collection of components, you could specify the forEach attribute. For example,

```
<listbox>
    <listitem label="${each}" forEach="Apple, Orange, Strawberry"/>
</listbox>
```

is equivalent to

```
<listbox>
    <listitem label="Apple"/>
    <listitem label="Orange"/>
    <listitem label="Strawberry"/>
</listbox>
```

When ZK Loader iterates through items of the given collection, it will update two implicit objects: each and forEachStatus. The `each` object represents the item being iterated, while forEachStatus is an instance of ForEachStatus [1], from which you could retrieve the index and the previous forEach, if any (nested iterations).

If you have a variable holding a collection of objects, you can specify it directly in the forEach attribute. For example, assume that you have a variable called `grades` as follows.

```
grades = new String[] {"Best", "Better", "Good"};
```

Then, you can iterate them by the use of the forEach attribute as follows. Notice that you have to use EL expression to specify the collection.

```
<listbox>
    <listitem label="${each}" forEach="${grades}"/>
</listitem>
```

The iteration depends on the type of the value of the forEach attribute:

- If it is java.util.Collection iterates each element of the collection.
- if it is java.util.Map, it iterates each Map.Entry of the map.
- If it is java.util.Iterator, it iterates each element from the iterator.
- If it is java.util.Enumeration, it iterates each element from the enumeration.
- If it is Object[], int[], short[], byte[], char[], float[] or double[] is specified, it iterates each element from the array.
- If it is null, nothing is generated (it is ignored).
- If neither of the above types is specified, the associated element will be evaluated once as if a collection with a single item is specified.

The each Object

During the evaluation, an object called `each` is created and assigned with the item from the specified collection. In the above example, `each` is assigned with "Best" in the first iteration, then "Better" and finally "Good".

Notice that the `each` object is accessible both in an EL expression and in zscript. ZK will preserve the value of the `each` object if it is defined before, and restore it after the evaluation of the associated element.

The forEachStatus Object

The `forEachStatus` object is an instance of ForEachStatus [2]. It holds the information about the current iteration. It is mainly used to get the item of the enclosing element that is also assigned with the forEach attribute.

In the following example, we use nested iterative elements to generate two listboxes.

```
<hlayout>
    <zscript>
    classes = new String[] {"College", "Graduate"};
    grades = new Object[] {
        new String[] {"Best", "Better"}, new String[] {"A++", "A+", "A"}
    };
    </zscript>
    <listbox width="200px" forEach="${classes}">
        <listhead>
            <listheader label="${each}"/>
        </listhead>
        <listitem label="${forEachStatus.previous.each}: ${each}"
         forEach="${grades[forEachStatus.previous.index]}"/>
    </listbox>
</hlayout>
```

Notice that the `each` and `forEachStatus` objects can be accessible both in an EL expression and in zscript.

Apply forEach to Muliple Elements

If you have to iterate a collection of items for multiple XML elements, you could group them with the zk element as shown below.

```
<zk forEach="${cond}">
    ${each.name}
    <textbox value="${each.value}"/>
    <button label="Submit"/>
</zk>
```

The zk element is a special element used to *group* a set of XML element nested. ZK Loader will not create a component for it. Rather, it interprets the `forEach`, if and unless attribute it might have.

Access each and forEachStatus in Java

You could access the `each` and `forEachStatus` object directly in zscript such as:

```
<window>
    <button label="${each}" forEach="apple, orange">
        <zscript>
   self.parent.appendChild(new Label("" + each));
        </zscript>
    </button>
</window>
```

In a composer, you could retrieve them from the attributes, because these objects are actually stored in the parent component's attributes (Component.getAttribute(java.lang.String) [3]). For example,

```
public class Foo implements Composer {
    public void doAfterCompose(Component comp) throws Exception {
        Object each = comp.getParent().getAttribute("each"); //retrieve
 the each object
        ForEachStatus forEachStatus =
(ForEachStatus)comp.getParent().getAttribute("forEachStatus");
        //...
    }
}
```

If the component is a root, you could retrieve them from the page's attributes (Page.getAttribute(java.lang.String) [4]).

Access each and forEachStatus in Event Listeners

However, you cannot access the values of `each` and `forEachStatus` in an event listener because their values are reset after the XML element which `forEach` is associated has been evaluated.

For example, the following code will not work:

```
<button label="${each}" forEach="${countries}"
    onClick="alert(each)"/> <!-- incorrect!! -->
```

When the onClick event is received, the `each` object no longer exists.

There is a simple solution: store the value in the component's attribute, so you can retrieve it when the event listener is called. For example,

```
<button label="${each}" forEach="${countries}"
    onClick='alert(self.getAttribute("country"))'>
        <custom-attributes country="${each}"/>
</button>
```

Iterate a Subset of a Collection

If you would like to iterate a subset of a collection, you could specify the forEachBegin and/or forEachEnd attributes.

```
<grid>
    <rows>
        <row forEach="${foos}" forEachBegin="${param.begin}" forEachEnd="${param.end}">
            ${each.name} ${each.title}
        </row>
    </rows>
</grid>
```

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ForEachStatus.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/ui/util/ForEachStatus.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#getAttribute(java.lang.String)
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Page.html#getAttribute(java.lang.String)

On-demand Evaluation

By default, ZK creates components based on what are defined in a ZUML document when loading the document. However, we can defer the creation of some sections of components, until necessary, such as becoming visible. This technique is called load-on-demand or render-on-demand.

For example, you could split a ZUML document into multiple pages, and then load the required ones when necessary. Please refer to the Load ZUML in Java section for how to load a ZUML document dynamically.

It improves the performance both at the server and client sides. It is suggested to apply this technique whenever appropriate. In addition, ZK Loader provides a standard on-demand evaluation called *fulfill* to simplify the implementation as described in the following section.

Load-on-Demand with the fulfill Attribute

The simplest way to defer the creation of the child components is to use the fulfill attribute. For example, the comboitem in the following code snippet will not be created, until the combobox receives the `onOpen` event, indicating that comboitem is becoming visible.

```
<combobox fulfill="onOpen">
    <comboitem label="First Option"/>
</combobox>
```

In other words, if a XML element is specified with the fulfill attribute, all of its child elements will not be processed until the event specified as the value of the fulfill attribute is received.

Specify Target with its ID

If the event to trigger the creation of children is targeted at another component, you can specify the target component's identifier in front of the event name as depicted below.

```
<button id="btn" label="show" onClick="content.visible = true"/>
<div id="content" fulfill="btn.onClick">
    Any content created automatically when btn is clicked
</div>
```

Specify Target with its Path

If the components belong to a different ID space, you can specify a path before the event name as follows:

```
<button id="btn" label="show" onClick="content.visible = true"/>
<window id="content" fulfill="../btn.onClick">
    Any content created automatically when btn is clicked
</window>
```

Specify Target with EL Expressions

EL expressions are allowed to specify the target, and it must return a component, an identifier or a path.

```
<div fulfill="${foo}.onClick">
...
</div>
```

Specify Multiple Fulfill Conditions

If there are multiple conditions to fulfill, you could specify all of them in the fulfill attribute by separating them with a comma, such as

```
<div fulfill="b1.onClick, ${another}.onOpen">
...
</div>
```

Load Another ZUML on Demand with the fulfill Attribute

You could specify an URI in the fulfill attribute when the fulfill condition is satisfied (i.e. if a specified event has been received). The ZUML document of the URI will be loaded and rendered as the children of the associated component. To specify an URI, just append it to the condition and separate with an equal sign (=). For example,

```
<zk>
    <button id="btn" label="Click to Load"/>
    <div fulfill="btn.onClick=another.zul"/>
</zk>
```

Then, `another.zul` will be loaded when the button is clicked.

Notice that even though you could specify multiple conditions, you could specify at most one URI. The ZUML document of the URI will be loaded no matter which condition is satisfied.

```
<div fulfill="btn.onClick, foo.onOpen=another.zul"/>
```

If you specify an URI without any conditions, the ZUML document of the URI will be loaded from the very beginning. In other words, it has the same effect of using include.

```
<div fulfill="=another.zul"/>
```

The onFulfill Event

After ZK applies the fulfill condition, i.e., creates all descendant components, it fires the `onFulfill` event with an instance of FulfillEvent [1] to notify the component for further processing if any.

For example, if you use the `wireVariables` method of the Components [2] class, you might have to call `wireVariables` again to wire the new components in the `onFulfill` event.

```
<div fulfill="b1.onClick, b2.onOpen" onFulfill="Components.wireVariables(self, controller
      ...
</div>
```

Version History

Version	Date	Content

References

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

Include

Include allows you to include a ZUML page, a static page, a JSP page or the result of a servlet. For example,

```
<include src="another.zul"/>
<include src="another.jsp"/>
```

When including a non-ZUML page (such as JSP), the output of the page will be the content of the Include [14] component. Thus, the output must be a valid HTML fragment.

When including a ZUML page, the components specified in the ZUML page will become the child components of the Include [14] component.

For example, suppose we have two ZUL pages as follows:

```
<!-- first.zul -->
<include src="second.zul"/>
```

and

```
<!-- second.zul -->
<listbox>
    <listitem label="foo"/>
</listbox>
```

Then, listbox in second.zul will become the child of include in first.zul.

If you prefer to create an independent page (Page [8]), or want to include a page rendered by `Richlet` while the value of src ends with `.zul` or `.zhtml`, you could specify the mode with defer (Include.setMode(java.lang.String) [1]). Then, include won't have any child. Rather, an instance of Page [8] will be created to hold the content of second.zul or the content generated by Richlet. For more information, please refer to ZK Component Reference: include.

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Include.html#setMode(java.lang.String)

Load ZUML in Java

Overview

Execution [1] provides a collection of methods to allow you to create components based on a ZUML document, such as org.zkoss.zk.ui.Component, java.util.Map) Execution.createComponents(java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) [2], java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) Execution.createComponentsDirectly(java.lang.String, java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) [3] and many others. In addition, Executions [4] provides a similar collection of shortcuts so that you do not have to retrieve the current execution first.

For example,

```
public class Controller extends SelectorComposer {
    @Wire
    private Window main; //assumed wired automatically
    @Listen(onClick = #main)
    public void createListbox() {
        Executions.createComponentsDirectly(
           "<listbox><listitem label=\"foo\"/></listbox>", "zul", this, null);
    }
...
```

Create from URI

There are several ways to create components based on a ZUML document. One of the most common approach is to create components from a URI.

```
Map arg = new HashMap();
arg.put("someName", someValue);
Executions.createComponents("/foo/my.zul", parent, arg); //attach to
page as root if parent is null
```

where parent (an instance of Component [1]) will become the parent of the components specified in the ZUML document. If parent is null, the components specified in the ZUML documents will become the root components of the current page. In other words, the components created by org.zkoss.zk.ui.Component, java.util.Map) Execution.createComponents(java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) [2] will be attached to the current page.

The arg Object

The map passed to the `createComponents` method can be accessed in the page being created by use of the arg object. For example,

```
<button label="Submit" if="${arg.someName}"/>
```

Create Components Not Attached to Any Pages

If you want to create components that will not be attached to a page, you could use java.util.Map) Execution.createComponents(java.lang.String, java.util.Map) [2]. It is useful if you wan to maintain a cache of components or implement a utility. For example,

```
Map arg = new HashMap();
arg.put("someName", someValue);
Component[] comps =
Executions.getCurrent().createComponents("/foo/my.zul", arg); //won't
be attached to a page
cache.put("pool", comps); //you can store and use them later since they
 are attached to any pages
```

Create Components in Working Thread

With java.lang.String, java.util.Map) Executions.createComponents(org.zkoss.zk.ui.WebApp, java.lang.String, java.util.Map) [5], you could create components in a working thread without execution[6] , though it is rare.

Of course, the components being created by java.lang.String, java.util.Map) Executions.createComponents(org.zkoss.zk.ui.WebApp, java.lang.String, java.util.Map) [5] will not be attached to any pages. You have to attach them manually, if you want to show them to the client.

---

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Execution.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Execution.html#createComponents(java.lang.String,
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Execution.html#createComponentsDirectly(java.lang.String,
[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#createComponents(org.zkoss.zk.ui.WebApp,
[6] It means Executions.getCurrent() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#getCurrent()) returns null. For example, it happens when the application starts, or in a working thread.

Create from Content Directly

If the ZUML document is a resource of Web application (i.e., not accessible through ServletContext), you could use one of the createComponentsDirectly methods. For example, you could read the content into a string from database and pass it to java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) Execution.createComponentsDirectly(java.lang.String, java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Execution.html#createComponentsDirectly(java.lang.String,). Or, you could represent the content as a reader (say, representing BLOB in database) and then pass it to java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) Execution.createComponentsDirectly(java.io.Reader,

java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Execution.html#createComponentsDirectly(java.io.Reader,)

For example, suppose we want to create a component from a remote site. Then, we could represent the resource as a URL and do as follows.

```
public void loadFromWeb(java.net.URL src, Component parent) {
    return Executions.createComponentsDirectly(
        new java.io.InputStreamReader(src.openStream(), "UTF-8"),
parent, null);
}
```

Create from Page Definition

When creating components from the URI (such as org.zkoss.zk.ui.Component, java.util.Map) Execution.createComponents(java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Execution.html#createComponents(java.lang.String,)), ZK Loader will cache the parsed result and reuse it to speed up the rendering.

However, if you create components from the content directly (such as java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) Execution.createComponentsDirectly(java.lang.String, java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Execution.html#createComponentsDirectly(java.lang.String,)), there is no way to cache the parsed result. In other words, the ZUML content will be parsed each time createComponentsDirectly is called.

It is OK if the invocation does not happen frequently. However, if you want to improve the performance, you could parse the content into PageDefinition (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/metainfo/PageDefinition.html#) by using java.lang.String, java.lang.String) Executions.getPageDefinitionDirectly(org.zkoss.zk.ui.WebApp, java.lang.String, java.lang.String) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#getPageDefinitionDirectly(org.zkoss.zk.ui.WebApp,), cache it, and then invoke org.zkoss.zk.ui.Component, java.util.Map) Executions.createComponents(org.zkoss.zk.ui.metainfo.PageDefinition, org.zkoss.zk.ui.Component, java.util.Map) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#createComponents(org.zkoss.zk.ui.metainfo.PageDefinition,) to create them repeatedly.

PageDefinition (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/metainfo/PageDefinition.html#) is a Java object representing a ZUML document. It is designed to allow ZK Loader to interpret even more efficiently. Unfortunately, it is not serializable, so you can not store it into database or other persistent storage. You could serialize or marshal the original content (i.e., ZUML document) if required.

Notices

There are a few notices worth to know.

No Page Created

When creating components from a ZUML document as described above, no page (Page (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Page.html#)) is created. Components are attached to the current page, to a component, or simply standalone. Since no page is created, there are a few differences than visiting a ZUML document directly[1] .

1. The <?page?>, <?script?>, <?link?>, <?header?> and other directives controlling a page (Page (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Page.html#)) have no function. It means that you could not change the page's title, add JavaScript code, or add CSS with these directive in a ZUML document loaded in this way.
2. On the other hands, when <?function-mapper?>, <?variable-resolver?> and <?component?> work correctly, they decide how a ZUML document is parsed rather than how the current page (Page (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Page.html#)) should be.
3. The variables, functions and classes defined in zscript will be stored in the interpreter of the current page (Page.getInterpreter(java.lang.String) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Page.html#getInterpreter(java.lang.String))).
   - If java.util.Map) Execution.createComponents(java.lang.String, java.util.Map) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Execution.html#createComponents(java.lang.String,), java.lang.String, java.util.Map) Executions.createComponents(org.zkoss.zk.ui.WebApp, java.lang.String, java.util.Map) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#createComponents(org.zkoss.zk.ui.WebApp,) or similar is used to create components not attached to any page, the variables, functions and classes defined in the ZUML document will be lost. Thus, it is a *not* a good idea to use zscript in this case.

---

[1] Don't confuse a ZUML page with Page (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Page.html#). The former refers to a file containing a ZUML document. The later is a Java object of represents a portion of a desktop.

Version History

Version	Date	Content

XML Namespaces

In a ZUML document, a XML namespace is used to identify either a special functionality or a component set. We call the former as a standard namespace, while the later as a language.

Standard Namespaces

For example, the client namespace is used to identify that a XML attribute shall be interpreted as a client-side control.

In the following example, `w:onFocus` is a client-side listener since `w:` is specified, while `onChange` is

```
<combobox xmlns:w="client" w:onFocus="this.open()" onChange="doOnChange()"/>
```

The native namespace is another standard namespace used to indicate that a XML element should be generated *natively* rather than a component. For example,

```
<n:table xmlns:n="native">
  <n:tr>
    <n:td>Username</n:td>
    <n:td><textbox/></n:td>
  </n:tr>
  <n:tr>
    <n:td>Password</n:td>
    <n:td><textbox type="password"/></n:td>
  </n:tr>
</n:table>
```

where `n:table`, `n:tr` and `n:td` are native, i.e., they are generated directly to the client without creating a component for each of them.

For more information, please refer to ZUML Reference.

Languages

A language (LanguageDefinition [1]) is a collection of component definitions. It is also known as a component set.

For example, Window [1], Button [2] and Combobox [2] all belong to the same language called xul/html. It is a ZK variant of XUL (and also known as zul).

Component designers are free to designate a component definition to any component sets they prefer, as long as there is no name conflict.

When parsing a ZUML document, ZK Loader have to decide the language that a XML element is associated, so that the correct component definition (ComponentDefinition [3]) can be resolved. For example, in the following example, ZK needs to know window belongs to the xul/html language, so its component definition can be retrieved correctly.

```
<window>
```

ZK Loader first decides the default language from the extension. For example, `foo.zul` implies the default language is ZUL. The default language is used if an XML element is not specified with any XML namespace. For example, `window` in the previous example will be considered as a component definition of the ZUL langauge.

If the extension is zhtml (such as `foo.zhtml`), the default language will be XHTML. Thus, `window` in the previous example will be interpreted incorrectly. To solve it, you could specify the XML namespace explicitly as

follows.

```
<!-- foo.zhtml -->
<p> <!-- assumed from the XHTML language -->
    <u:window xmlns:u="zul"/> <!-- ZK Loader will search the ZUL language instead -->
</p>
```

For more information about identifying a language, pelase refer to ZUML Reference.

Version History

Version	Date	Content
5.0.4	August, 2010	The shortcut was introduced to make it easy to specify a standard namespace, such as native, client and zk.
5.0.5	October, 2010	The shortcut was introduced to make it easy to specify a component set, such as zul and zhtml.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/metainfo/LanguageDefinition.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Combobox.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/metainfo/ComponentDefinition.html#

Richlet

Overview

A richlet is a small Java program that composes a user interface in Java for serving the user's request.

When a user requests the content of an URL, ZK Loader checks if the resource of the specified URL is a ZUML page or a richlet. If it is a ZUML page, ZK Loader will create components automatically based on the ZUML page's content as we described in the previous chapters.

If the resource is a richlet, ZK Loader hands over the processing to the richlet. What and how to create components are all handled by the richlet. In other words, it is the developer's job to create all the necessary components programmatically in response to the request.

The choice between the ZUML pages and richlets depends on your preference. However, the performance should not cause any concern since parsing ZUML is optimized.

Implement a Richlet

It is straightforward to implement a richlet. First, you have to implement the Richlet [1] interface before mapping a URL to the richlet.

Implement a Richlet as a Java class

A richlet must implement the Richlet [1] interface. However, you generally do not have to implement it from scratch. Rather, you could extend the richlet from GenericRichlet [2]. With GenericRichlet [2], the only thing you have to do is to implement Richlet.service(org.zkoss.zk.ui.Page) [13]. It is called when an associated URL is requested. For example,

```
package org.zkoss.zkdemo;

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

public class TestRichlet extends GenericRichlet {
    //Richlet//
    public void service(Page page) {
        page.setTitle("Richlet Test");

        final Window w = new Window("Richlet Test", "normal", false);
        new Label("Hello World!").setParent(w);
        final Label l = new Label();
        l.setParent(w);

        final Button b = new Button("Change");
        b.addEventListener(Events.ON_CLICK,
            new EventListener() {
                int count;
                public void onEvent(Event evt) {
                    l.setValue("" + ++count);
                }
            });
        b.setParent(w);

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

As shown above, we have to invoke page) Component.setPage(Page page) [3] explicitly to attach a root component to a page so it will be available at the client.

To have better control, you can even implement the Richlet.init(org.zkoss.zk.ui.RichletConfig) [4] and Richlet.destroy() [5] methods to initialize and to destroy any resources required by the richlet when it is loaded.

In addition, you could implement Richlet.getLanguageDefinition() [6] to use a different language as default (for example, implementing a richlet for mobile devices [7]). By default, ZUL (aka., xul/html) is assumed.

Richlet Must Be Thread-Safe

Like a servlet, a single instance of richlet is created and shared with all users for all requests for the mapped URL. A richlet must handle the concurrent requests, and be careful to syncrhronize access to shread resources. In other words, a richlet (the implementation of the `service` method) must be thread-safe.

Don't Share Components

When a request (not Ajax request but regular HTTP request) is made by an user, a Desktop [9] and a Page [8] are created first, and then Richlet.service(org.zkoss.zk.ui.Page) [13] is invoked to serve the request[8] . In other words, each request is served with an individual desktop and page. Therefore, we *cannot* share components among different invocation of Richlet.service(org.zkoss.zk.ui.Page) [13].

For example, the following code is illegal:

```
public class MyRichlet extends GenericRichlet {
    private Window main; //Not a good idea to share
    public void service(Page page) {
        if (main == null) {
            main = new Window();
        }
        main.setPage(main); //ERROR! Causes an exception if the same
URL is requested twice!
...
```

Why? Each deskop should have its own set of component instances[9] . When the URL associated **MyRichlet** is requested a second time, an exception will be thrown because the **main** window is already instantiated and associated with the first desktop created from the first request. We cannot assign it to the second desktop.

---

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Richlet.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/GenericRichlet.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#setPage(Page
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Richlet.html#init(org.zkoss.zk.ui.RichletConfig)
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Richlet.html#destroy()
[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Richlet.html#getLanguageDefinition()
[7] http://code.google.com/p/zkreach/
[8] A normal HTTP request; not an Ajax request. Ajax requests are handled in the same way as ZUML. For more information please refer to the Event Handling section
[9] For more information, please refer to Component-based UI section

Map URL to a Richlet

To map URL to a richlet, there are two steps.

1. Turn on the support of Richlet (in WEB-INF/web.xml)
2. Map URL pattern to Richlet (in WEB-INF/zk.xml)

Turn on Richlet

By default, richlets are disabled. To enable them, please add the following declaration to `WEB-INF/web.xml`. Once enabled, you can add as many as richlets as you want without modifying `web.xml`.

```
<servlet-mapping>
    <servlet-name>zkLoader</servlet-name>
```

```
    <url-pattern>/zk/*</url-pattern>
</servlet-mapping>
```

where you can replace /zk/* to any pattern you like, such as /do/*. Notice that you *cannot* map it to an extension (such as *.do) since it will be considered as a ZUML page (rather than a richlet).

Map URL pattern to Richlet

For each richlet you implement, you can define it in `WEB-INF/zk.xml` with the statement similar to the following:

```
<richlet>
    <richlet-name>Test</richlet-name><!-- your preferred name -->
    <richlet-class>org.zkoss.zkdemo.TestRichlet</richlet-class><!-- your class name, of c
</richlet>
```

After defining a richlet, you can map it to any number of URLs using the `richlet-mapping` element as shown below.

```
<richlet-mapping>
    <richlet-name>Test</richlet-name>
    <url-pattern>/test</url-pattern>
</richlet-mapping>
<richlet-mapping>
    <richlet-name>Test</richlet-name>
    <url-pattern>/some/more/*</url-pattern>
</richlet-mapping>
```

Then, you can visit http://localhost:8080/PROJECT_NAME/zk/test (http://localhost:8080/PROJECT_NAME/zk/test) to request the richlet.

The URL specified in the `url-pattern` element must start with `/`. If the URI ends with `/*`, it is matched to all request with the same prefix. To retrieve the request's actual URL, you can check the value returned by the `getRequestPath` method of the current page.

```
public void service(Page page) {
    if ("/some/more/hi".equals(page.getRequestPath()) {
        ...
    }
}
```

Tip: By specifying `/*` as the `url-pattern`, you can map all unmatched URLs to your richlet.

Load ZUML in Richlet

Execution (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Execution.html#) provides a collection of methods, such as org.zkoss.zk.ui.Component, java.util.Map) Execution.createComponents(java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zk/ ui/ Execution.html#createComponents(java.lang.String,), allowing developers to load ZUML documents dynamically. You could load a ZUML document from any source you like, such as database. Please refer to the Load ZUML in Java for details.

Use Spring in Richlet

To use Spring-managed beans in richlets you need the context loader listener that creates spring application context as described in ZK Spring Essentials/Getting Started with ZK Spring/Setting Up ZK Spring. Then you could load Spring beans by using a utility class SpringUtil (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/spring/SpringUtil.html#):

```
    Object bean = SpringUtil.getBean(beanName);
```

Version History

Version	Date	Content

Macro Component

There are two ways to implement a component. One is to implement a component in a Java class, extending from other component or one of the skeletal implementations with an optional JavaScript class. It is flexible and, technically, is also able to implement any functionality you want. For more information please refer to ZK Component Development Essentials.

On the other hand, we could implement a new component by using the others and composing them in a ZUML page. In other words, we could define a new component by expressing it in a ZUML page. It works like composition, macro expansion, or inline replacement.

For the sake of convenience, we call the first type of components as *primitive components*, while the second type as *macro components*. In this section we will get into more details on how to implement a macro component and how to use it.

There is a similar concept called composite components. Unlike macros, you could derive from any component but you have to do the loading of ZUML manually. For more information please refer to the Composite Component section.

Definition, Declaration and Use

It is straightforward to apply macro components to an application:

1. Define (aka., Implement) a macro component in a ZUML page.
2. Declare the macro component in the page or the whole application that is going to use the macro component.
3. Use the macro components. The use of a macro component is the same of using primitive components.

Define Macro Component

The definition of a macro component is expressed in a ZUML page. In other words, the page is the template of the macro component. It is the same as any other ZUML pages as it does not require any special syntaxes at all. Furthermore, any ZUML page can be used as a macro component too.

For example, assume that we want to pack a label and a text box as a macro component. Then we could create page, say `/WEB-INF/macros/username.zul`, as follows.

```
<hlayout>
      Username: <textbox/>
```

```
</hlayout>
```

It is done.

Declare Macro Component

Before using a macro component, you have to declare it first. It is straightforward to use component directives. For example, we could add the first line to the page that is going to use the *username* macro component:

```
<?component name="username" macroURI="/WEB-INF/macros/username.zul"?>
```

As shown, we have to declare the component's name (the `name` attribute) and the URI of the page defining the macro component (the `macroURI` attribute).

If you prefer to make a macro component available to all pages, you could add the component definition to the so-called language addon and add it to WEB-INF/zk.xml.

Use Macro Component

Using a macro component in a ZUML page is the same as the use of any other components. There is no difference at all

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

Pass Properties to Macro Component

Like an ordinary component, you can specify properties (a.k.a., attributes) when using a macro component. For example,

```
<?component name="username" macroURI="/WEB-INF/macros/username.zul"?>
<window>
	<username who="John" label="Username"/>
</window>
```

All these properties specified are stored in a map that is then passed to the template (aka., the macro definition; `macroURI`) via a variable called `arg`. Then, from the template, you could access these properties by the use of EL expressions as shown below:

```
<hlayout>
	${arg.label}: <textbox value="${arg.who}"/>
</hlayout>
```

arg.includer

In addition to properties (aka., attributes), a property called `arg.includer` is always passed. It refers the macro component itself. With this, we could reference it to other information such as parent:

```
${arg.includer.parent}
```

Notice that `arg.includer` is different from the so-called inline macros. The inline macros are special macro components and used for inline expansion. For more information please refer to Inline Macros section.

Pass Initial Properties

Sometimes it is helpful to pass a list of initial properties that will be used to initialize a component when it is instantiated. It can be done easily as follows.

```
<?component name="mycomp" macroURI="/macros/mycomp.zul"
   myprop="myval" another="anotherval"?>
```

Therefore,

```
<mycomp/>
```

is equivalent to

```
<mycomp myprop="myval1" another="anotherval"/>
```

Control Macro in Java

Instantiate Macro in Java

To instantiate a macro component in Java, you could do the followings.

1. Looks up the component definition (ComponentDefinition [3]) with the use of boolean) Page.getComponentDefinition(java.lang.String, boolean) [1].
2. Invokes java.lang.String) ComponentDefinition.newInstance(org.zkoss.zk.ui.Page, java.lang.String) [2] to instantiate the component.
3. Invokes Component.setParent(org.zkoss.zk.ui.Component) [3] to attach the macro to a parent, if necessary
4. Invokes Component.applyProperties() [4] to apply the initial properties defined in the component definition.
5. Invokes java.lang.Object) DynamicPropertied.setDynamicProperty(java.lang.String, java.lang.Object) [5] to assign any properties you want.
6. Finally, invokes AfterCompose.afterCompose() [6] to create components defined in the template

For example,

```
HtmlMacroComponent ua = (HtmlMacroComponent)
    page.getComponentDefinition("username", false).newInstance(page,
null);
ua.setParent(wnd);
ua.applyProperties(); //apply properties defined in the component
definition
ua.setDynamicProperty("who", "Joe");
ua.afterCompose(); //then the ZUML page is loaded and child components
are created
```

It is a bit tedious. If you implement you own custom Java class (instead of HtmlMacroComponent [7]), it will be simpler. For example,

```
Username ua = new Username();
ua.setParent(wnd);
ua.setWho("Joe");
```

Please refer to the Implement Custom Java Class section for details.

Change Template at Runtime

You could change the template dynamically by the use of HtmlMacroComponent.setMacroURI(java.lang.String) [8]. For example,

```
<username id="ua"/>
<button onClick="ua.setMacroURI(&amp;quot;another.zul&amp;quot;)"/>
```

If the macro component was instantiated, all of its children will be removed first, and then the new template will be appled (so-called recreation).

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Page.html#getComponentDefinition(java.lang.String,
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/metainfo/ComponentDefinition.html#newInstance(org.zkoss.zk.ui.Page,
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#setParent(org.zkoss.zk.ui.Component)
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#applyProperties()
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/ext/DynamicPropertied.html#setDynamicProperty(java.lang.String,
[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/ext/AfterCompose.html#afterCompose()
[7] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlMacroComponent.html#
[8] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlMacroComponent.html#setMacroURI(java.lang.String)

Inline Macros

Overview

An inline macro is a special macro component which behaves like *inline-expansion*. Unlike a regular macro component, ZK does not create a macro component. Rather, it inline-expands the components defined in the macro URI, as if the content of the in-line macro's template is entered directly into the target page.

Declare an Inline Macro

To declare an inline macro, we have to specify `inline="true"` in the component directive, while the definition and the use of an inline macro is the same as the regular macro components (i.e., non-inline).

For example, suppose we have a macro definition (aka., template) as follows:

```
<!-- username.zul: (macro definition) -->
<row>
      Username
      <textbox id="${arg.id}" value="${arg.name}"/>
</row>
```

We can declare it as an line macro as follows:

```
<!-- target page -->
<?component name="username" inline="true" macroURI="username.zul"?>
<grid>
      <rows>
            <username id="ua" name="John"/>
      </rows>
</grid>
```

Then, it is equivalent to:

```
<grid>
      <rows>
            <row>
                  Username
                  <textbox id="ua" value="John"/>
            </row>
      </rows>
</grid>
```

Notice that all the properties, including `id`, are passed to the inline macro too.

Inline versus Regular Macro

As described above, an inline macro is expanded inline when it is used as if they are entered directly. On the other hand, ZK will create a component (an instance of HtmlMacroComponent [7] or deriving) to represent the regular macro. That is, the macro component is created as the parent of the components that are defined in the template.

Inline macros are easier to integrate into sophisticated pages. For example, you *cannot* use *regular* macro components in the previous example since `rows` accepts only `row` as children, not macro components. It is also easier to access to all components defined in a macro since they are expanded inline. However, it also means that the developers must take care of id themselves. In addition, there is no way to instantiate an inline macros in pure Java (rather, org.zkoss.zk.ui.Component, java.util.Map) Execution.createComponents(java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) [2] shall be used)[1] .

On the other hand, regular macros allow the component developers to provide a custom Java class to represent the component so a better abstraction and addition functionality can be implemented. We will discuss it more in the following section.

---

[1] ZK Loader does create an component for an inline macro when rendering, and then drop it after *expanding* into the parent component. Technically, an application can do the same thing but it is not recommended since we might change it in the future.

arg.includer

Unlike regular macros, arg.includer for a inline macro is the parent component of the macro (after all, the inline macro does not really exist).

An Example

`inline.zul`: (the macro definition)

```
<row>
      <textbox value="${arg.col1}"/>
      <textbox value="${arg.col2}"/>
</row>
```

`useinline.zul`: (the target page)

```
<?component name="myrow" macroURI="inline.zul" inline="true"?>
<window title="Test of inline macros" border="normal">
      <zscript><![CDATA[
            import  org.zkoss.util.Pair;
            List  infos = new LinkedList();
            for(int j = 0; j <10; ++j){
                  infos.add(new Pair("A" + j, "B" +j));
            }
      ]]>
      </zscript>
      <grid>
            <rows>
                  <myrow col1="${each.x}" col2="${each.y}" forEach="${infos}"/>
            </rows>
      </grid>
</window>
```

Version History

Version	Date	Content

Implement Custom Java Class

Overview

As described in the earlier sections, a macro component is instantiated to represent a regular macro. By default, HtmlMacroComponent [7] is assumed (and instantiated). However, you provide a custom Java class to provide a better API to simplify the access and to encapsulate the implementation.

Implement Custom Java Class for Macro

The implementation is straightforward. First, the custom Java class for macro components must extend from HtmlMacroComponent [7]. Second, thought optional, it is suggested to invoke HtmlMacroComponent.compose() [1] in the constructor[2] [3] , such that the template and the wiring of the data members will be applied in the constructor.

For example, suppose we have a macro template as follows.

```
<hlayout>
      Username: <textbox id="mc_who"/>
</hlayout>
```

Then, we could implement a Java class for it:

```
package foo;

import org.zkoss.zk.ui.select.annotation.*;
import org.zkoss.zk.ui.HtmlMacroComponent;
import org.zkoss.zul.Textbox;

@VariableResolver(org.zkoss.zkplus.spring.DelegatingVariableResolver)
public class Username extends HtmlMacroComponent {
    @WireVariable
    private User currentUser; //will be wired if currentUser is a
Spring-managed bean, when compose() is called
    @Wire
    private Textbox mc_who; //will be wired when compose() is called
    public Username() {
        compose(); //fore the template to be applied, and to wire
members automatically
    }
    public String getWho() {
        return mc_who.getValue();
    }
    public void setWho(String who) {
        mc_who.setValue(who);
```

```
    }
    @Listen("onClick=#submit")
    public void submit() { //will be wired when compose() is called.
    }
}
```

As shown, HtmlMacroComponent.compose() [1] will wire variables, components and event listener automatically, so we could access them directly (such as the mc_who member). For more information, please refer to the the Wire Components section, the Wire Variables section and Wire Event Listeners sections.

Also notice that the arg variable is still available to the template so as to represent properties set by java.lang.Object) DynamicPropertied.setDynamicProperty(java.lang.String, java.lang.Object) [5], though it is really useful if a custom implementation is provided.

---

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlMacroComponent.html#compose()

[2] By default, HtmlMacroComponent.compose() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlMacroComponent.html#compose()) is invoked when HtmlMacroComponent.afterCompose() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlMacroComponent.html#afterCompose()) is called. In many cases, it is generally too late, so we suggest to invoke it in the contructor.

[3] HtmlMacroComponent.compose() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlMacroComponent.html#compose()) is available in 5.0.5. For 5.0.4 or earlier, please invoke HtmlMacroComponent.afterCompose() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlMacroComponent.html#afterCompose()) instead.

Declare Macro with Custom Java Class

To make ZK Loader know which custom Java class to use, we have to specify the class attribute when declaring it in the component directives. For example,

```
<?component name="username" macroURI="/WEB-INF/macros/username.zul"
   class="foo.Username"?>
```

Use Macro with Custom Java Class

In ZUML

The use of the macro component with a custom Java class in a ZUML page is the same as other macro components.

In Java

The main purpose of introducing a custom Java class is to simplify the use of a macro component in Java. For example, you could invoke a more meaningful setter, say, setWho, directly rather than java.lang.Object) DynamicPropertied.setDynamicProperty(java.lang.String, java.lang.Object) (http:/ / www. zkoss. org/ javadoc/ latest/zk/org/zkoss/zk/ui/ext/DynamicPropertied.html#setDynamicProperty(java.lang.String,). In addition, the instantiation could be as simple as follows:

```
Username ua = new Username();
ua.setParent(wnd);
ua.setWho("Joe");
```

Macro Component and ID Space

Like Window (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Window.html#), HtmlMacroComponent (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlMacroComponent.html#) also implements IdSpacce (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/IdSpacce.html#). It means that a macro component (excluding inline macros) is a space owner. In other words, it is free to use whatever the identifiers to identify components inside the template.

For example, assume we have a macro defined as follows.

```
<hlayout>
      Username: <textbox id="who" value="${arg.who}"/>
</hlayout>
```

Then, the following codes work correctly.

```
<?component name="username" macroURI="/WEB-INF/macros/username.zul"?>
<zk>
      <username/>
      <button id="who"/> <!-- no conflict because it is in a different ID space -->
</zk>
```

However, the following codes *do not* work.

```
<?component name="username" macroURI="/WEB-INF/macros/username.zul"?>
<username id="who"/>
```

Why? Like any ID space owner, the macro component itself is in the same ID space with its child components. There are two alternative solutions:

1. Use a special prefix for the identifiers of child components of a macro component. For example, `"mc_who"` instead of `"who"`.

```
<hlayout>
      Username: <textbox id="mc_who" value="${arg.who}"/>
</hlayout>
```

2. Use the `window` component to create an additional ID space.

```
<window>
      <hlayout>
            Username: <textbox id="who" value="${arg.who}"/>
      </hlayout>
</window>
```

The first solution is suggested, if applicable, due to the simplicity.

Version History

Version	Date	Content
5.0.5	October, 2010	HtmlMacroComponent.compose() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlMacroComponent.html#compose()) was introduced.

Composite Component

Like a macro component, a composite component is an approach to compose a component based on a template. Unlike a macro component, a composite component has to create and wire the child components by itself, and handle ID space if necessary. The advantage is that a composite component can extend from any component, such as Row [1], such that it is easier to fit to any situation (and no need for the inline concept).

In short, it is suggested to use a macro component if applicable (since it is easier), while using a composite component otherwise.

> If you'd like to assemble UI at runtime (aka., templating), please refer to the Templating section for more information.

Implement a Composite Component

First, you have to decide which component to extend from. Div [2] is a common choice as it is a simple component. However, here our example extends from Row [1], so it can be used under Rows [5], which the regular macros cannot.

Second, you have to implement a template (in a ZUML document) to define what child components the composite component has. Then, you have to implement a Java class to put them together.

Implement a Template

The implementation of a template is straightforward. There is nothing special to handle. Since it is rendered by org.zkoss.zk.ui.Component, java.util.Map) Execution.createComponents(java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) [2], you could pass whatever data you prefer to it (through the arg argument).

Suppose we have a template as follows, and it is placed at /WEB-INF/composite/username.zul.

```
<zk>
  Usename: <textbox id="mc_who"/>
</zk>
```

Implement a Java Class

To implement a Java class we shall:

1. Extend from the component class you want.
2. (Optional) Implement IdSpace [1] to make it an ID space owner.
3. Render the template in the constructor by the use of org.zkoss.zk.ui.Component, java.util.Map) Executions.createComponents(java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) [3] or others.
4. (Optional) Wire variables, components and event listeners after rendering with the use of java.lang.Object, java.util.List) Selectors.wireVariables(org.zkoss.zk.ui.Component, java.lang.Object, java.util.List) [4] (wiring variables), java.lang.Object, boolean) Selectors.wireComponents(org.zkoss.zk.ui.Component, java.lang.Object, boolean) [5] (wiring components) and java.lang.Object) Selectors.wireEventListeners(org.zkoss.zk.ui.Component, java.lang.Object) [6] (wiring event listener).

For example,

```
package foo;

import org.zkoss.zk.ui.IdSpace;
import org.zkoss.zk.ui.select.Selectors;
import org.zkoss.zul.Row;
import org.zkoss.zul.Textbox;

public class Username extends Row implements IdSpace {
    @Wire
    private Textbox mc_who; //will be wired when
Components.wireVariables is called

    public Username() {
        //1. Render the template
        Executions.createComponents("/WEB-INF/composite/username.zul",
this, null);

        //2. Wire variables, components and event listeners (optional)
        Selectors.wireVariables(this, this, null);
        Selectors.wireComponents(this, this, false);
        Selectors.wireEventListeners(this, this);
    }
    public String getWho() {
        return mc_who.getValue();
    }
    public void setWho(String who) {
        mc_who.setValue(who);
    }
    //public void onOK() {..} //Add event listeners if required, and
wired by Components.addForwards
}
```

After org.zkoss.zk.ui.Component, java.util.Map) Executions.createComponents(java.lang.String, org.zkoss.zk.ui.Component, java.util.Map) [3] is called, all components specified in the template will be instantiated

and become the child component of the composite component (Row). Notice that the URI must match the location of the template correctly.

Depending on the implementation you want, you could wire the data members (mc_who) by calling java.lang.Object, boolean) Selectors.wireComponents(org.zkoss.zk.ui.Component, java.lang.Object, boolean) [5]. This method will search all data members and setter methods and *wire* the component with the same ID. Similarly, java.lang.Object) Selectors.wireEventListeners(org.zkoss.zk.ui.Component, java.lang.Object) [6] is used to wire event listeners.

For more information, please refer to the the Wire Components section and Wire Event the Listeners section sections.

> Notice that there is a utility called ZK Composite [7]. With the help of ZK Composite [7], components are created and wired automatically based on the Java annoataions you provide. In other words, Step 3 and 4 are done automatically. For more information, please refer to the Define Components with Java Annotations section.

Wire Spring-managed Beans

java.lang.Object, java.util.List) Selectors.wireVariables(org.zkoss.zk.ui.Component, java.lang.Object, java.util.List) [4] will wire variables that can be resolved by the registered variable resolver. In additions to the variable-resolver directive, you can create any variable resolver manually and pass it as the third argument. java.lang.Class) Selectors.newVariableResolvers(java.lang.Class, java.lang.Class) [8] provides a convenient way to instantiate variable resolvers. For example, let us say we'd like to wire Spring-manage beans, then we can do as follows.

```
@VariableResolver(org.zkoss.zkplus.spring.DelegatingVariableResolver)
public class Username extends Row implements IdSpace {
    @WireVariable
    private User user;

    public Username() {
        Executions.createComponents("/WEB-INF/composite/username.zul",
this, null);

        Selectors.wireVariables(this, this,
            Selectors.newVariableResolvers(getClass(), Row.class));
        Selectors.wireComponents(this, this, false);
        Selectors.wireEventListeners(this, this);
    }
...
```

java.lang.Class) Selectors.newVariableResolvers(java.lang.Class, java.lang.Class) [8] will look for the @VariableResolver annotation and instantiate it automatically. As shown, we annotate DelegatingVariableResolver [9] to resolve Spring managed bean.

For more information, please refer to the Wire Variables section.

ID Space

Unless you extend a component that is an ID space owner (such as Window [1]), all child components specified in the template will be in the same ID space as its parent. It might be convenient at the first glance. However, it will cause the ID conflict if we have multiple instances of the same composite component. Thus, it is generally suggested to make the composite component as a space owner

It can be done easily by implementing an extra interface IdSpace [1]. No other method needs to be implemented.

```
public class Username extends Row implements IdSpace {
...
```

Of course, if you prefer not to have an additional ID space, you don't need to implement an IdSpace [1].

Use Composite Component

Like macros and any other primitive components, you have to declare a composite component before using it. This can be done by using component directives. Then, we could use it the same way (they are actually primitive components). For example,

```
<?component name="username" extends="row" class="foo.Username"?>

<grid>
    <rows>
      <username who="Joe"/>
      <username who="Hellen"/>
    </rows>
</grid>
```

Define Composite Components as Standard Components

If a composite component is used in multiple pages, it is better to define it in the application level, such that it can be accessed in any page without any component directives.

There are basic two approaches to define a component in the application level:

1. Define it in a XML file which is called language addon.
2. Define it with Java annoataions.

Define Components in a Language Addon

A language addon is a XML file providing additional component definitions or customizing the standard components. For example, you can define the username component described in the previous section as follows.

```
<language-addon>
    <addon-name>myapp</addon-name>
    <component>
        <component-name>username</component-name>
        <extends>rows</extends>
        <component-class>foo.Username</component-class>
    </component>
</language-addon>
```

For more information, please refer to Customization: Component Properties.

Define Components with Java Annotations

Instead of maintaining the definitions in the language addon as described above, you can define the component with Java annotation with a utility called ZK Composite [10]. For example,

```
@Composite(name="username", macroURI="/WEB-INF/partial/username.zul")
public class Username extends Rows implements IdSpace {
    @Wire
    private Textbox mc_who; //will be wired when
Components.wireVariables is called

    //Note: no need to create components and wire variables/components

    public String getWho() {
        return mc_who.getValue();
    }
    public void setWho(String who) {
        mc_who.setValue(who);
    }
}
```

This approach is suggested if you have to develop several composite components. As shown, it is more convenient since you don't have to maintain a separated XML file (the language addon). Furthermore, it will create the components and wire them automatically based on the annotations.

Notice that it requires additional JAR files [11], please refer to Small Talks: Define Composite Component using Java Annotation in ZK6 for the details.

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Row.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Div.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#createComponents(java.lang.String,
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/Selectors.html#wireVariables(org.zkoss.zk.ui.Component,
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/Selectors.html#wireComponents(org.zkoss.zk.ui.Component,
[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/Selectors.html#wireEventListeners(org.zkoss.zk.ui.Component,
[7] http://github.com/zanyking/ZK-Composite|
[8] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/Selectors.html#newVariableResolvers(java.lang.Class,
[9] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/spring/DelegatingVariableResolver.html#
[10] https://github.com/zanyking/ZK-Composite
[11] http://github.com/zanyking/ZK-Composite/downloads

Client-side UI Composing

Though optional, you could have the total control of the client's functionality without the assistance of server-side coding. Generally, you don't need to do it. You don't even need to know how ZK Client Engine and client-side widgets communicate with the server. Their states can be synchronized automatically with ZK. However, you can still control this type of synchronization if you want. It is the so-called Server-client fusion.

A good rule of thumb is that you should handle events and manipulate UI mostly, if not all, on the server, since it is more productive. Then, you could improve the responsiveness and visual effects, and/or reduce server loading by handling them at the client, when it is appropriate. Notice that JavaScript is readable by any user, so be careful not to expose sensitive data or business logic when migrating some code from server to client.

- For information about client-side UI composing, please refer to ZK Client-side Reference: UI Composing.
- For information about customizing client-side widget's behavior, please refer to ZK Client-side Reference: Widget Customization.
- For information about client-side markup language (iZUML), please refer to ZK Client-side Reference: iZUML.
- For information about client-side event handling, please refer to ZK Client-side Reference: Event Listening

Version History

Version	Date	Content

Event Handling

An event (Event [1]) is used to abstract an activity made by user, a notification made by an application, and an invocation of server push. Thus, the application can handle different kind of notifications and sources with a universal mechanism. By and large, developers can even use the same approach to handle, say, message queues.

In this section we will discuss how to handle events, such as listening, posting and forwarding.

References

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

Event Listening

There are two ways to listen an event: an event handler and an event listener.

Listen by the use of an Event Handler

An event handler is a method specified as an event attribute of a ZK page or as a member of a component class.

Declare an Event Handler in ZUML

An event handler can be declared in a ZUL page by specifying an event attribute[1] . For example,

```
<button label="hi" onClick='alert("Hello")'/>
```

where the content of the event handler is the code snippet in Java. The event handler will be interpreted at the run time (by use of BeanShell). If you prefer to use other language, you could specify the language name in front of it. For example, the following uses Groovy as the interpreter:

```
<button label="hi" onClick="groovy:alert('Hi, Groovy')"/>
```

Important Builtin Variables

- self - the component receiving the event. In the previous example, it is the button itself.
- event - the event being received. In the previous example, it is an instance of MouseEvent [2].

Notice that the event handler declared in this way is interpreted at the run time, so it inherits all advantages and disadvantage of interpreter-based execution.

Advantange:

- It can be changed on the fly without recompiling and reloading the application.
- Easy to maintain if the code snippet is small.

Disadvantage:

- Slower to run.
- Compilation error can not be known in advance.
- Hard to maintain if mixing business logic with user interface.

Suggestion:

- It is generally suggested to use this approach for 1) prototyping, or 2) simple event handling.

Declare an Event Handler in Java

The other way to have an event listener is to declare it as a member of a component class. For example,

```
public class MyButton extends Button {
    public void onClick() {
        Messagebox.show("Hello");
    }
}
```

If the event handler needs to handle the event, it can declare the event as the argument as follows.

```
public class MyButton extends Button {
    public void onClick(MouseEvent event) {
        Messagebox.show("Hello, "+event.getName());
```

```
    }
}
```

Suggestions:

- It is suggested to use this approach for component development, since it is subtle for application developers to notice its existence. In addition, it requires to extend the component class.

---

[1] An event attribute is an attribute starting with `on`

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

Listen by Use of an Event Listener

Event Listener

An event listener is a class implementing EventListener (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/EventListener.html#). For example,

```
public class MyListener implements EventListener {
    public void onEvent(Event event) {
        Messages.show("Hello");
    }
}
```

Then, you can register an event listener to the component that might receive the event by the use of org.zkoss.zk.ui.event.EventListener) Component.addEventListener(java.lang.String, org.zkoss.zk.ui.event.EventListener) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#addEventListener(java.lang.String,). For example,

```
button.addEventListener("onClick", new MyListener());
```

This is a typical approach to handle events. However, it is a bit tedious to register event listeners one-by-one if there are a lot of listeners. Rather, it is suggested to use a composer as described in the following section.

Composer and Event Listener Autowiring

With ZK Developer's Reference/MVC, you generally do not need to register event listeners manually. Rather, they could be registered automatically by the use of the auto-wiring feature of a composer. For example,

```
public class MyComposer extends SelectorComposer {
    @Listen("onClick = button#hi")
    public void showHi() {
        Messsagebox.show("Hello");
    }
    @Listen("onClick = button#bye")
    public void showBye() {
        Messsagebox.show("Bye");
    }
    @Listen("onOK = window#mywin")
    public void onOK() {
        Messsagebox.show("OK pressed");
    }
```

```
}
```

As shown above, the method to listen an event shall be named by starting with the event name, separating with `$`, and ending with the component's ID. The composer will search all matched methods and register the event listener automatically. Then, in the ZUL page, you can specify the `apply` attribute to associate the composer with a component.

```
<window id="mywin" apply="MyComposer">
    <textbox/>
    <button id="hi"/>
    <button id="bye"/>
</window>
```

If the listener needs to access the event, just declare it as the argument:

```
    @Listen("onClick = button#hi")
    public void showHi(MouseEvent event) {
      Messsagebox.show("Hello, " + event.getName());
    }
```

Though not limited, a composer is usually associated with an ID space (such as Window (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Window.html#)) to handle events and component within the give ID space. You could associate any component that properly represents a scope of your application to manage.

For more information please refer to the Wire Event Listeners section.

Deferrable Event Listeners

By default, events are sent to the server when it is fired at the client. However, many event listeners are just used to maintain the status on the server, rather than providing visual response to the user. In other words, there is no need to send the events for these listeners immediately. Rather, they shall be sent at once at a time to minimize the traffic between the client and the server so as to improve the server's performance. For the sake of the convenience, we call them the deferrable event listeners.

To make an event listener deferrable, you have to implement Deferrable (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Deferrable.html#) (with `EventListener`) and return true for the `isDeferrable` method as follows.

```
public class DeferrableListener implements EventListener, Deferrable {
    private boolean _modified;
    public void onEvent(Event event) {
        _modified = true;
    }
    public boolean isDeferrable() {
        return true;
    }
}
```

When an event is fired at the client (e.g., the user selects a list item), ZK won't send the event if no event listener is registered for it or only deferrable listeners are registered. instead, the event is queued at the client.

On the other hand, if at least one non-deferrable listener is registered, the event will be sent immediately with all queued events to the server at once. No event is lost and the arriving order is preserved.

Tip: Use the deferrable listeners for maintaining the server status, while the non-deferrable listeners for providing the visual responses for the user.

Page-level Event Listener

Developers could add event listeners to a page (Page (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Page. html#)) dynamically by org.zkoss.zk.ui.event.EventListener) Page.addEventListener(java.lang.String, org.zkoss.zk.ui.event.EventListener) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zk/ ui/ Page. html#addEventListener(java.lang.String,). Once added, all events of the specified name sent to any components of the specified page will be sent to the listener.

All event listeners added to a page (so-called page-level event listeners) are assumed to be deferrable, no matter if Deferrable (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Deferrable.html#) is implemented or not.

A typical example is to use a page-level event listener to maintain the modification flag as follows (pseudo code).

```
page.addEventListener("onChange", new EventListener() {
    public void onEvent(Event event) {
        modified = true;
    }
});
```

Precedence of Listeners

The order of precedence for listeners from the highest to the lowest is as follows.

1. Event listeners implemented with Express (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Express.html#), and registered by org.zkoss.zk.ui.event.EventListener) Component.addEventListener(java.lang.String, org.zkoss.zk.ui.event.EventListener) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#addEventListener(java.lang.String,)
2. Event handlers defined in a ZUML document
3. Event listeners registered by org.zkoss.zk.ui.event.EventListener) Component.addEventListener(java.lang.String, org.zkoss.zk.ui.event.EventListener) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#addEventListener(java.lang.String,) (and without Express (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Express.html#))
   - It includes the method of a composer wired by GenericForwardComposer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/GenericForwardComposer.html#) because the event listener is used.
4. Event handlers defined as a class's method
5. Event listeners registered to a page by org.zkoss.zk.ui.event.EventListener) Page.addEventListener(java.lang.String, org.zkoss.zk.ui.event.EventListener) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Page.html#addEventListener(java.lang.String,)

Abort the Invocation Sequence

You could abort the calling sequence by calling Event.stopPropagation() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Event.html#stopPropagation()). Once one of the event listeners invokes this method, all the following event handlers and listeners are ignored.

Version History

Version	Date	Content
5.0.6	November 2010	SerializableEventListener (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/SerializableEventListener.html#) was introduced to simplify the instantiation of a serializable anonymous class

Event Firing

Events are usually fired (aka., triggered) by a component (when serving the user at the client). However, applications are allowed to fire events too.

There are three ways to trigger an event: post, send and echo.

Post an Event

Posting is the most common way to trigger an event. By posting, the event is placed at the end of the system event queue[1] . Events stored in the system event queue are processed one-by-one in first-in-first-out order. Each desktop has one system event queue and all events are handled sequentially.

To trigger an event, you could invoke org.zkoss.zk.ui.Component, java.lang.Object) Events.postEvent(java.lang.String, org.zkoss.zk.ui.Component, java.lang.Object) [2]. For example,

```
Events.postEvent("onClick", button, null); //simulate a click
```

In addition to posting an event to the end of the system event queue, you could specify a priority with java.lang.String, org.zkoss.zk.ui.Component, java.lang.Object) Events.postEvent(int, java.lang.String, org.zkoss.zk.ui.Component, java.lang.Object) [3]. By default, the priority is 0. The higher the priority the earlier an event is processed.

Notice that the invocation returns after placing the event to the system event queue. In other words, the event won't be processed unless all other events posted earlier or with higher priority are processed.

---

[1] Please don't confuse it with the event queues discussed in the event queues section, which are application-specific, while the system event queue is invisible to application developers.

[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Events.html#postEvent(java.lang.String,

[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Events.html#postEvent(int,

Send an Event

If you prefer to trigger an event to a component directly and process it immediately, rather than placing in the system event queue and waiting for execution, you could use org.zkoss.zk.ui.Component, java.lang.Object) Events.sendEvent(java.lang.String, org.zkoss.zk.ui.Component, java.lang.Object) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Events.html#sendEvent(java.lang.String,) to trigger the event.

```
Events.sendEvent("onMyEvent", component, mydata);
```

org.zkoss.zk.ui.Component, java.lang.Object) Events.sendEvent(java.lang.String, org.zkoss.zk.ui.Component, java.lang.Object) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zk/ ui/ event/ Events. html#sendEvent(java. lang. String,) won't return until all handlers and listeners registered for this event has been processed. You could image it as a method of invocation. Also notice that the event handlers and listeners are invoked directly without starting any event threads (no matter whether the event thread is enabled or not[1]).

[1] By default, the event thread is disabled. Please refer to the Event Threads section for more information.

Echo an Event

Echoing is a way to delay event processing until the next AU request (aka., Ajax) is received.

More precisely, the event being echoed won't be queued into the system event queue. Rather, it asks the client to send back an AU request immediately. Furthermore, after the server receives the AU request, the event is then posted to the system event queue for processing.

In other words, the event won't be processed in the current execution. Rather, it is processed in the following request when the event is *echoed* back from the client. Here is an example of using org.zkoss.zk.ui.Component, java.lang.Object) Events.echoEvent(java.lang.String, org.zkoss.zk.ui.Component, java.lang.Object) (http:/ / www. zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Events.html#echoEvent(java.lang.String,):

```
Events.echoEvent("onMyEvent", component, mydata);
```

Event echoing is useful for implementing a long operation. HTTP is a request-and-response protocol, so the user won't receive any feedback until the request has been served and responsed. Thus, we could send back some busy message to let the user know what has happened, and echo back an event to do the long operation. For more information, please refer to the Long Operations: Use Echo Events section.

Version History

Version	Date	Content

Event Forwarding

Overview

For easy programming, ZK does not introduce any complex event flow. When an event is sent to a target component, only the event listeners registered for the target component will be called. It is the application's job to forward an event to another component if necessary.

For example, you might have a menu item and a button to trigger the same action, say, opening a dialog, and then it is more convenient to have a single listener to open the dialog, and register the listener to the main window rather than register to both the menu item and button.

Event Forwarding in Java

Forwarding an event is straightforward: just posting or sending the event again. However, there is a better way: composer. The composer can be the central place to handle the events. For example, you could invoke openDialog in the event handler for the menu item and button as shown below:

```
public class FooComposer extends SelectorComposer {
   @Listen("onClick = menuitem#item1; onClick = button#btn")
   private void openDialog() {
     //whatever you want
   }
}
```

Event Forwarding in ZUML

Event forwarding can be done with the forward attribute in ZUML. For example,

```
<window id="mywin">
      <button label="Save" forward="onSave"/>
      <button label="Cancel" forward="onCancel"/>
</window>
```

Then, `window` will receive the `onSave` event when the Save button is clicked.

With this approach we could introduce an abstract layer between the event and the component. For example, `window` needs only to handle the `onSave` event without knowing which component causes it. Therefore, you could introduce another UI to trigger onSave without modifying the event listener. For example,

```
      <menuitem label="Save" forward="onSave"/>
```

Of course, you can use the composer and ZUML's forward together to have more maintainable code.

```
public class BetterComposer
extends org.zkoss.zk.ui.select.SelectorComposer {
    @Listen(onSave = #mywin)
    public void doSave(ForwardEvent event) { //signature if you care
about event
        ...
    }
```

```
    @Listen(onCancel = #mywin)
    public void doCancel() { //signature if you don't care the event
        ...
```

Notice that, as shown above, the event being forwarded is wrapped as an instance of ForwardEvent [1]. To retrieve the original event, you could invoke ForwardEvent.getOrigin() [2]

Using a component Path

You can also use a component Path [3] within your ZUML pages to specify a target component to which you would like to forward a specific event. This is especially useful if you want to forward events across different IdSpace [4] such as forwarding event from a component in an included ZUML page to the main page component. For example

```
<?page id="mainPage" ?>
<window id="mainWindow" apply="BetterComposer">
...
    <include src="incDetails.zul" />
...
</window>
```

Now in your included page use Path [3] while forwarding event to mainWindow `Window` component

```
   <button forward="//mainPage/mainWindow.onSave" /> <!-- default forward event is onClic
```

You could specify any application-specific data in the forward attribute by surrounding it with the parenthesis as shown below.

```
 <button forward="onCancel(abort)"/><!-- "abort" is passed -->
 <button forward="onPrint(${inf})"/><!-- the object returned by ${inf} is passed -->
```

Then, the application-specific data can be retrieved by use of ForwardEvent.getData() [5].

Notice : When using *forward* attribute in the ZUML(.zul) with ZK MVC control, you have to get the original event by using **getOrigin()**, then you can access the data by **getData()**

- Example : ZUL

```
<tabbox id="ctrl" apply="composer1">
  <tabs>
     <tab id="tb1" label="News" forward="ctrl.onSelectTab(0)"></tab>
     <tab id="tb2" label="News Images" forward="ctrl.onSelectTab(1)"></tab>
  <tabs>
</tabbox>
```

- Example Composer (composer1)

```
@Listen(onSelectTab = #ctrl)
public void doChangeTab(ForwardEvent e) {
      ForwardEvent fe = (ForwardEvent) e.getOrigin();
      System.out.println(fe.getData());
}
```

If you want to forward several events at once, you can specify them in the forward attribute by separating them with the comma (,). For example,

```
<textbox forward="onChanging=onUpdating, onChange=some.onUpdate"/>
```

In addition, the target component and the event data can be specified in EL expressions, while the event names cannot.

Version History

Version	Date	Content

References

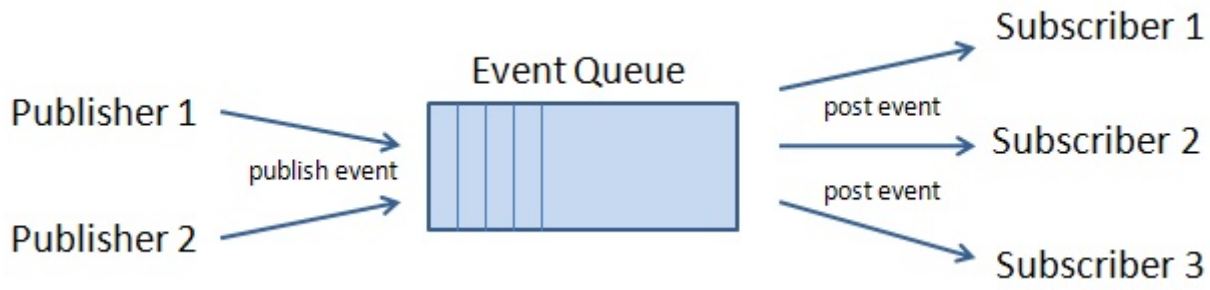
[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/ForwardEvent.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/ForwardEvent.html#getOrigin()
[3] http://books.zkoss.org/wiki/ZK_Developer's_Guide/ZK_in_Depth/Component_Path_and_Accesibility/Access_UI_Component
[4] http://books.zkoss.org/wiki/ZK_Developer's_Reference/UI_Composing/ID_Space
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/ForwardEvent.html#getData()

Event Queues

Overview

An event queue is an event-based publish-subscriber solution for application information delivery and messaging. It provides asynchronous communications for different modules/roles in a loosely-coupled and autonomous fashion.

By publishing, a module (publisher) sends out messages without explicitly specifying or having knowledge of intended recipients. By subscribing, a receiving module (subscriber) receives messages that the subscriber has registered an interest in it, without explicitly specifying or knowing the publisher.



The purpose of event queues are two folds:

1. Simplify the many-to-many communication.
2. Make the application independent of the underlining communication mechanism. The application remains the same, while the event queue can be implemented with the use of Ajax, server push and even message queue.

Identification of an Event Queue

An event queue is identified by a name and a scope. The scope represents the visibility of an event queue. For example, while a desktop-scoped event queue is visible in the same desktop, the application-scoped event queue is only visible in the whole application.

Locate an Event Queue

You could locate an event queue by invoking one of the `lookup` method of EventQueues [1]. For example,

```
EventQueues.lookup("myQueue"); //assumes the desktop scope
EventQueues.lookup("anotherQueue", EventQueues.SESSION);
EventQueues.lookup("anotherQueue", session);
```

Notice that if you want to locate an event queue in a working thread (rather than an event listener), you have to use org.zkoss.zk.ui.Session) EventQueues.lookup(java.lang.String, org.zkoss.zk.ui.Session) [2] or org.zkoss.zk.ui.Application) EventQueues.lookup(java.lang.String, org.zkoss.zk.ui.Application) [2], depending on your requirement.

The Scope of an Event Queue

There are currently four different scopes: desktop, group, session and application. In addition, you add your own scope, such as that you can include a message queue to communicate among several servers.

Name	API	Description
desktop	java.lang.String) EventQueues.lookup(java.lang.String, java.lang.String) [2] java.lang.String, boolean) EventQueues.lookup(java.lang.String, java.lang.String, boolean) [2]	The event queue is visible only in the same desktop.
group	java.lang.String) EventQueues.lookup(java.lang.String, java.lang.String) [2] java.lang.String, boolean) EventQueues.lookup(java.lang.String, java.lang.String, boolean) [2]	`[since 5.0.4][ZK EE]` The event queue is visible only in a group of desktops that belongs to the same browser. It is formed if iframe or frameset is used. Some portal container might cause a group of desktops to be formed too. Unlike the session and application scope, the group scope doesn't require the server push, so the communication is more efficient.
session	java.lang.String) EventQueues.lookup(java.lang.String, java.lang.String) [2] java.lang.String, boolean) EventQueues.lookup(java.lang.String, java.lang.String, boolean) [2] org.zkoss.zk.ui.Session) EventQueues.lookup(java.lang.String, org.zkoss.zk.ui.Session) [2] org.zkoss.zk.ui.Session, boolean) EventQueues.lookup(java.lang.String, org.zkoss.zk.ui.Session, boolean) [2]	The event queue is visible only in the same session. The server push will be enabled automatically if it subscribes a session-scoped event queue. Notice that the server push is disabled automatically if the current desktop doesn't subscribe to any session- or application-scoped event queue. Also notice that the locating and creating of an event queue and publishing an event won't start the server push. ZK 5.0.5 and Prior: When a server push is enabled, a working thread is instantiated and started. It means this feature cannot be used in the environment that doesn't allow working threads, such Google App Engine. No such limitation is likely to occur in ZK 5.0.6 or later.

application	java.lang.String) EventQueues.lookup(java.lang.String, java.lang.String) [2] java.lang.String, boolean) EventQueues.lookup(java.lang.String, java.lang.String, boolean) [2] org.zkoss.zk.ui.WebApp) EventQueues.lookup(java.lang.String, org.zkoss.zk.ui.WebApp) [2] org.zkoss.zk.ui.WebApp, boolean) EventQueues.lookup(java.lang.String, org.zkoss.zk.ui.WebApp, boolean) [2]	The event queue is visible only in the whole application. The server push will be enabled automatically. Notice that the server push is disabled automatically if the current desktop doesn't subscribe to any session- or application-scoped event queue. Also notice that the locating and creating of an event queue and publishing an event won't start the server push. ZK 5.0.5 and Prior: When a server push is enabled, a working thread is instantiated and started. It means this feature cannot be used in the environment that doesn't allow working threads, such Google App Engine. No such limitation is likely to occur in ZK 5.0.6 or later.

Publish and Subscribe

Publish an Event

To publish, just invoke one of the `publish` methods of EventQueue [3] (returned by `lookup`). For example,

```
EventQueues.lookup("my super queue", EventQueues.APPLICATION, true)
  .publish(new Event("onSomethingHapping", null, new
SomeAdditionInfo()));
```

The message used to communicate among publishers and subscribes are the event instances, so you can use any event you prefer.

Subscribe to an Event Queue

The event being published will be sent to each subscriber by calling the event listener the subscriber to subscribe. To subscribe, just invoke one of the `subscribe` methods of EventQueue [3] (returned by `lookup`). For example,

```
EventQueues.lookup("my super queue", EventQueues.APPLICATION,
true).subscribe(
  new EventListener() {
    public void onEvent(Event evt) {
       //handle the event just like any other event listener
    }
  });
```

The event listener is invoked just like a normal event. You can manipulate the component or do whatever as you want.

Example: Chat

Here is an example: chat.

```
<window title="Chat" border="normal">
      <zscript><![CDATA[
      import org.zkoss.zk.ui.event.*;
      EventQueue que = EventQueues.lookup("chat",
EventQueues.APPLICATION, true);
      que.subscribe(new EventListener() {
```

```
            public void onEvent(Event evt) {
                  new Label(evt.getData()).setParent(inf);
            }
      });
      void post(Textbox tb) {
            String text = tb.value;
            if (text.length() > 0) {
                  tb.value = "";
                  que.publish(new Event("onChat", null, text));
            }
      }
      ]]></zscript>

      Say <textbox onOK="post(self)" onChange="post(self)"/>
      <separator bar="true"/>
      <vbox id="inf"/>
</window>
```

Then, you can chat among two or more different computers.

Example: interactive between multiple ZUL pages

It is a typical approach to split page layout with several ZUL pages, and EventQueue could also be used for communicating between these pages. Here is an example for using EventQueue to interactive between two composers.

The first ZUL page and composer class:

```
<window title="ZUL page 1" border="normal" apply="demo.WindowComposer1">
      <button id="btn" label="change label in ZUL page 2" />
</window>
```

```
package demo;
public class WindowComposer1 extends SelectorComposer {

      private EventQueue eq;

      public void doAfterCompose(Component comp) throws Exception {
            super.doAfterCompose(comp);
      }

      @Listen("onClick = button#btn")
      public void changeLabel() {
            eq = EventQueues.lookup("interactive", EventQueues.DESKTOP,
 false);
            eq.publish(new Event("onButtonClick", btn, "label is
Changed!"));
      }
}
```

The second ZUL page and composer class:

```
<window title="ZUL page 2" border="normal" apply="demo.WindowComposer2">
      <label id="lbl" value="label in ZUL page 2" />
</window>
```

```
package demo;
public class WindowComposer2 extends SelectorComposer {

      private EventQueue eq;
      @Wire
      private Label lbl;

      public void doAfterCompose(Component comp) throws Exception {
            super.doAfterCompose(comp);
            eq = EventQueues.lookup("interactive", EventQueues.DESKTOP,
 true);
            eq.subscribe(new EventListener() {
                  public void onEvent(Event event) throws Exception {
                        String value = (String)event.getData();
                        lbl.setValue(value);
                  }
            });
      }
}
```

By doing this, you can change data in ZUL page 2 by clicking the button in ZUL page 1. If there are any other ZUL pages that have subscribed to the same name of the EventQueues, the content in those ZUL pages will also be updated.

Asynchronous Event Listener

By default, the subscribed event listeners are invoked the same way as invocation of the listeners for a posted event. They are invoked one-by-one. No two event listeners belonging to the same desktop will be invoked at the same time. In addition, it is invoked under an execution (i.e., Executions.getCurrent [4] is never null). It is allowed to manipulate the components belonging to the current execution. For sake of description, we call them synchronous event listeners.

On the other hand, the event queue also supports the so-called asynchronous event listener, which is invoked asynchronously in another thread. There is no current execution available. It is *not* allowed to access any component. It is designed to execute a long operation without blocking users from accessing the other functions.

For more information and examples, please refer to the Long Operations: Use Event Queue section.

More About Scopes

Here is a summary of the differences.

	desktop	group	session	application
visibility	desktop	group	session	application
publish	only in an event listener, or the current execution is available.	only in an event listener, or the current execution is available.	no limitation	no limitation
subscribe	only in an event listener, or the current execution is available.	only in an event listener, or the current execution is available.	only in an event listener, or the current execution is available.	only in an event listener, or the current execution is available.
multi-thread	Not used	Not used	5.0.5 or prior: Used (transparent) 5.0.6 or later: Not used	5.0.5 or prior: Used (transparent) 5.0.6 or later: Not used
server-push	Not used	Not used	Used (transparent)	Used (transparent)
Availability	CE	EE	CE	CE

Extend Event Queue: Add a Custom Scope

The location and creation of an event queue is actually done by a so-called event queue provider. An event-queue provider must implement the EventQueueProvider [5] interface.

To customize it, just provide an implementation, and then specify the class name in the property called `org.zkoss.zk.ui.event.EventQueueProvider.class`.

For example, let us say we want to introduce the JMS scope, then we can implement as follows (only pseudo-code):

```
public class MyEventQueueProvider extends
org.zkoss.zk.ui.event.impl.EventQueueProviderImpl {
  public EventQueue lookup(String name, String scope, boolean
autocreate) {
    if ("jms".equals(scope)) {
      //create an event queue based on JMS's name
    } else
      return super.lookup(name, scope, autocreate);
  }
  public boolean remove(String name, String scope) {
    if ("jms".equals(scope)) {
      //remove the event queue based on JMS's name
    } else
      return super.removename, scope);
  }
}
```

Then, specify the property in `WEB-INF/zk.xml`

```
<library-property>
      <name>org.zkoss.zk.ui.event.EventQueueProvider.class</name>
      <value>MyEventQueueProvider</value>
</library-property>
```

Event Queues and Server Push

When an application-scope event queue is used, the server push is enabled for each desktop that subscribers belong to. In additions, a thread is created to forward the event to subscribers.

ZK has two kinds of server push: client-polling and comet[6] . The client-polling server push is implemented with an implicit timer at the client. The interval of the timer depends on the loading of the server. For example, the interval becomes longer if the time to get a response has become longer.

On the other hand, the comet server push is implemented with a pre-established and 'virtual' permanent connection. It is like sending a taxi to the server, and waiting in the server until there is data to send back. Meanwhile, the client-polling server is like sending a taxi periodically to the server, and leave immediately if no data is available.

By default, the comet server push is used. If you prefer to use the client-polling approach, just specify the following in `WEB-INF/zk.xml`[7] .

```
<device-config>
      <device-type>ajax</device-type>
      <server-push-class>org.zkoss.zk.ui.impl.PollingServerPush</server-push-class>
</device-config>
```

---

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/EventQueues.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/EventQueues.html#lookup(java.lang.String,
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/EventQueue.html#
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#getCurrent
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/impl/EventQueueProvider.html#
[6] Comet Programming (http://en.wikipedia.org/wiki/Comet_(programming))
[7] Like most ZK features, you can provide your own implementation if you like.

Version History

Version	Date	Content
5.0.4	August 2010	The group scope was introduced to allow the communication among inline frames without Server Push (minimizing the network bandwidth consumption).
5.0.6	November 2010	The event queue won't start any working threads and they are serializable, so it is safe to use them in a clustering environment.

Client-side Event Listening

Overview

ZK allows applications to handle events at both the server and client side. Handling events at the server side, as described in the previous sections, are more common, since the listeners can access the backend services directly. However, handling event at the client side improves the responsiveness. For example, it is better to be done with a client-side listener if you want to open the drop-down list when a comobox gains focus.

A good rule of thumb is to use server-side listeners first since it is easier, and then improve the responsiveness of the critical part, if any, with the client-side listener.

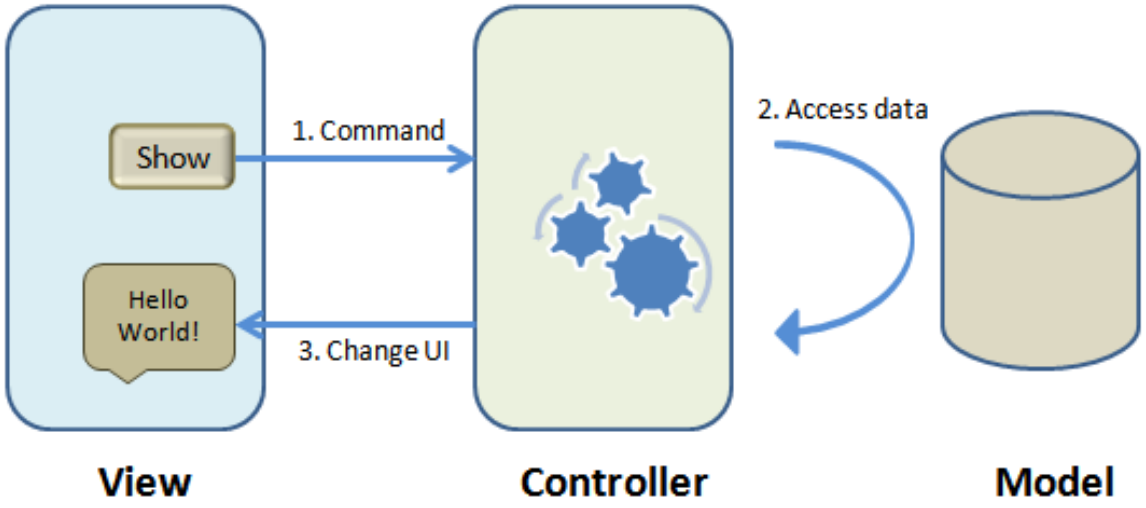
For more information about handling events at the client, please refer to ZK Client-side Reference: Event Listening.

Version History

Version	Date	Content

MVC

MVC (Model-View-Control[1]) is a design pattern designed to separate the model, view and controller. It is strongly suggested to apply MVC pattern to your application, not only because it easy to develop and maintain, but also the performance is great.



Alternative: MVVM

MVVM represents **Model**, **View**, and **ViewModel**[2] . It is a variant of the MVC design pattern. Unlike MVC, the control logic is implemented in a POJO class called the *view model*. It provides the further abstraction that a view model assumes *nothing* about any visual element in the view. It thus avoids mutual programming ripple effects between UI and the view model. On the other hand, some developers might find it not as intuitive as MVC. For more information, please refer to the MVVM section.

[1] More precisely, it is known as MVP (Model-View-Presenter)

[2] MVVM is identical to the Presentation Model (http://martinfowler.com/eaaDev/PresentationModel.html) introduced by Martin Fowler.

View

The *view* is UI -- a composition of components. As described in the UI Composing section, UI can be implemented by a ZUML document or in Java. For sake of description, ZUML is used to illustrate the concept and features.

Controller

The *controller* is a Java program that is used to glue UI (view) and Data (model) together.

For a simple UI, there is no need to prepare a controller. For example, the data of a Listbox (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Listbox.html#) could be abstracted by implementing ListModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModel.html#).

For typical database access, the glue logic (i.e., control) can be handled by a generic feature called Data Binding. In other words, the read and write operations can be handled automatically by a generic Data Binding, and you don't need to write the glue logic at all.

To implement a custom controller, you could extend from SelectorComposer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/SelectorComposer.html#), or implement Composer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Composer.html#) from scratch. Then, specify it in the element it wants to handle in a ZUML document.

Model

The *model* is the data an application handles. Depending on the application requirement, it could be anything as long as your controller knows it. Typical objects are POJOs, beans, Spring-managed beans, and DAO.

In additions to handling the data in a controller, some components supports the abstraction model to uncouple UI and data. For example, grid, listbox and combobox support ListModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModel.html#), while tree supports TreeModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/TreeModel.html#).

Controller

Overview

The *controller* is a Java program that is used to glue UI (view) and Data (model) together.

A simple UI does not require any controllers. For example, the data of a Listbox [1] could be abstracted by implementing ListModel [2] as described in the Model section.

For typical database access, the glue logic (i.e., controller) can be handled by a generic feature called Data Binding. In other words, the create, read, update and delete operations (CRUD) can be handled automatically by a generic Data Binding mechanism, and you don't need to write the glue logic at all as described in the Data Binding section.

If none of above fulfills your requirement, you could implement a custom controller(which is called a composer in ZK terminology). In the following sections we will discuss how to implement a custom controller in details.

References

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

Composer

Custom Controller

A custom controller is called a composer in ZK. To implement it, you could extend from SelectorComposer [3], or implement Composer [2] from scratch. Then, specify it in the UI element that it wants to handle in a ZUML document.

A composer usually does, but not limited to:

- Load data to components, if necessary.
- Handle events and manipulate components accordingly, if necessary.
- Provide the data, if necessary.

In additions, a composer can be used to involve the lifecycle of ZK Loader for doing:

- Exception handling
- Component instantiation monitoring and filtering

A composer can be configured as a system-level composer, such that it will be called each time a ZUML document is loaded.

Implement Composers

Implementing Composer [2] is straightfoward: just override Composer.doAfterCompose(T) [1] and do whatever you want. For example,

```
package foo;
import org.zkoss.zk.ui.util.Composer;
import org.zkoss.zk.ui.event.Events;
import org.zkoss.zk.ui.event.Events;
import org.zkoss.zul.*;
```

```
public class MyComposer implements Composer<Window> {
    public void doAfterCompose(Window main) throws Exception {
        main.addEventListener(Events.ON_OK, new EventListener<KeyEvent>() {
            public void onEvent(KeyEvent event) throws Exception {
                //do whatever you want
            }
        });
    }
}
```

To simplify the implementation, ZK provides several skeleton implementations. For example, SelectorComposer [3], as one of the most popular skeletons, wires components, variables and event listeners automatically based on Java annotations you specify. For example, in the following zul and controller,

ZUL:

```
<window apply="foo.MyComposer">
      <div>
            Input: <textbox id="input" />
      </div>
      <div>
            Output: <label id="output" />
      </div>
      <button id="ok" label="Submit" />
      <button id="cancel" label="Clear" />
</window>
```

Controller:

```
package foo;
import org.zkoss.zk.ui.select.SelectorComposer;
import org.zkoss.zk.ui.select.annotation.Wire;
import org.zkoss.zk.ui.select.annotation.Listen;
import org.zkoss.zul.*;

public class MyComposer extends SelectorComposer<Window> {

      @Wire
      Textbox input;
      @Wire
      Label output;

      @Listen("onClick=#ok")
      public void submit() {
            output.setValue(input.getValue());
      }
      @Listen("onClick=#cancel")
      public void cancel() {
            output.setValue("");
      }
```

```
}
```

the member fields input, output are automatically assigned with components with identifiers of "input" and "output", respectively. The methods submit() and cancel() will be called when user clicks on the corresponding buttons.

In additions to wiring via identifiers, you could wire by a CSS3-like selector (Selector [2]), such as

- @Wire("#foo")
- @Wire("textbox, intbox, decimalbox, datebox")
- @Listen("onClick = button[label='Clear']")
- @Wire("window > div > button")

For more information, please refer to the following sections: Wire Components, Wire Variables and Wire Event Listeners.

Apply Composers

Once a composer is implemented, you could associate it with a component, such that the composer can control the UI components that rooted the given component.

Associating a composer to a component is straightforward: just specify the class to the apply attribute of the XML element you want to control. For example,

```
<grid apply="foo.MyComposer">
    <rows>
        <row>
            <textbox id="input"/>
            <button label="Submit" id="submit"/>
            <button label="Reset" id="reset"/>
        </row>
    </rows>
</grid>
```

If you have to handle the components after ZK Loader initializing them, you could override SelectorComposer.doAfterCompose(T) [3]. It is important to call back super.doAfterCompose(comp). Otherwise, the wiring won't work. It also means that none of the data members are wired before calling `super.doAfterCompose(comp)`.

```
public class MyComposer extends SelectorComposer<Grid> {
   public void doAfterCompose(Grid comp) {
      super.doAfterCompose(comp); //wire variables and event listners
      //do whatever you want (you could access wired variables here)
   }
...
```

where comp is the component that the composer is applied to. In this example, it is the grid. As the name indicates, doAfterCompose is called after the grid and all its descendants are instantiated.

Applying Multiple Composers

You could specify multiple composers; just separate them with comma. They will be called from left to right.

```
<div apply="foo.Composer1, foo2.Composer2">
```

Apply Composer Instances

In additions to the class name, you could specify an instance too. For example, suppose you have an instance called fooComposer, then

```
<grid apply="${fooComposer}">
```

If a class name is specified, each time the component is instantiated, an instance of the specified composer class is instantiated too. Thus, you don't have to worry about the concurrency issue. However, if you specify an instance, it will be used directly. Thus, you have to either create an instance for each request, or make it thread-safe.

Retrieve Composer in EL Expressions

If you have to retrieve the composer back later (such as reference it in an EL expression), you can store the composer into a component's attribute[4] .

If the composer extends from one of ZK skeletal implementations (such as SelectorComposer [3] and GenericForwardComposer [5]), it will be stored into an attribute automatically. Thus, for sake of convenience, you could extend from one of these classes, if you'd like to retrieve the composer back.

Every ZK skeletal implementation provides several ways to name the composer as described in the following sections.

---

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Composer.html#doAfterCompose(T)
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/Selector.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/SelectorComposer.html#doAfterCompose(T)
[4] It can be done by invoking java.lang.Object) Component.setAttribute(java.lang.String, java.lang.Object) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#setAttribute(java.lang.String,), because the component's attribute can be referenced directly in EL expressions. Notice that if you want to reference it in EL expressions, you'd better to set the attribute in ComposerExt.doBeforeComposeChildren(T) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComposerExt.html#doBeforeComposeChildren(T)). Composer.doAfterCompose(T) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Composer.html#doAfterCompose(T)) was called after all child components are instantiated.
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/GenericForwardComposer.html#

Default Names of Composer

If a composer extends from one of ZK skeletal implementations (such as SelectorComposer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/SelectorComposer.html#) and GenericForwardComposer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/GenericForwardComposer.html#)), the composer is stored in three component attributes called `$composer`, *id*`$composer` and *id*`$`*ClassName*, where *id* is the component's ID, and *ClassName* is the class name of the composer. If ID is not assigned, it is default to an empty string, so the composer will be stored to two component attributes: `$composer` and `$`*ClassName*.

For example,

```
<window id="mywin" apply="MyComposer">
     <textbox value="${mywin$composer.title}"/>
     <textbox value="${$composer.title}"/> <!- also refer to MyComposer -->
 </window>
```

Notice that `$composer` is always assigned no matter the ID is, so it is more convenient to use. However, if there are several components assigned with composers, you might have to use ID to distinguish them.

The second name (*id$ClassName*) is useful, If there are multiple composers applied.

```
<window apply="foo.Handle1, foo.Handle2">
    <textbox value="${$Handle1.title}"/>
    <textbox value="${$Handle2.name}"/>
 </window>
```

Specify Name for Composer

If you prefer to name the composer by yourself, you could specify the name in a component attribute called `composerName`. For example,

```
<window apply="MyComposer">
    <custom-attributes composerName="mc"/> <!-- name the composer as mc -->

    <textbox value="${mc.title}"/>
 </window>
```

Notice that once you assign a name to a composer, the default name (*id$composer* and *id$Xxx*) won't be assigned.

Prepare Data for EL Expressions in Composer

It is a common practice to prepare some data in a composer, such that those data are available when rendering the child components. As described above, the composer will be stored as a component attribute that is accessible directly in EL expressions, Thus, you could provide the data easily by declaring a public getter method. For example,

```
public class UsersComposer extends
org.zkoss.zk.ui.select.SelectorComposer<Window> {
    public ListModel<User> getUsers() {
        //return a collection of users
    }
}
```

Then, you could access it as follows.

```
<window title="User List" border="normal" apply="foo.UsersComposer">
    <grid model="${$composer.users}>
...
```

Wire Spring-managed beans

Here is another example that we wire Spring-managed beans with the WireVariable (http:/ / www. zkoss. org/ javadoc/latest/zk/org/zkoss/zk/ui/select/WireVariable.html#) annotation.

```
@VariableResolver(org.zkoss.zkplus.spring.DelegatingVariableResolver.class)
public class UsersComposer extends SelectorComposer<Window> {
    @WireVariable
    private List<User> users;

    public ListModel<User> getUsers() {
        return new ListModelList<User>(users);
```

```
    }
}
```

where we register a variable resolver called DelegatingVariableResolver (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/spring/DelegatingVariableResolver.html#) with the VariableResolver (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/VariableResolver.html#) annotation. As its name suggests, DelegatingVariableResolver (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/spring/DelegatingVariableResolver.html#) will be used to retrieve Spring-managed beans when @WireVariable is encountered. For more information, please refer to the Wire Variables section.

Notice that the variables will be wired before instantiating the component and its children, so it is OK to access them in the ZUML document, as below.

```
<window title="User List" border="normal" apply="foo.UsersComposer">
    <grid model="${$composer.users}>
...
```

Composer with More Control

A composer could also handle the exceptions, if any, control the life cycle of rendering, and intercept how a child component is instantiated. It can be done by implementing the corresponding interfaces, ComposerExt (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComposerExt.html#) and/or FullComposer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/FullComposer.html#).

Exception and Lifecycle Handling with ComposerExt

If you want a composer to handle the exception and/or control the life cycle of rendering, you could also implement ComposerExt (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComposerExt.html#). Since SelectorComposer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/SelectorComposer.html#) already implements this interface, you only need to override the method you care if you extends from it.

For example, we could handle the exception by overriding ComposerExt.doCatch(java.lang.Throwable) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComposerExt.html#doCatch(java.lang.Throwable)) and/or ComposerExt.doFinally() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComposerExt.html#doFinally()).

```
public class MyComposer<T extends Component> extends SelectorComposer<T> {
    public boolean doCatch(Throwable ex) {
        return ignorable(ex); //return true if ex could be ignored
    }
}
```

For involving the life cycle, you could override org.zkoss.zk.ui.Component, org.zkoss.zk.ui.metainfo.ComponentInfo) ComposerExt.doBeforeCompose(org.zkoss.zk.ui.Page, org.zkoss.zk.ui.Component, org.zkoss.zk.ui.metainfo.ComponentInfo) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComposerExt.html#doBeforeCompose(org.zkoss.zk.ui.Page,) and/or ComposerExt.doBeforeComposeChildren(T) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComposerExt.html#doBeforeComposeChildren(T)).

Fine-grained Full Control with FullComposer

In addition to controlling the given component, a composer can monitor the instantiation and exceptions for each child and the descendant component. It is done by implementing FullComposer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/FullComposer.html#). SelectorComposer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/SelectorComposer.html#) does not implement this interface by default. Thus, you have to implement it explicitly.

There is no implementation method needed for this interface. It is like a decorative interface to indicate that it requires the fine-grained full control. In other words, all methods declared in Composer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Composer.html#) and ComposerExt (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComposerExt.html#) will be invoked one-by-one against each child and the descendant component.

For example, suppose we have a composer implementing both Composer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Composer.html#) and FullComposer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/FullComposer.html#), and it is assigned as followed

```
<panel apply="foo.MyComposer">
    <div>
        <datebox/>
        <textbox/>
    </div>
</panel>
```

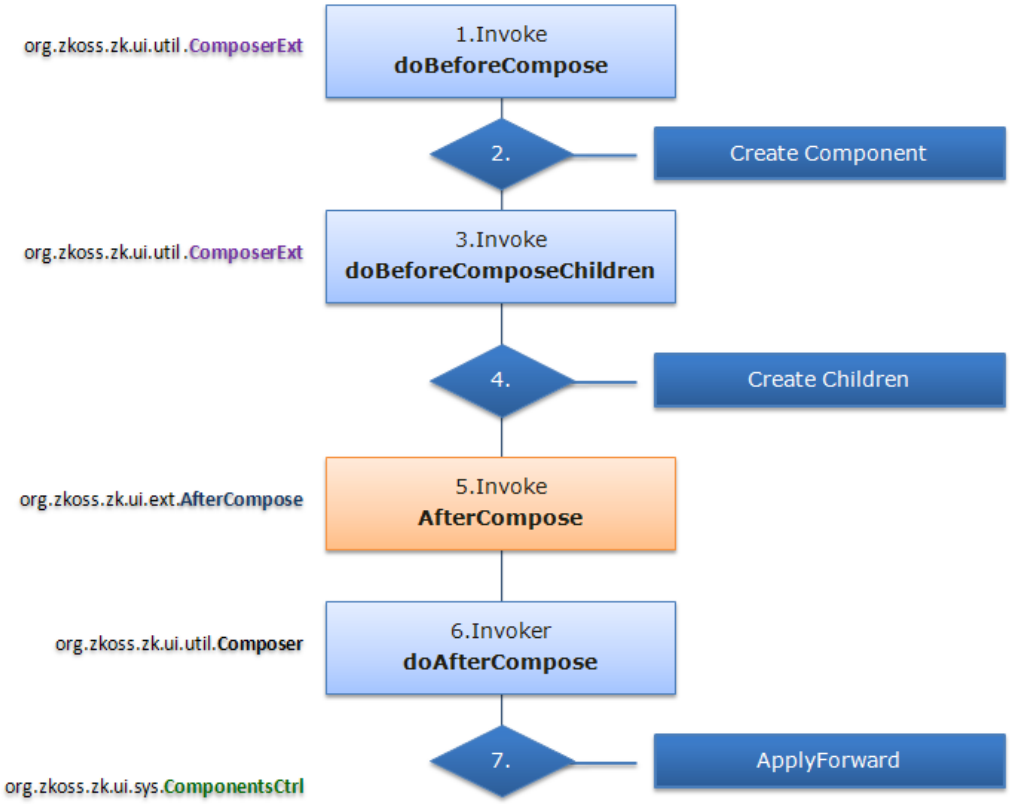
Then, Composer.doAfterCompose(T) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Composer.html#doAfterCompose(T)) will be called for datebox, textbox, div and then panel (in the order of *child-first-parent-last*). If FullComposer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/FullComposer.html#) is not implemented, only the panel will be called.

Notice that, because Composer.doAfterCompose(T) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Composer.html#doAfterCompose(T)) will be called for each child, the generic type is better to Compoonent (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Compoonent.html#) rather than the component's type which the composer is applied to. For example,

```
public class MyFullComposer extends SelectorComposer<Component>, FullComposer {
...
```

Lifecycle

Here is a lifecylce of the invocation of a composer:



System-level Composer

If you have a composer that shall be invoked for every page, you could register a system-level composer rather than specifying it on every page.

It could be done by specifying the composer you implemented in WEB-INF/zk.xml[1] :

```
<listener>
    <listener-class>foo.MyComposer</listener-class>
</listener>
```

Each time a ZK page, including ZK pages and richlets, is created, ZK will instantiate one instance for each registered system-level composer and the invoke Composer.doAfterCompose(T) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Composer.html#doAfterCompose(T)) for each root component. The system-level composer is usually used to process ZK pages after all components are instantiated successfully, such as adding a trademark. If you want to process only certain pages, you can check the request path by calling Desktop.getRequestPath() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Desktop.html#getRequestPath()) (the desktop instance can be found through the given component).

If the system-level composer also implements ComposerExt (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComposerExt.html#), it can be used to handle more situations, such as exceptions, like any other composer can do.

If the system-level composer also implements FullComposer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/FullComposer.html#), it will be invoked when each component is created. It provides the finest grain of control but a wrong implementation might degrade the performance.

Notice that since a new instance of the composer is created for each page, there is no concurrency issues.

[1] For more information, please refer to ZK Configuration Reference

Richlet

A system-level composer can implement ComposerExt (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComposerExt.html#) to handle exceptions for a richlet, such as `doCatch` and `doFinally`. However, `doBeforeCompose` and `doBeforeComposeChildren` won't be called.

FullComposer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/FullComposer.html#) is not applicable to richlets. In other words, system-level composers are called only for root components.

Version History

Version	Date	Content
5.0.8	June, 2011	GenericAutowireComposer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/GenericAutowireComposer.html#) and its derives allow developers to specify a custom name by use of a component attribute called `composerName`.

Wire Components

Wire Components

In SelectorComposer [3], when you specify a @Wire annotation on a field or setter method, the SelectorComposer will automatically find the component and assign it to the field or pass it into the setter method.

You can either give a string value, which is interpret as a **component selector**, as the matching criteria for wiring, or leave it empty to wire by component id. For example,

ZUL:

```
<window apply="foo.MyComposer">
	<textbox />
	<button id="btn" />
</window>
```

Controller:

```
	@Wire("window > textbox")
	Textbox tb; // wire to the first textbox whose parent is a window
	@Wire
	Button btn; // wire to the button with id "btn"
```

CSS3-like Selectors

The string value in @Wire annotation is a **component selector**, which shares an analogous syntax of CSS3 selector. The selector specifies matching criteria against the component tree under the component which applies to this composer.

Given a selector in @Wire annotation, the SelectorComposer will wire a field to the component of **the first match** (in a depth-first-search sense) if the data type of the field is a subtype of Component. Alternatively, if the field type is subtype of Collection, it will wire to an instance of Collection containing all the matched components.

The syntax element of selectors are described as the following:

Type

The component type as in ZUML definition, case insensitive.

```
	@Wire("button")
	Button btn; // wire to the first button.
```

Combinator

Combinator constraints the relative position of components.

```
	@Wire("window button")
	Button btn0; // wire to the first button who has an ancestor
window
	@Wire("window > button") // ">" refers to child
	Button btn1; // wire to the first button whose parent is a window
	@Wire("window + button") // "+" refers to adjacent sibling (next
sibling)
	Button btn2; // wire to the first button whose previous sibling
is a window
	@Wire("window ~ button") // "~" refers to general sibling
	Button btn3; // wire to the first button who has an older sibling
 window
```

You can have any number of levels of combinators:

```
	@Wire("window label + button")
	Button btn4; // wire to the first button whose previous sibling
is a label with an ancestor window.
```

ID

The component id.

```
	@Wire("label#lb")
	Label label; // wire to the first label of id "lb" in the same id
 space of root component
	@Wire("#btn")
	Button btn; // wire to the first component of id "btn", if not a
Button, an exception will be thrown.
```

Unlike CSS3, the id only refer to the component in the same IdSpace of the previous level or root component. For example, given zul

```
<window apply="foo.MyComposer">
	<div>
		<window id="win">
			<div>
				<button id="btn" /><!-- button 1 -->
				<textbox id="tb" /><!-- textbox 1 -->
			</div>
		</window>
		<button id="btn" /><!-- button 2 -->
	</div>
</window>
```

```
	@Wire("#btn")
	Button btnA; // wire to button 2
	@Wire("#win #btn")
	Button btnB; // wire to button 1
	@Wire("#win + #btn")
	Button btnC; // wire to button 2
	@Wire("#tb")
	Textbox tbA; // fails, as there is no textbox of id "tb"
	             // in the id space of the root window (who applies
to the composer).
	@Wire("#win #tb")
	Textbox tbB; // wire to textbox 1
```

Class

The sclass of component. For example,

```
<window apply="foo.MyComposer">
	<div>
		<button /><!-- button 1 -->
	</div>
	<span sclass="myclass">
		<button /><!-- button 2 -->
	</span>
	<div sclass="myclass">
		<button /><!-- button 3 -->
	</div>
</window>
```

```
	@Wire(".myclass button")
	Button btnA; // wire to button 2
	@Wire("div.myclass button")
	Button btnB; // wire to button 3
```

Attribute

The attributes on components, which means the value obtained from calling corresponding getter method on the component.

- Note: [id="myid"] does not restrict id space like #myid does, so they are **not** equivalent.

```
	@Wire("button[label='submit']")
	Button btn; // wire to the first button whose getLabel() call
returns "submit"
```

Pseudo Class

A pseudo class is a custom criterion on a component. There are a few default pseudo classes available:

```
	@Wire("div:root") // matches only the root component
	@Wire("div:first-child") // matches if the component is the first
 child among its siblings
	@Wire("div:last-child") // matches if the component is the last
child among its siblings
	@Wire("div:only-child") // matches if the component is the only
child of its parent
	@Wire("div:empty") // matches if the component has no child
	@Wire("div:nth-child(3)") // matches if the component is the 3rd
child of its parent
	@Wire("div:nth-child(even)") // matches if the component is an
even child of its parent
	@Wire("div:nth-last-child(3)") // matches if the component is the
 last 3rd child of its parent
	@Wire("div:nth-last-child(even)") // matches if the component is
an even child of its parent, counting from the end
```

The nth-child and nth-last-child pseudo classes parameter can also take a pattern, which follows CSS3 specification [1].

Asterisk

Asterisk simply matches anything. It is more useful when working with combinators:

```
	@Wire("*")
	Component rt; // wire to any component first met, which is the
root.
	@Wire("window#win > * > textbox")
	Textbox textbox; // wire to the first grandchild textbox of the
window with id "win"
	@Wire("window#win + * + textbox")
	Textbox textbox; // wire to the second next sibling textbox of
the window with id "win"
```

Multiple Selectors

Multiple selectors separated by comma refers to an OR condition. For example,

```
	@Wire("grid, listbox, tree")
	MeshElement mesh; // wire to the first grid, listbox or tree
component
	@Wire("#win timebox, #win datebox")
	InputElement input; // wire to the first timebox or datebox under
 window with id "win"
```

Wiring by Method

You can either put the @Wire annotation on field or method. In the latter case, it is equivalent to call the method with the matched component as the parameter. This feature allows a more delicate control on handling auto-wires.

```
	@Wire("grid#users")
	private void initUserGrid(Grid grid) {
		// ... your own handling
	}
```

In the example above, the SelectorComposer will find the grid of id "users" and call initUserGrid with the grid as parameter.

- If the method is static or has wrong signature (more than one parameter), an exception will be thrown.
- Wiring by method **requires a selector** on @Wire annotation, otherwise an exception will be thrown.
- If the component is not found, the method is still called, but with null value passed in.
- Do not confuse @Wire with @Listen, while the latter wires to events.

Wiring a Collection

You can also wire **all** matched components to a Collection field or by method, if the field is of Collection type or the method takes a Collection as the parameter.

- If the field starts null or uninitialized or wiring by method, SelectorComposer will try to construct an appropriate instance and assign to the field or pass to method call.
- If the field starts with an instance of Collection already, the collection will be cleared and filled with matched components.
- If it wires by method and the selector matches no components, an empty collection will be passed into the method call.

```
	@Wire("textbox")
	List<Textbox> boxes; // wire to an ArrayList containing all matched
textboxes
	@Wire("button")
	Set<Button> buttons; // wire to a HashSet containing all matched buttons
	@Wire("grid#users row")
	List<Row> rows = new LinkedList<Row>(); // the LinkedList will be filled
with matched row components.
	@Wire("panel")
	public void initPanels(List<Panel> panels) {
		// ...
	}
```

Wiring Sequence

While extending from SelectorComposer [3], the fields and methods with the proper annotations will be wired automatically. Here is the sequence of wiring:

- In Composer.doAfterCompose(T) [1], it wires components to the fields and methods with the Wire [2] annotation.
- Before onCreate event of the component which applies to the composer, the SelectorComposer will attempt to wire the **null fields** and **methods** again, for some of the components might have been generated after doAfterCompose() call.

Performance Tips

The selector utility is implemented by a mixed strategy. In a selector sequence, the first few levels with id specified are handled by Component.getFellow(Component) [3], and the rest are covered by depth first search (DFS). In brief, the more ids you specifies in the **first few levels** of a selector string, the more boost you can obtain in component finding. For example,

```
	@Wire("#win #hl > #btn") // fast, as it is entirely handled by
getFellow()

	@Wire("window hlayout > button") // slower, entirely handled by
DFS

	@Wire("#win hlayout > button") // first level is handled by
getFellow(), other handled by DFS

	@Wire("window #hl > #btn") // slower, as the first level has no
id, all levels are handled by DFS
```

- Note: specifying id via attribute (for instance, [id='myid']) does not lead to the same performance boost.

In the case of multiple selectors, only the first few **identical levels with ids** enjoy the performance gain.

```
	@Wire("#win #hl > button, #win #hl > toolbarbutton")
	// the first two levels have boost

	@Wire("#win #hl > #btn, #win #hl > #toolbtn")
	// the first two levels have boost

	@Wire("#win + #hl > #btn, #win #hl > #btn")
	// only the first level has boost, as they differ in the first
combinator

	@Wire("#win hlayout > #btn, #win hlayout > #toolbtn")
	// only the first level has boost, as the second level has no id
specified
```

In brief, it is recommended to specify id in selector, when you have a large component tree. If possible, you can specify id on all levels, which maximize out the performance gain from the algorithm.

Version History

Version	Date	Content
6.0.0	February 2012	@Wire was introduced.

References

[1] http://www.w3.org/TR/css3-selectors/#nth-child-pseudo
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/annotation/Wire.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#getFellow(Component)

Wire Variables

Wire Variables

SelectorComposer [3] not only wires UI components, but also wires beans from implicit objects and registered variable resolvers.

Wire from Implicit Objects

Wiring from implicit object is equivalent to calling java.lang.String) Components.getImplicit(org.zkoss.zk.ui.Page, java.lang.String) [1], by the name specified on @WireVariable. If the name is absent and the field or method parameter is of type Execution [1], Page [8], Desktop [9], Session [2], or WebApp [3], it still will be wired to the correct implicit object. However, in other cases, an exception will be thrown.

```
public class FooComposer extends SelectComposer<Window> {

	@WireVariable
	private Page _page;

	@WireVariable
	private Desktop _desktop;

	@WireVariable
	private Session _sess;

	@WireVariable
	private WebApp _wapp;

	@WireVariable("desktopScope")
	private Map<String, Object> _desktopScope;

}
```

Wire from Variable Resolver

There are two approaches to register a variable resolver: the VariableResolver [4] annotation or the variable-resolver directive. Here is the example of registering variable resolvers with annotations.

```
@VariableResolver({foo1.MyResolver.class, foo2.AnotherResolver.class})
public class FooComposer extends SelectComposer<Gird> {
....
}
```

To have SelectorComposer [3] to wire a variable, you have to annotate it with the WireVariable [5] annotation. For example,

```
@VariableResolver({foo1.MyResolver.class, foo2.AnotherResolver.class})
public class FooComposer extends SelectComposer<Gird> {
    @WireVariable
    Department department;
    @WireVariable
    public void setManagers(Collection<Manager> managers) {
        //...
    }
}
```

Wire Spring-managed Beans

If you'd like SelectorComposer [3] to wire the Spring-managed beans, you could register the Spring variable resolver, DelegatingVariableResolver [9] with @VariableResolver. Then, you could annotate @WireVariable for wiring a Spring managed bean. For example,

```
@VariableResolver(org.zkoss.zkplus.spring.DelegatingVariableResolver)
public class PasswordSetter extends SelectorComposer<Window> {
    @WireVariable
    private User user;
    @Wire
    private Textbox password; //wired automatically if there is a
textbox named password

    @Listen("onClick=#submit")
    public void submit() {
        user.setPassword(password.getValue());
    }
}
```

DelegatingVariableResolver [9] is a variable resolver used to retrieve the Spring-managed bean, so the variable will be retrieved and instantiated by Spring.

Notice that the variables are wired before instantiating the component and its children, so you can use them in EL expressions. For example, assume we have a composer as follows.

```
@VariableResolver(org.zkoss.zkplus.spring.DelegatingVariableResolver)
public class UsersComposer extends SelectorComposer<Window> {
    @WireVariable
```

```
    private List<User> users;

    public ListModel<User> getUsers() {
        return new ListModelList<User>(users);
    }
}
```

Then, you could reference to getUsers() in the ZUML document. For example,

```
<window apply="UsersComposer">
    <grid model="${$composer.users}">
...
```

where $composer is a built-in variable referring to the composer. For more information, please refer to the Composer section.

---

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Components.html#getImplicit(org.zkoss.zk.ui.Page,
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Session.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/WebApp.html#
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/annotation/VariableResolver.html#
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/annotation/WireVariable.html#

Warning: Not a good idea to have Spring managing the composer

There is a tendency to make the composer as a Spring-managed bean. For example, assume we have a composer called passwordSetter and managed by Spring, then we might do as follows.

```
<?variable-resolver class="org.zkoss.zkplus.spring.DelegatingVariableResolver"?>
<window apply="${passwordSetter}">
...
```

```
@Component
public class PasswordSetter extends SelectorComposer {
   @Autowired User user;
...
```

Unfortunately, this approach is error-prone. The reason is that none of Spring's scopes matches correctly with the lifecycle of the composers. For example, if the Session scope is used, it will cause errors when the user opens two browser windows to visit the same page. In this case, the same composer will be used to serve all desktops in the given session, and it is wrong.

The Prototype scope is a better choice since a new instance is instantiated for each request. However, it also implies another new instance will be instantiated if the Spring variable resolver is called to resolve the same name again in the later requests. It is unlikely, but it might be triggered implicitly and hard to debug. For example, it happens if some of your code evaluates an EL expression that references the composer's name, when an event received.

ZK Spring (http://www.zkoss.org/product/zkspring) is recommended if you want to use Spring intensively. It extends Spring to provide the scopes matching ZK lifecycle, such as the IdSpace and Component scopes. Please refer to ZK Spring Essentials for more detailed information.

Wire CDI-managed Beans

The approach to work with CDI is similar to the approach for Spring, except the variable resolver for CDI is DelegatingVariableResolver (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zkplus/ cdi/ DelegatingVariableResolver.html#).

Wiring Sequence

When extending from SelectorComposer (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zk/ ui/ select/ SelectorComposer. html#), the fields and methods with the proper annotations will be wired automatically. Here is the sequence of wiring:

- In org.zkoss.zk.ui.Component, org.zkoss.zk.ui.metainfo.ComponentInfo) ComposerExt.doBeforeCompose(org.zkoss.zk.ui.Page, org.zkoss.zk.ui.Component, org.zkoss.zk.ui.metainfo.ComponentInfo) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComposerExt.html#doBeforeCompose(org.zkoss.zk.ui.Page,), it wires variables to the fields and methods annotated with the WireVariable (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/annotation/WireVariable.html#) annotation. Here is the sequence how it looks for the variable:
  1. First, it will look for the variable resolver defined in the ZUML document first (by use of Page.addVariableResolver(org.zkoss.xel.VariableResolver) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Page.html#addVariableResolver(org.zkoss.xel.VariableResolver))).
  2. Second, it looks for the variable resolver annotated at the class with the VariableResolver (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/annotation/VariableResolver.html#) annotation.
  3. If none is found, it looks for the implicit objects, such as session and page.

Version History

Version	Date	Content
6.0.0	February 2012	@WireVariable was introduced.

Wire Event Listeners

Wire Event Listeners

To wire an event listener, you need to declare a method with `@Listen` annotation. The method should be public, with return type void, and has either no parameter or one parameter of the specific event type (corresponding to the event listened). The parameter of `@Listen` should be pairs of event name and selector, separated by semicolon.

For example,

```
@Listen("onClick = #btn0")
public void submit(MouseEvent event) {
	// called when onClick is received on the component of id btn0.
}
@Listen("onSelect = #listbox0")
public void select(SelectEvent event) {
	// called when onSelect is received on the component of id
listbox0.
}
```

Event Listener Parameter

There are three ways to declare the method signature of the event listener:

1. No parameter
2. One parameter of the corresponding event type
3. One parameter of a super class of the corresponding event type

For example,

```
@Listen("onChange = textbox#input0")
public void change() {
	// called when onChange is received on the textbox of id input0.
}
@Listen("onChange = textbox#input1")
public void change(InputEvent event) {
	// called when onChange is received on the textbox of id input1.
}
@Listen("onChange = textbox#input2")
public void change(Event event) {
	// called when onChange is received on the textbox of id input2.
}
```

Multiple Targets

If the selector matches multiple components, the event listener will be wired to **all** matched components. In such case, if you need to know which component receives the event, you can retrieve it from `Event#getTarget()`.

For example,

```
@Listen("onClick = grid#myGrid > rows > row")
public void click(MouseEvent event) {
	// called when onClick is received on any Row directly under the
Grid of id myGrid
}
@Listen("onClick = #btn0, #btn1, #btn2")
public void click(MouseEvent event) {
	// called when onClick is received on components of id #btn0,
#btn1, or #btn2
}
```

Multiple Event Types

By separating multiple pairs of event names and selectors by **semicolon**, you can wire different types of event to a single method.

For example,

```
@Listen("onClick = button#submit; onOK = textbox#password")
public void submit(Event event) {
	// called when onClick is received on #submit, or onOK (Enter key
 pressed) is received on #password
}
```

Version History

Version	Date	Content
6.0.0	February 2012	@Listen was introduced.

Model

The *model* is the data an application handles. Depending on the application requirement, it could be anything as long as your controller knows it. Typical objects are POJOs, beans, Spring-managed beans, and DAO. Examples of manipulating the model in the controller was discussed in the previous sections.

In this section and subsections, we will focus on the model that ZK components support directly without custom glue logic. For example, implementing ListModel [2] to control the display of Listbox [1] and Grid [3], and ChartModel [1] to control Chart [2].

In addition to implementing these models, you could use one of the predefined implementation such as SimpleListModel [3] and SimplePieModel [4]. For detailed description, please refer to the following sections.

How to Assign Model to UI

Depending on the requirements, there are a few ways to assign a model to a UI component.

Use Composer to Assign Model

A typical way is to use a composer to assign the model. For example, assume the UI component is a grid and we have a method called `getFooModel` returning the data to show on the grid, then we could implement a composer, say `foo.FooComposer` as follows:

```
public class FooComposer implements Composer {
    public void doAfterCompose(Component comp) throws Exception {
        ((Grid)comp).setModel(getFooModel());
    }
}
```

Then, you could assign it in ZUML as follows:

```
<grid apply="foo.FooComposer">
...
```

Use Databinder

If you are using data binding to handle the database, you could have the data binder to assign the model for you. For example, assume that you have a collection called `persons` (an implementation of `java.util.List`), then:

```
<listbox model="@{persons}">
...
```

Use EL Expressions

EL is another common way to assign the model. For example, assume you have a variable resolver called `foo.FooVariableResolver` implementing VariableResolver [2] as follows.

```
public class FooVariableResolver implements VariableResolver {
    public Object resolveVariable(String name) {
        if ("persons".equals(name)) //found
            return getPersons(); //assume this method returns an
instance of ListModel
        //... you might support more other variables
```

```
        return null; //not found
    }
}
```

Then, you could specify it in ZUML as follows:

```
<?variable-resolver class="foo.FooVariableResolver"?>

<listbox model="${persons">
...
```

The other approach is to use the function mapper. For example, assume you have an implementation called foo.CustomerListModel, then you could use it to drive a listbox as follows.

```
<?taglib uri="http://www.zkoss.org/dsp/web/core" prefix="c" ?>
<listbox model="${c:new('foo.CustomerListModel')}"/>
```

Use zscript

If you are building a prototype, you could use zscript to assign the model directly. For example,

```
<zk>
      <zscript>
      ListModel infos = new ListModelArray(
            new String[][] {
                  {"Apple", "10kg"},
                  {"Orange", "20kg"},
                  {"Mango", "12kg"}
            });
      </zscript>
      <listbox model="${infos}"/>
</zk>
```

Notice that, since the performance of zscript is not good and the mix of Java code in ZUML is not easy to maintain, it is suggested **not** to use this approach in a production system. Please refer to Performance Tips for more information.

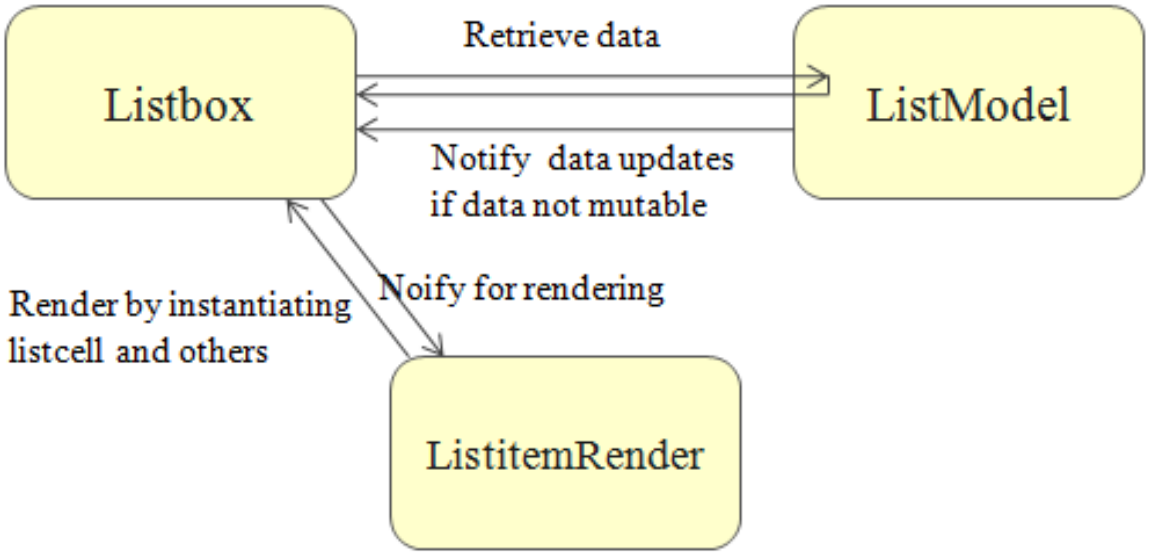
References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ChartModel.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Chart.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/SimpleListModel.html#
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/SimplePieModel.html#

List Model

Listbox [1] and Grid [3] allow developers to separate the view and the model by implementing ListModel [2]. Once the model is assigned (with Listbox.setModel(org.zkoss.zul.ListModel) [1]), the display of the listbox is controlled by the model, and an optional renderer. The model is used to provide data, while the renderer is used to provide the custom look. By default, the data is shown as a single-column grid/listbox. If it is not what you want, please refer to the View section for writing a custom renderer.

Model-driven Display



As shown, the listbox retrieves elements from the specified model[2] , and then invokes the renderer, if specified, to compose the listitem for the element.

The retrieval of elements is done by invoking ListModel.getSize() [3] and ListModel.getElementAt(int) [4].

The listbox will register itself as a data listener to the list model by invoking ListModel.addListDataListener(org.zkoss.zul.event.ListDataListener) [5]. Thus, if the list model is not mutable, the implementation has to notify all the registered data listeners. It is generally suggested to extend from AbstractListModel [6], or use any of the default implementations, which provide a set of utilities for handling data listeners transparently. We will talk about it later in #Notify for Data Updates.

---

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Listbox.html#setModel(org.zkoss.zul.ListModel)

[2] The listbox is smart enough to read the elements that are visible at the client, such the elements for the active page. It is called *Live Data* or *Render on Demand*.'

[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModel.html#getSize()

[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModel.html#getElementAt(int)

[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModel.html#addListDataListener(org.zkoss.zul.event.ListDataListener)

[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/AbstractListModel.html#

Small Amount of Data

If your data can be represented as a list, map, set or array (java.util.List, java.util.Map, etc.), you could use one of the default implementations, such as ListModelList (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModelList.html#), ListModelMap (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModelMap.html#), ListModelSet (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModelSet.html#) and ListModelArray (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModelArray.html#). For example,

```
void setModel(List data) {
    listbox.setModel(new ListModelList(data));
}
```

If the amount of your data is small, you could load them all into a list, map, set or array. Then, you could use one of the default implementations as described above.

Alternatively, you could load all data when ListModel.getSize() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModel.html#getSize()) is called. For example,

```
public class FooModel extends AbstractListModel {
    private List _data;
    public int getSize() {
        //load all data into _data
        return _data.size();
    }
    public Object getElementAt(int index) {
        return _data.get(index);
    }
}
```

Huge Amount of Data

If the data amount is huge, it is not a good idea to load all of them at once. Rather, you shall load only the required subset. On the other hand, it is generally not a good idea to load single elements when ListModel.getElementAt(int) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModel.html#getElementAt(int)) is called, since the overhead loading from the database is significant.

Thus, it is suggested to use SQL LIMIT or similar feature to load only a subset of data. For example, if the total number of visible elements is about 30, you could load 30 (or more, say 60, depending on performance or memory is more important to you). If an element is not loaded, you have to discard the previous loaded data. If any. If the next invocation of ListModel.getElementAt(int) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModel.html#getElementAt(int)) is in the subset, we could return it immediately. Here is the pseudo code:

```
public class FooModel extends AbstractListModel {
    public List _subset;
    public int _startAt;

    public Object getElementAt(int index) {
        if (index >= _startAt && _subset != null && index - _startAt < _subset.size())
            return _subset.get(index - _startAt); //cache hit
        _subset = new LinkedList(); //drop _subset, and load a subset
of data, say, 60, to _subset
```

```
        ...
```

For more realist examples, please refer to Small Talks: Handling huge data using ZK.

Notify for Data Updates

If the data in the model is changed, the implementation must notify all the data listeners that are registered by ListModel.addListDataListener(org.zkoss.zul.event.ListDataListener) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModel.html#addListDataListener(org.zkoss.zul.event.ListDataListener)). It can be done by invoking int, int) AbstractListModel.fireEvent(int, int, int) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/AbstractListModel.html#fireEvent(int,) if your implementation is extended from AbstractListModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/AbstractListModel.html#) or derived.

> Notice that if you use one of the default implementations, such as ListModelList (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModelList.html#), you don't need to worry about it. The notification is handled transparently.

For example, (pseudo code)

```
public void removeRange(int fromIndex, int toIndex) {
    removeElements(fromIndex, toIndex); //remove elements from
fromIndex (inclusive) to toIndex (exclusive)
    fireEvent(ListDataEvent.INTERVAL_REMOVED, fromIndex, index - 1);
}
public void add(int index, Object element){
    addElements(index, element); //add an element at index
    fireEvent(ListDataEvent.INTERVAL_ADDED, index, index);
}
public void set(int index, Object element) {
    setElement(index, element); //change the element at index
    fireEvent(ListDataEvent.CONTENTS_CHANGED, index, index);
}
```

Once a model is assigned to a component, the component will register itself as a data listener such that any changes can be updated to UI.

Notice that you shall not update the component (such as listbox) directly. Rather, you shall update to the modal and then the model shall fire the event to notify the components to update accordingly.

Selection

```
Interface:  TreeSelectableModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/TreeSelectableModel.html#)

Implementation: Implemented by  AbstractListModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/AbstractListModel.html#)
```

If your data model also provides the collection of selected elements, you shall also implement Selectable (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Selectable.html#). When using with a component supporting the selection (such as Listbox (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Listbox.html#)), the component will invoke Selectable.isSelected(E) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Selectable.html#isSelected(E)) to display the selected elements correctly. In additions, if the end user selects or deselects an item, Selectable.addSelection(E) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Selectable.html#addSelection(E)) and Selectable.removeSelection(java.lang.Object) (http://www.zkoss.org/

javadoc/latest/zk/org/zkoss/zul/ext/Selectable.html#removeSelection(java.lang.Object)) will be called by the component to notify the model that the selection is changed. Then, you can update the selection into the persistent layer (such as database) if necessary.

On the other hand, when the model detects the selection is changed (such as Selectable.addSelection(E) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Selectable.html#addSelection(E)) is called), it has to fire the event, such as ListDataEvent.SELECTION_CHANGED (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/event/ListDataEvent.html#SELECTION_CHANGED) to notify the component. It will cause the component to correct the selection<reference>Don't worry. The component is smart enough to prevent the dead loop, even though components invokes addSelection to notify the model while the model fire the event to notify the component.</reference>.

All default implementations, including AbstractListModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/AbstractListModel.html#), implement Selectable (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Selectable.html#). Thus, your implementation generally doesn't need to handle the selection if it extends one of these classes.

It is important to note that, once a listbox is assigned with a list model, the application shall not manipulate the list items and/or change the selection of the listbox directly. Rather, the application shall access only the list model to add, remove and select data elements. Let the model notify the component what have been changed.

Sorting

```
Interface:  Sortable (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Sortable.html#)
Implementation: You have to implement it explicitly
```

To support the sorting, the model must implement Sortable (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Sortable.html#) too. Thus, when the end user clicks the header to request the sorting, boolean) Sortable.sort(java.util.Comparator, boolean) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Sortable.html#sort(java.util.Comparator,) will be called.

For example, (pseudo code)

```
public class FooModel extends AbstractListModel implements Sortable {
    public void sort(Comparator cmpr, final boolean ascending) {
        sortData(cmpr); //sort your data here
        fireEvent(ListDataEvent.CONTENTS_CHANGED, -1, -1); //ask
component to reload all
    }
...
```

Notice that the `ascending` parameter is used only for reference and you usually don't need it, since the `cmpr` is already a comparator capable to sort in the order specified in the `ascending` parameter.

Version History

Version	Date	Content
6.0.0	February 2012	All selection states are maintained in the list model. And, the application shall *not* access the component for the selection. Rather, the application shall invoke Selectable (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Selectable.html#) for retrieving or changing the selection.
6.0.0	February 2012	Sortable (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Sortable.html#) was introduced and replaced `ListModelExt`.

Groups Model

`Available in ZK PE and EE only` [1]

Here we describe how to implement a groups model (GroupsModel [2]). For the concept of component, model and render, please refer to the Model-driven Display section.

A groups model is used to drive components that support groups of data. The groups of data is a two-level tree of data: a list of grouped data and each grouped data is a list of elements to display. Here is a live demo [3]. Currently, both Listbox [1] and Grid [3] support a list of grouped data.

Instead of implementing GroupsModel [2], it is suggested to extend from AbstractGroupsModel [4], or to use one of the default implementations as following:

	SimpleGroupsModel [5]	GroupsModelArray [6]
Usage	The grouping is immutable, i.e., re-grouping is not allowed	Grouping is based on a comparator (java.util.Comparator)
Constructor	The data must be grouped, i.e., data[0] is the first group, data[1] the second, etc.	The data is *not* grouped, i.e., data[0] is the first element. The constructor requires a comparator that will be used to group them.
Version	Since 3.5.0	Since 5.0.5; For 5.0.4 or prior, please use ArrayGroupsModel [7] (the same).

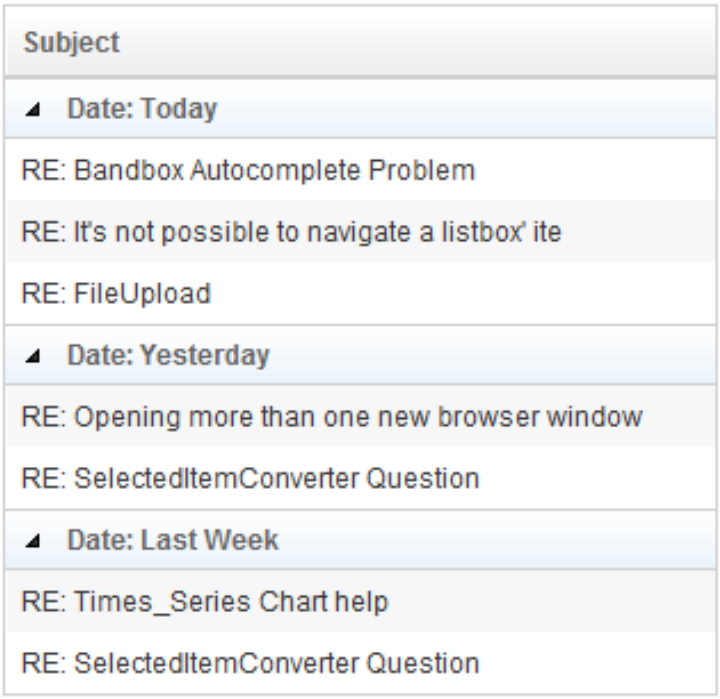
Example: Immutable Grouping Data

If your data is already grouped and the grouping won't be changed, then you could use SimpleGroupsModel [5] as follows:

```
<zk>
    <zscript>
        String[][] datas = new String[][] {
            new String[] { //group 1
                // Today
                "RE: Bandbox Autocomplete Problem",
                "RE: It's not possible to navigate a listbox' ite",
                "RE: FileUpload"
            },
            new String[] { //group 2
                // Yesterday
                "RE: Opening more than one new browser window",
                "RE: SelectedItemConverter Question"
            },
```

```
            new String[] { //group 3
                "RE: Times_Series Chart help",
                "RE: SelectedItemConverter Question"
            }
        };
        GroupsModel model = new SimpleGroupsModel(datas,
            new String[]{"Date: Today", "Date: Yesterday", "Date: Last
Week"});
            //the 2nd argument is a list of group head
    </zscript>
    <grid model="${model}">
        <columns sizable="true">
            <column label="Subject"/>
        </columns>
    </grid>
</zk>
```

Then, the result



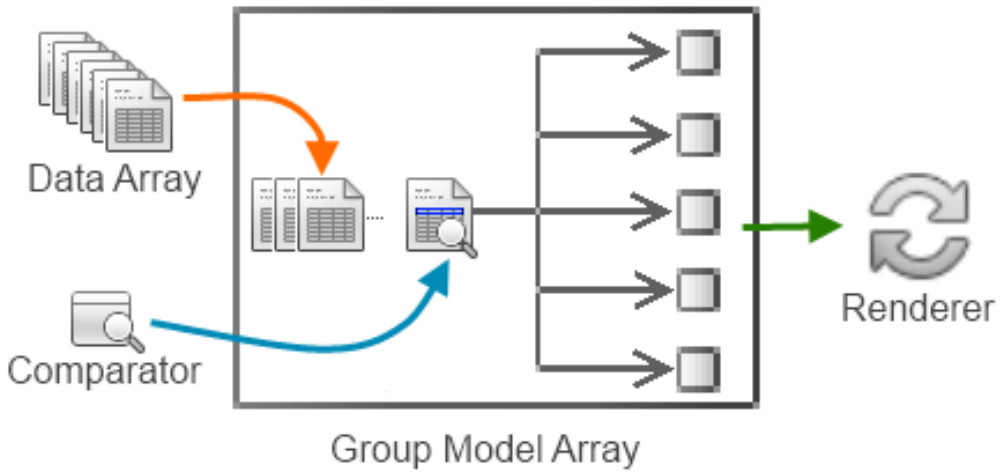
Sorting and Regrouping

If your groups model allows the end user to sort and/or to re-group (i.e., grouping data based on different criteria), you have to implement GroupsModelExt [8] too. Then, boolean, int) GroupsModelExt.group(java.util.Comparator, boolean, int) [9] will be called if the user requests to sort the data based on particular column. And, boolean, int) GroupsModelExt.sort(java.util.Comparator, boolean, int) [10] will be called if the user requests to re-group the data based on particular column.

GroupsModelArray [6] support both sorting and re-grouping as described below:

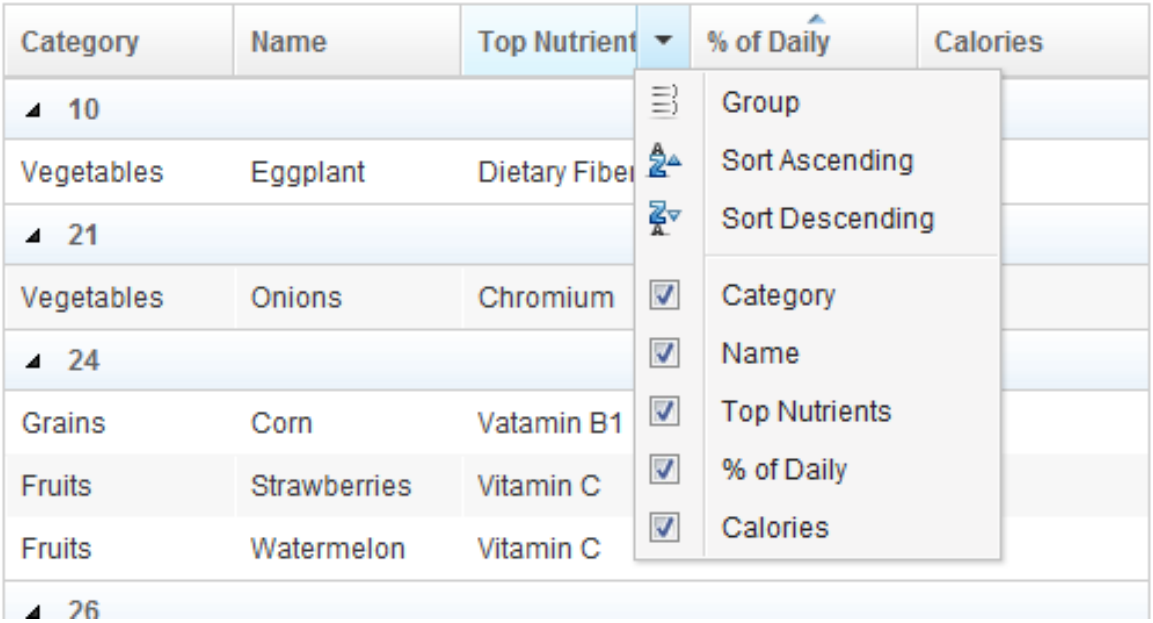
- Sorting: GroupsModelArray [6] sorts each group separately by using the specified comparator (java.util.Comparator).

- Re-grouping: GroupsModelArray [6] re-groups by assuming two data belong to the same group if the compared result is the same (i.e., the given java.util.Comparator returns 0).
  - For better control, you could implement GroupComparator [11], and pass an instance to, say, Column.setSortAscending(java.util.Comparator) [12] and Column.setSortDescending(java.util.Comparator) [13].



Example: Grouping Tabular Data

Suppose you have the data in a two-dimensional array (see below), and you want to allow the user to group them based on a field selected by the user (such as food's name or food's calories).




```
//Data
Object[][] _foods = new Object[][] { //Note: the order does not matter
    new Object[] { "Vegetables", "Asparagus", "Vitamin K", 115, 43},
    new Object[] { "Vegetables", "Beets", "Folate", 33, 74},
    new Object[] { "Vegetables", "Bell peppers", "Vitamin C", 291, 24},
    new Object[] { "Vegetables", "Cauliflower", "Vitamin C", 92, 28},
    new Object[] { "Vegetables", "Eggplant", "Dietary Fiber", 10, 27},
    new Object[] { "Vegetables", "Onions", "Chromium", 21, 60},
```

```
    new Object[] { "Vegetables", "Potatoes", "Vitamin C", 26, 132},
    new Object[] { "Vegetables", "Spinach", "Vitamin K", 1110, 41},
    new Object[] { "Vegetables", "Tomatoes", "Vitamin C", 57, 37},
    new Object[] { "Seafood", "Salmon", "Tryptophan", 103, 261},
    new Object[] { "Seafood", "Shrimp", "Tryptophan", 103, 112},
    new Object[] { "Seafood", "Scallops", "Tryptophan", 81, 151},
    new Object[] { "Seafood", "Cod", "Tryptophan", 90, 119},
    new Object[] { "Fruits", "Apples", "Manganese", 33, 61},
    new Object[] { "Fruits", "Cantaloupe", "Vitamin C", 112, 56},
    new Object[] { "Fruits", "Grapes", "Manganese", 33, 61},
    new Object[] { "Fruits", "Pineapple", "Manganese", 128, 75},
    new Object[] { "Fruits", "Strawberries", "Vitamin C", 24, 48},
    new Object[] { "Fruits", "Watermelon", "Vitamin C", 24, 48},
    new Object[] { "Poultry & Lean Meats", "Beef, lean organic",
"Tryptophan", 112, 240},
    new Object[] { "Poultry & Lean Meats", "Lamb", "Tryptophan", 109,
229},
    new Object[] { "Poultry & Lean Meats", "Chicken", "Tryptophan",
121, 223},
    new Object[] { "Poultry & Lean Meats", "Venison ", "Protein", 69,
179},
    new Object[] { "Grains", "Corn ", "Vatamin B1", 24, 177},
    new Object[] { "Grains", "Oats ", "Manganese", 69, 147},
    new Object[] { "Grains", "Barley ", "Dietary Fiber", 54, 270}
};
```

Then, we can make it a groups model by extending from GroupsModelArray [6]:

```
//GroupsModel
package foo;
public class FoodGroupsModel extends GroupsModelArray {
    public FoodGroupsModel(java.util.Comparator cmpr) {
        super(_foods, cmpr); //assume we
        //cmpr is used to group
    }
    protected Object createGroupHead(Object[] groupdata, int index, int
 col) {
        return ((Object[])groupdata[0])[col];
        //groupdata is one of groups after GroupsModelArray groups
_foods
        ///here we pick the first element and use the col-th column as
the group head
    }
    private static Object[][] _foods = new Object[][] {
        ...tabular data as shown above
    };
};
```

In additions, we have to implement a comparator to group the data based on the given column as follows.

```
package foo;
public class FoodComparator implements java.util.Comparator {
      int _col;
      boolean _asc;
       public FoodComparator(long col, boolean asc) {
            _col = (int) col; //which column to compare
           _asc = asc; //ascending or descending
      }
      public int compare(Object o1, Object o2) {
            Object[] data = (Object[]) o1;
            Object[] data2 = (Object[]) o2;
            int v = ((Comparable)data[_col]).compareTo(data2[_col]);
            return _asc ? v: -v;
      }
}
```

Since the data will be displayed in a multiple column, we have to implement a renderer. Here is an example.

```
public class FoodGroupRenderer implements RowRenderer {
    public void render(Row row, java.lang.Object obj, int index) {
        if (row instanceof Group) {
            //display the group head
            row.appendChild(new Label(obj.toString()));
        } else {
            //display an element
            Object[] data = (Object[]) obj;
            row.appendChild(new Label(data[0].toString()));
            row.appendChild(new Label(data[1].toString()));
            row.appendChild(new Label(data[2].toString()));
            row.appendChild(new Label(data[3].toString()));
            row.appendChild(new Label(data[4].toString()));
        }
    }
};
```

Finally we could group them together in a ZUML document as follows.

```
<?taglib uri="http://www.zkoss.org/dsp/web/core" prefix="c" ?>
<grid rowRenderer="${c:new('foo.FoodGroupRenderer')}"
   model="${c:new1('foo.FoodGroupsModel', c:new2('foo.FoodComparator',
0, true))}">
   <!-- Initially, we group data on 1st column in ascending order -->
    <columns menupopup="auto"> <!-- turn on column's menupopup -->
        <column label="Category"
         sortAscending="${c:new2('foo.FoodComparator', 0, true)}"
         sortDescending="${c:new2('foo.FoodComparator', 0, false)}"
         sortDirection="ascending"/> <!-- since it is initialized as sorted -->
        <column label="Name"
         sortAscending="${c:new2('foo.FoodComparator', 1, true)}"
```

```
            sortDescending="${c:new2('foo.FoodComparator', 1, false)}"/>
        <column label="Top Nutrients"
            sortAscending="${c:new2('foo.FoodComparator', 2, true)}"
            sortDescending="${c:new2('foo.FoodComparator', 2, false)}"/>
        <column label="% of Daily"
            sortAscending="${c:new2('foo.FoodComparator', 3, true)}"
            sortDescending="${c:new2('foo.FoodComparator', 3, false)}"/>
        <column label="Calories"
            sortAscending="${c:new2('foo.FoodComparator', 4, true)}"
            sortDescending="${c:new2('foo.FoodComparator', 4, false)}"/>
    </columns>
</grid>
```

If it is not the behavior you want, you could override java.lang.Object[[14], java.util.Comparator, boolean, int) GroupsModelArray.sortGroupData(java.lang.Object, java.lang.Object[], java.util.Comparator, boolean, int)]. Of course, you could extend from AbstractGroupsModel [4] to have total control.

5.0.6 and Later

Since 5.0.6, it is much easier to handle tabular data:

First, [[15], int, int) GroupsModelArray.createGroupHead(java.lang.Object[], int, int)] will return the correct element, so you don't have to override it as shown above.

Second, ArrayComparator [16] was introduced, so foo.FoodComparator is not required in the above example.

Third, Column.setSort(java.lang.String) [17] supports `auto(0)`, `auto(1)`, etc.

Thus, we can simplify the above example as follows.

```
<grid apply="foo.FoodComposer">
    <columns menupopup="auto"> <!-- turn on column's menupopup -->
        <column label="Category" sort="auto(0)"
            sortDirection="ascending"/> <!-- since it is initialized as sorted -->
        <column label="Name" sort="auto(1)"/>
        <column label="Top Nutrients" sort="auto(2)"/>
        <column label="% of Daily" sort="auto(3)"/>
        <column label="Calories" sort="auto(4)"/>
    </columns>
</grid>
```

And, the composer is as follows.

```
package foo;
import org.zkoss.zk.ui.Component;
import org.zkoss.zk.ui.util.Composer;
import org.zkoss.zul.*;
public class FoodComposer implements Composer {
    public void doAfterCompose(Component comp) throws Exception {
        Grid grid = (Grid)comp;
        grid.setModel(new GroupsModelArray(_foods, new
ArrayComparator(0, true)));
            //Initially, we group data on 1st column in ascending
```

```
order
        grid.setRowRenderer(new FoodGroupRenderer());
    }
}
```

Example: Grouping Array of JavaBean

Suppose you have a collection of JavaBean objects (i.e., with the proper getter methods) as follows.

```
public class Food {
    String _category,  _name, _nutrients;
    int _percentageOfDaily, _calories;

    public Food(String cat, String nm, String nutr, int pod, int cal) {
        _category = cat;
        _name = nm;
        _nutrients = nutr;
        _percentageOfDaily = pod;
        _calories = cal;
    }
    public String getCategory() {
        return _category;
    }
    public String getName() {
        return _name;
    }
    public String getNutrients() {
        return _nutrients;
    }
    public int getPercentageOfDaily() {
        return _percentageOfDaily;
    }
    public int getCalories() {
        return _calories;
    }
}
```

Assume you want to use the value of the field that the user uses to group the data, then you could override GroupModelArray [18] as follows.

```
public class FoodGroupsModel extends GroupsModelArray {
    public FoodGroupsModel(Food[] foods) {
        super(foods, new FieldComparator("category", true));
    }
    protected Object createGroupHead(Object[] groupdata,int index,int
col) {
          return new Object[] {groupdata[0], new Integer(col)};
    }
};
```

where

- We use FieldComparator [19] to initialize the groups at the `category` field.
- We use an object array as the group head that carries the first element of the given group (Food[]), and the index of the column that causes the grouping. We will use the index later to retrieve the field's value

We also need a custom renderer:

```
package foo;
import org.zkoss.lang.reflect.Fields;
import org.zkoss.zk.ui.*;
import org.zkoss.zul.*;
public class FoodGroupRenderer implements RowRenderer {
    public void render(Row row, java.lang.Object obj, int index) {
        if (row instanceof Group) {
            Object[] data = (Object[])obj; //prepared by
createGroupHead()
            row.appendChild(new Label(getGroupHead(row, (Food)data[0],
(Integer)data[1])));
        } else {
            Food food = (Food) obj;
            row.appendChild(new Label(food.getCategory()));
            row.appendChild(new Label(food.getName()));
            row.appendChild(new Label(food.getNutrients()));
            row.appendChild(new Label(food.getPercentageOfDaily() +
""));
            row.appendChild(new Label(food.getCalories() + ""));
        }
    }
    private String getGroupHead(Row row, Food food, int index) {
        Column column =
(Column)row.getGrid().getColumns().getChildren().get(index);
        String orderBy =
((FieldComparator)column.getSortAscending()).getOrderBy();
        int j = orderBy.indexOf("name="),
            k = orderBy.indexOf(' ');
        try {
            return Fields.get(food, orderBy.substring(j+1, k>0 ? k:
orderBy.length())).toString();
        } catch (NoSuchMethodException ex) {
            throw UiException.Aide.wrap(ex);
        }
    }
};
```

The retrieval of the field's value is a bit tricky: since we will use auto(fieldName) to group and sort data for a given column (see the ZUML content listed below), we could retrieve the field's name by use of FieldComparator.getOrderBy() [20], which returns something like "name=category ASC". Then, use java.lang.String) Fields.get(java.lang.Object, java.lang.String) [21] to retrieve it. If the field name is in a compound format, such as `something.yet.another`, you could use java.lang.String) Fields.getByCompound(java.lang.Object,

java.lang.String) [22]

For 5.0.6 or later, you could use FieldComparator.getRawOrderBy() [23] instead, which returns the field name you passed to Column.setSort(java.lang.String) [17], i.e., "category".

```
        Column column =
(Column)row.getGrid().getColumns().getChildren().get(index);
        String field =
((FieldComparator)column.getSortAscending()).getRawOrderBy();
        return Fields.get(food, field).toString();
```

Then, you could have the ZUML document as follows.

```
<grid apply="foo.FoodComposer">
    <columns menupopup="auto">
        <column label="Category" sort="auto(category)" sortDirection="ascending"/>
        <column label="Name" sort="auto(name)"/>
        <column label="Top Nutrients" sort="auto(nutrients)"/>
        <column label="% of Daily" sort="auto(percentageOfDaily)"/>
        <column label="Calories" sort="auto(calories)"/>
    </columns>
</grid>
```

And, the composer is as follows.

```
package foo;
import org.zkoss.zk.ui.Component;
import org.zkoss.zk.ui.util.Composer;
import org.zkoss.zul.*;
public class FoodComposer implements Composer {
    Food[] _foods = new Food[] { //assume we have a collection of foods
        new Food("Vegetables", "Asparagus", "Vitamin K", 115, 43),
        new Food("Vegetables", "Beets", "Folate", 33, 74)
        //...more
    };

    public void doAfterCompose(Component comp) throws Exception {
        Grid grid = (Grid)comp;
        grid.setModel(new FoodGroupsModel(_foods));
             //Initially, we group data on 1st column in ascending order
        grid.setRowRenderer(new FoodGroupRenderer());
    }
}
```

Group Foot

If the groups model supports a foot (such as a summary of all data in the same group), you could return an object to represent the footer when GroupsModel.getGroupfoot(int) [24] is called (similar to GroupsModel.getGroup(int) [25] shall return an object representing the group).

If you use GroupsModelArray [6], you could override [[26], int, int) GroupsModelArray.createGroupFoot(java.lang.Object[], int, int)]. For example,

```
public class FoodGroupsModel extends GroupsModelArray {
    protected Object createGroupFoot(Object[] groupdata, int index, int
 col) {
        return "Total " + groupdata.length + " items";
    }
...
```

Version History

Version	Date	Content
5.0.6	December 2010	Enhanced the support of tabular data as described in #5.0.6 and Later.

References

[1] http://www.zkoss.org/product/edition.dsp
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/GroupsModel.html#
[3] http://www.zkoss.org/zkdemo/grid/grouping
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/AbstractGroupsModel.html#
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/SimpleGroupsModel.html#
[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/GroupsModelArray.html#
[7] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ArrayGroupsModel.html#
[8] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/GroupsModelExt.html#
[9] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/GroupsModelExt.html#group(java.util.Comparator,
[10] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/GroupsModelExt.html#sort(java.util.Comparator,
[11] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/GroupComparator.html#
[12] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Column.html#setSortAscending(java.util.Comparator)
[13] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Column.html#setSortDescending(java.util.Comparator)
[14] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/GroupsModelArray.html#sortGroupData(java.lang.Object,
[15] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/GroupsModelArray.html#createGroupHead(java.lang.Object
[16] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ArrayComparator.html#
[17] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Column.html#setSort(java.lang.String)
[18] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/GroupModelArray.html#
[19] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/FieldComparator.html#
[20] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/FieldComparator.html#getOrderBy()
[21] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/lang/reflect/Fields.html#get(java.lang.Object,
[22] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/lang/reflect/Fields.html#getByCompound(java.lang.Object,
[23] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/FieldComparator.html#getRawOrderBy()
[24] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/GroupsModel.html#getGroupfoot(int)
[25] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/GroupsModel.html#getGroup(int)
[26] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/GroupsModelArray.html#createGroupFoot(java.lang.Object

Tree Model

Here we describe how to implement a tree model (TreeModel [1]). For the concepts of component, model and render, please refer to the Model-driven Display section.

A tree model is used to control how to display a tree-like component, such as Tree [2].

Instead of implementing TreeModel [1] from scratch, it is suggested to extend from AbstractTreeModel [3], which will handle the data listeners transparently, while it allows the maximal flexibility, such as load-on-demand and caching.

In addition, if the tree is small enough to be loaded completely, you could use the default implementation, DefaultTreeModel [4], which uses DefaultTreeNode [5] to construct a tree[6] .

---

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/TreeModel.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Tree.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/AbstractTreeModel.html#
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeModel.html#
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeNode.html#
[6] DefaultTreeModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeModel.html#) was available in 5.0.6. For 5.0.5 or prior, please use SimpleModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/SimpleModel.html#), which is similar except it assumes the tree structure is immutable

Example: Load-on-Demand Tree with AbstractTreeModel

Implementing all TreeModel (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zul/ TreeModel. html#) directly provides the maximal flexibility, such as load-on-demand and caching. For example, you don't have to load a node until int) TreeModel.getChild(java.lang.Object, int) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/TreeModel.html#getChild(java.lang.Object,) is called. In addition, you could load and cache all children of a given node when int) TreeModel.getChild(java.lang.Object, int) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/TreeModel.html#getChild(java.lang.Object,) is called the first time against a particular node, and then return a child directly if it is in the cache.

For example (pseudo code):

```
public class MyModel extends AbstractTreeModel<Object> {
    public Object getChild(Object parent, int index) {
        Object[] children = _cache.get(parent); //assume you have a
cache for children of a given node
        if (children == null)
            children = _cache.loadChildren(parent); //ask cache to load
 all children of a given node
        return children[index];
    }
...
```

By extending from AbstractTreeModel (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zul/ AbstractTreeModel. html#), you have to implement three methods: int) TreeModel.getChild(java.lang.Object, int) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zul/ TreeModel. html#getChild(java. lang. Object,), TreeModel.getChildCount(java.lang.Object) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zul/ TreeModel.html#getChildCount(java.lang.Object)), and TreeModel.isLeaf(java.lang.Object) (http://www.zkoss.org/ javadoc/ latest/ zk/ org/ zkoss/ zul/ TreeModel. html#isLeaf(java. lang. Object)). Optionally, you could implement java.langObject) TreeModel.getIndexOfChild(java.lang.Object, java.langObject) (http:/ / www. zkoss.

org/ javadoc/ latest/ zk/ org/ zkoss/ zul/ TreeModel. html#getIndexOfChild(java. lang. Object,)[1] , if you have a better algorithm than iterating through all children of a given parent.

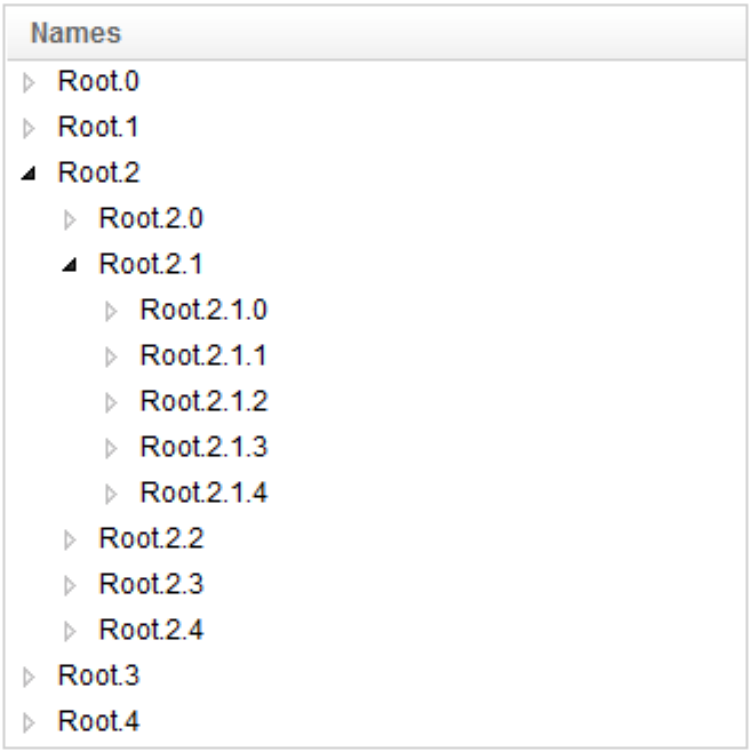
Here is a simple example, which generates a four-level tree and each branch has five children:

```
package foo;
public class FooModel extends AbstractTreeModel<Object> {
    public FooModel() {
        super("Root");
    }
    public boolean isLeaf(Object node) {
        return getLevel((String)node) >= 4; //at most 4 levels
    }
    public Object getChild(Object parent, int index) {
        return parent + "." + index;
    }
    public int getChildCount(Object parent) {
        return isLeaf(parent) ? 0: 5; //each node has 5 children
    }
    public int getIndexOfChild(Object parent, Object child) {
        String data = (String)child;
        int i = data.lastIndexOf('.');
        return Integer.parseInt(data.substring(i + 1));
    }
    private int getLevel(String data) {
        for (int i = -1, level = 0;; ++level)
            if ((i = data.indexOf('.', i + 1)) < 0)
                return level;
    }
};
```

Then, we could have a ZUML document to display it as follows.

```
<?taglib uri="http://www.zkoss.org/dsp/web/core" prefix="c" ?>
<tree model="${c:new('foo.FooModel')}">
    <treecols>
        <treecol label="Names"/>
    </treecols>
</tree>
```

And, the result



[1] java.langObject) TreeModel.getIndexOfChild(java.lang.Object, java.langObject) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/TreeModel.html#getIndexOfChild(java.lang.Object,) is available in 5.0.6 and later.

Example: In-Memory Tree with DefaultTreeModel

```
Since 5.0.6
```

If you prefer to use TreeNode (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/TreeNode.html#) to construct the tree dynamically, you could use DefaultTreeModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeModel.html#) and DefaultTreeNode (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeNode.html#). The use is straightfoward, but it means that the whole tree must be constructed before having it being displayed.

For example, suppose we want to show up a tree of file information, and the file information is stored as FileInfo:

```
package foo;
public class FileInfo {
    public final String path;
    public final String description;
    public FileInfo(String path, String description) {
            this.path = path;
            this.description = description;
    }
}
```

Then, we could create a tree of file information with DefaultTreeModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeModel.html#) as follows.

```
TreeModel model = new DefaultTreeModel(
  new DefaultTreeNode(null,
    new DefaultTreeNode[] {
```

```
      new DefaultTreeNode(new FileInfo("/doc", "Release and License
Notes")),
      new DefaultTreeNode(new FileInfo("/dist", "Distribution"),
        new DefaultTreeNode[] {
          new DefaultTreeNode(new FileInfo("/lib", "ZK Libraries"),
            new DefaultTreeNode[] {
              new DefaultTreeNode(new FileInfo("zcommon.jar", "ZK
Common Library")),
              new DefaultTreeNode(new FileInfo("zk.jar", "ZK Core
Library"))
            }),
          new DefaultTreeNode(new FileInfo("/src", "Source Code")),
          new DefaultTreeNode(new FileInfo("/xsd", "XSD Files"))
        })
      }
  ));
```

To render FileInfo, you have to implement a custom renderer. For example,

```
package foo;
import org.zkoss.zul.*;
public class FileInfoRenderer implements TreeitemRenderer {
    public void render(Treeitem item, Object data, int index) throws
Exception {
        FileInfo fi = (FileInfo)data.getData();
        Treerow tr = new Treerow();
        item.appendChild(tr);
        tr.appendChild(new Treecell(fi.path));
        tr.appendChild(new Treecell(fi.description));
    }
}
```

Then, we could put them together in a ZUML document:

```
<div apply="foo.FileInfoTreeController">
    <tree id="tree">
        <treecols>
            <treecol label="Path"/>
            <treecol label="Description"/>
        </treecols>
    </tree>
</div>
```

where we assume you have a controller, foo.FileInfoTreeController, to bind them together. For example,

```
package foo;
import org.zkoss.zul.Tree;
public class FileInfoTreeController implements
org.zkoss.zk.ui.util.GenericForwardComposer {
    private Tree tree;
```

```
    public void doAfterCompose(org.zkoss.zk.ui.Component comp) {
        tree.setModel(new DefaultTreeModel(..../*as shown above*/));
        tree.setItemRenderer(new FooRenderer());
    }
}
```

Then, the result:

Path	Description
/doc	Release and License Notes
◢ /dist	Distribution
◢ /lib	ZK Libraries
zcommon.jar	ZK Common Library
zk.jar	ZK Core Library
/src	Source Code
/xsd	XSD Files

Notice that you could manipulate the tree dynamically (such as adding a node with DefaultTreeNode.add(org.zkoss.zul.TreeNode) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zul/ DefaultTreeNode. html#add(org. zkoss. zul. TreeNode))). The tree shown at the browser will be modified accordingly.

Example: Create/Update/Delete operation with DefaultTreeNode

```
Since 5.0.6
```

To demonstrate the example, first we add create, update and delete buttons in the ZUML document:

```
<tree id="tree">
...
</tree>
<grid>
    <auxhead>
        <auxheader colspan="2" label="Add/Edit FileInfo" />
    </auxhead>
    <columns visible="false">
        <column />
        <column />
    </columns>
    <rows>
        <row>
            <cell><textbox id="pathTbx" /></cell>
            <cell><textbox id="descriptionTbx" width="300px"/></cell>
        </row>
        <row>
            <cell colspan="2" align="center">
                index: <intbox id="index" /><button id="create" label="Add to selected pa
                <button id="update" label="update" />
                <button id="delete" label="delete" />
```

```
            </cell>
        </row>
    </rows>
</grid>
```

The intbox here is for specifying index to insert before the selected tree item.

Add/Insert

DefaultTreeNode (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeNode.html#) provides DefaultTreeNode.add(org.zkoss.zul.TreeNode) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeNode.html#add(org.zkoss.zul.TreeNode)) and int) DefaultTreeNode.insert(org.zkoss.zul.TreeNode, int) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeNode.html#insert(org.zkoss.zul.TreeNode,) that can manipulate the tree dynamically.

Here we register onClick event to create Button in foo.FileInfoTreeController:

```
//wire component as member fields
private Textbox pathTbx;
private Textbox descriptionTbx;
private Intbox index;
//register onClick event for creating new object into tree model
public void onClick$create() {
    String path = pathTbx.getValue();
    String description = descriptionTbx.getValue();
    if ("".equals(path)) {
        alert("no new content to add");
    } else {
        Treeitem selectedTreeItem = tree.getSelectedItem();
        DefaultTreeNode newNode = new DefaultTreeNode(new
FileInfo(path, description));
        DefaultTreeNode selectedTreeNode = null;
        Integer i = index.getValue();
        // if no treeitem is selected, append child to root
        if (selectedTreeItem == null) {
            selectedTreeNode = (DefaultTreeNode) ((DefaultTreeModel)
tree.getModel()).getRoot();
            if (i == null) // if no index specified, append to last.
                selectedTreeNode.add(newNode);
            else // if index specified, insert before the index number.
                selectedTreeNode.insert(newNode, i);
        } else {
            selectedTreeNode = (DefaultTreeNode)
selectedTreeItem.getValue();

            if (selectedTreeNode.isLeaf())
                selectedTreeNode = selectedTreeNode.getParent();

            if (i == null)
                selectedTreeNode.add(newNode);
```

```
            else
                selectedTreeNode.insert(newNode, i);
        }
    }
}
```

If index is not specified, we add a new node using DefaultTreeNode.add(org.zkoss.zul.TreeNode) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeNode.html#add(org.zkoss.zul.TreeNode)) at the bottom of the parent node by default, or we can also use int) DefaultTreeNode.insert(org.zkoss.zul.TreeNode, int) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeNode.html#insert(org.zkoss.zul.TreeNode,) to insert a new node before the specified index.

Update/Delete

DefaultTreeNode (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeNode.html#) provides DefaultTreeNode.setData(java.lang.Object) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeNode.html#setData(java.lang.Object)) which can update selected tree items and DefaultTreeNode.removeFromParent() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeNode.html#removeFromParent()) that can delete the selected tree item from its parent node.

Here we register onClick event to update and delete Button in foo.FileInfoTreeController:

```
//register onClick event for updating edited data in tree model
public void onClick$update() {
    Treeitem selectedTreeItem = treeGrid.getSelectedItem();
    if(selectedTreeItem == null) {
        alert("select one item to update");
    } else {
        DefaultTreeNode selectedTreeNode = (DefaultTreeNode)
selectedTreeItem.getValue();
        //get current FileInfo from selected tree node
        FileInfo fileInfo = (FileInfo) selectedTreeNode.getData();
        //set new value of current FileInfo
        fileInfo.setPath(pathTbx.getValue());
        fileInfo.setDescription(descriptionTbx.getValue());
        //set current FileInfo in the selected tree node
        selectedTreeNode.setData(fileInfo);
    }
}

//register onClick event for removing data in tree model
public void onClick$delete() {
    final Treeitem selectedTreeItem = treeGrid.getSelectedItem();
    if(selectedTreeItem == null) {
        alert("select one item to delete");
    } else {
        DefaultTreeNode selectedTreeNode = (DefaultTreeNode)
selectedTreeItem.getValue();
        selectedTreeNode.removeFromParent();
    }
```

```
}
```

For updating tree node data, we have to modify foo.FileInfoRenderer:

```
DefaultTreeNode treeNode = (DefaultTreeNode) data;
//for treeItem.getValue() in Controller class
item.setValue(treeNode);
//for update treeNode data
Treerow tr = item.getTreerow();
if(tr == null) {
    tr = new Treerow();
} else {
    tr.getChildren().clear();
}
//renderer...
```

Sorting

```
Interface:  Sortable (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Sortable.html#)
Implementation: You have to implement it explicitly
```

To support the sorting, the model must implement Sortable (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Sortable.html#) too. Thus, when the end user clicks the header to request the sorting, boolean) Sortable.sort(java.util.Comparator, boolean) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Sortable.html#sort(java.util.Comparator,) will be called.

For example, (pseudo code)

```
public class FooModel extends AbstractTreeModel implements Sortable {
    public void sort(Comparator cmpr, final boolean ascending) {
        sortData(cmpr); //sort your data here
        fireEvent(ListDataEvent.CONTENTS_CHANGED, -1, -1); //ask
component to reload all
    }
...
```

Notice that the `ascending` parameter is used only for reference and you usually don't need it, since the `cmpr` is already a comparator capable to sort in the order specified in the `ascending` parameter.

Selection

```
Interface:  TreeSelectableModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/TreeSelectableModel.html#)
Implementation: Implemented by  AbstractTreeModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/AbstractTreeModel.html#)
```

If your data model also provides the collection of selected elements, you shall also implement TreeSelectableModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/TreeSelectableModel.html#). When using with a component supporting the selection (such as Tree (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Tree.html#)), the component will invoke [(http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/TreeSelectableModel.html#isPathSelected(int)) TreeSelectableModel.isPathSelected(int[])] to display the selected elements correctly. In additions, if the end user selects or deselects an item, [(http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/TreeSelectableModel.html#addSelectionPath(int))

TreeSelectableModel.addSelectionPath(int[])] and [(http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/TreeSelectableModel.html#removeSelectionPath(int)) TreeSelectableModel.removeSelectionPath(int[])] will be called by the component to notify the model that the selection is changed. Then, you can update the selection into the persistent layer (such as database) if necessary.

On the other hand, when the model detects the selection is changed (such as [(http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/TreeSelectableModel.html#addSelectionPath(int)) TreeSelectableModel.addSelectionPath(int[])] is called), it has to fire the event, such as TreeDataEvent.SELECTION_CHANGED (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/event/TreeDataEvent.html#SELECTION_CHANGED) to notify the component. It will cause the component to correct the selection<reference>Don't worry. The component is smart enough to prevent the dead loop, even though components invokes addSelectionPath to notify the model while the model fire the event to notify the component.</reference>.

All default implementations, including AbstractTreeModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/AbstractTreeModel.html#) and DefaultTreeModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeModel.html#) implements TreeSelectableModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/TreeSelectableModel.html#). Thus, your implementation generally doesn't have to implement it explicitly.

It is important to note that, once a tree is assigned with a tree model, the application shall not manipulate the tree items and/or change the selection of the tree directly. Rather, the application shall access only the list model to add, remove and select data elements. Let the model notify the component what have been changed.

Open Tree Nodes

```
Interface:  TreeOpenableModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/TreeOpenableModel.html#)

Implementation: Implemented by  AbstractTreeModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/AbstractTreeModel.html#)
```

By default, all tree nodes are closed. To control whether to open a tree node, you could implement TreeOpenableModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/TreeOpenableModel.html#).
More importantly, to open a tree node, the application shall access the model's TreeOpenableModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/TreeOpenableModel.html#) API, rather than accessing Treeitem (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Treeitem.html#) directly.

All default implementations, including AbstractTreeModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/AbstractTreeModel.html#) and DefaultTreeModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeModel.html#) implements TreeOpenableModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/TreeOpenableModel.html#). Thus, your implementation generally doesn't have to implement it explicitly.

Version History

Version	Date	Content
5.0.6	January 2011	TreeNode (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/TreeNode.html#), DefaultTreeNode (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeNode.html#) and DefaultTreeModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeModel.html#) were intrdocued.
6.0.0	February 2012	TreeSelectableModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/TreeSelectableModel.html#) and TreeOpenableModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/TreeOpenableModel.html#) were introduced to replace Selectable (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Selectable.html#) and Openable (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Openable.html#).

Chart Model

Here we describe how to implement a chart model (ChartModel [1]). For the concept of component, model and render, please refer to the Model-driven Display section.

Depending on the type of the chart you want, you could implement one of PieModel [1], XYModel [2], GanttModel [3], HiLoModel [4], etc. In addition, there are the default implementations for them you could use directly, such as SimplePieModel [4], SimpleXYModel [5], etc.

For example, we could have a composer as follows.

```
public class ProgrammerModel implements Composer {
    public void doAfterCompose(Component comp) throws Exception {
        PieModel piemodel = new SimplePieModel();
        piemodel.setValue("C/C++", new Double(12.5));
        piemodel.setValue("Java", new Double(50.2));
        piemodel.setValue("VB", new Double(20.5));
        piemodel.setValue("PHP", new Double(15.5));
        ((Chart)comp).setModel(piemodel);
    }
}
```

Then, you could use it in a ZUML document:

```
<chart title="Pie Chart" width="500" height="250" type="pie" threeD="false" fgAlpha="128"
 apply="foo.ProgrammerModel"/>
```

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/PieModel.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/XYModel.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/GanttModel.html#
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/HiLoModel.html#
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/SimpleXYModel.html#

View

The view is the UI of an application. It totally depends on the application's requirements.

As described in the Model section, many ZK components could operate based on models, such as Listbox [1]. There are two approaches to customize the rendering of each item in model: Template and Renderer.

A template is a fragment of the ZUML document that defines how to render each item in ZUML. On the other hand, a renderer is a Java class that renders each item in Java.

Template

A template is a ZUML fragment that defines how to create components. A template is enclosed with the template element as shown below.

```
<window>
    <template name="foo">
      <textbox/>
      <grid model=${data}>
         <columns/>
         <template name="model"> <!-- nested template -->
            <row>Name: <textbox value=${each.name}"/></row>
         </template>
      </grid>
   </template>
...
```

A template can contain any ZUML elements you want, including other templates. When a ZUML document is interpreted, a template won't be interpreted immediately. Rather, it will be encapsulated as an instance of Template [1], and be associated to a component. Then, the component or a tool can create the components repeatedly based on the template by invoking org.zkoss.zk.ui.Component, org.zkoss.xel.VariableResolver, org.zkoss.zk.ui.util.Composer) Template.create(org.zkoss.zk.ui.Component, org.zkoss.zk.ui.Component, org.zkoss.xel.VariableResolver, org.zkoss.zk.ui.util.Composer) [2].

A component can be assigned with multiple templates. Each of them is identified by the **name** attribute.

```
<div>
    <template name="t1">
        <grid model="${foo}/>
    </template>
    <template name="t2">
        <listbox model="${foo}/>
    </template>
```

How a template is used depends on the component it associates with and the tools you use. Currently, all components that support the concept of model allow you to specify a template to control how to render each item. In the following sections, we discuss them in details. If you'd like to know how to use templates manually in Java, please refer to the UI Patterns: Templates section.

Notice that please read the Listbox Template section first, even though you're rendering other kind of UI. It described the common concepts and tricks of using templates.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Template.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Template.html#create(org.zkoss.zk.ui.Component,

Listbox Template

The template used to control the rendering of each item must be named model and declared right inside the listbox element. For example,

```
<listbox model="${$composer.fruits}" apply="foo.FruitProvider">
      <listhead>
            <listheader label="Name" sort="auto"/>
            <listheader label="Weight" sort="auto"/>
      </listhead>
      <template name="model">
            <listitem>
                  <listcell label="${each[0]}"/>
                  <listcell label="${each[1]}"/>
            </listitem>
      </template>
</listbox>
```

The template's name is important because users are allowed to associate multiple templates to one component, and listbox's default renderer looks only for the template called model.

When the template is rendered, a variable called each is assigned with the data being rendered. Thus, you could retrieve the information to render with EL expressions, such as ${each[0]}, if it is an array, or ${each.name}, if it is a bean with a getter called name.

In this example, we assume the $composer.fruits expression returns a two-dimensional array[1] , and is provided by the foo.FruitProvider composer such as follows[2] .

```
public class FruitProvider extends
org.zkoss.zk.ui.select.SelectorComposer {
    public String[][] fruits = new ListModelArray(
            new String[][] {
                {"Apple", "10kg"},
                {"Orange", "20kg"},
                {"Mango", "12kg"}
            });

    public String[][] getFruits() {
         return fruits;
    }
}
```

Apple	10kg
Orange	20kg
Mango	12kg

[1] Of course, it can be anything you like. Just make sure it matches the EL expressions specified in the template.

[2] Here we use SelectorComposer (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/SelectorComposer.html#) for simplicity. There are several ways to implement a composer, such as wiring a Spring-managed bean. For more information, please refer to the Composer section

Component's Value

By default, the data used to render a component will be stored to the component's value property automatically. For listitem, it is Listitem.setValue(T) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zul/ Listitem. html#setValue(T)). Thus, you retrieve it back easily by invoking Listitem.getValue() (http:/ / www. zkoss. org/ javadoc/latest/zk/org/zkoss/zul/Listitem.html#getValue()).

Of course, if you prefer to store other values, you can simply specify value="${whatever}" to the listitem element in the template.

The forEachStatus Variable

There is a variable called forEachStatus providing the information of the iteration. It is an instance of ForEachStatus (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ForEachStatus.html#). For example, you could retrieve the iteration's index by use of ${forEachStatus.index}.

Lifecycle and the arg Variable

When using the template, it is important to remember that the template is rendered on demand. It means the template can be rendered very late, after the page is rendered, after the user scrolls down to make an item visible, and so on. Thus, in the template, you *cannot* reference anything that is available only in the page rendering phase. For example, you can't reference the arg variable in a template:

```
<listbox model="${$composer.fruits}" apply="foo.FruitProvider">
    <template name="model">
        <listitem>
            <listcell label="${arg.foo}"/> <!-- Wrong! it is always empty -->
            <listcell label="${each}"/>
        </listitem>
    </template>
</listbox>
```

To work around, you have to store the value in, say, component's custom attributes (Component.getAttributes() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Component.html#getAttributes()). For example,

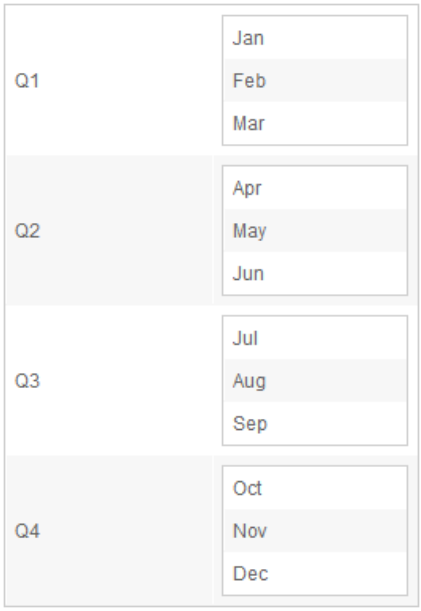
```
<listbox model="${$composer.fruits}" apply="foo.FruitProvider">
    <custom-attributes foo="${arg.foo}"/><!-- store it for later use -->
    <template name="model">
        <listitem>
            <listcell label="${foo}"/> <!-- Correct! Use the stored copy. -->
            <listcell label="${each}"/>
```

```
        </listitem>
    </template>
</listbox>
```

Nested Listboxes

The template can be applied recursively. Here is an example of a listbox-in-listbox:

```
<zk>
      <zscript><![CDATA[
      ListModel quarters = new ListModelArray(new String[] {"Q1", "Q2",
 "Q3", "Q4"});
      Map months = new HashMap();
      months.put("Q1", new ListModelArray(new String[] {"Jan", "Feb",
"Mar"}));
      months.put("Q2", new ListModelArray(new String[] {"Apr", "May",
"Jun"}));
      months.put("Q3", new ListModelArray(new String[] {"Jul", "Aug",
"Sep"}));
      months.put("Q4", new ListModelArray(new String[] {"Oct", "Nov",
"Dec"}));
      ListModel qs = (quarters);
      ]]></zscript>
      <listbox model="${quarters}">
            <template name="model">
                  <listitem>
                        <listcell>${each}</listcell>
                        <listcell>
                              <listbox model="${months[each]}">
                                    <template name="model">
                                          <listitem label="${each}"/>
                                    </template>
                              </listbox>
                        </listcell>
                  </listitem>
            </template>
      </listbox>
</zk>
```



How to retrieve the outer template's data in the inner template

Although forEachStatus has an API called ForEachStatus.getPrevious() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ForEachStatus.html#getPrevious()), it always returns null[1] . It is because the template is rendered on demand. When ZK is rendering the inner template, the previous iteration has already gone. There is no way to retrieve the iteration information of the outer template.

Rather, you have to traverse the component tree or use the custom-attributes element.

Here is an example of traversing the component tree to retrieve the data in the outer template, as shown at line 9 below. Notice that, each data is, as described before, stored in the component's value property.

```
<listbox model="${quarters}">
      <template name="model">
            <listitem>
                  <listcell>
                        <listbox model="${months[each]}">
                              <template name="model">
                                    <listitem>
                                          <listcell label="${forEachStatus.index}" />

                                          <listcell>${self.parent.parent.parent.parent.pa
                                          <listcell>${each}</listcell>
                                    </listitem>
                              </template>
                        </listbox>
                  </listcell>
            </listitem>
      </template>
</listbox>
```

If the component tree is deep, It is tedious and somehow error prone. Alternatively, you can store the information into a custom attribute and then retrieve it later, as shown at line 4 and 10 below.

```
<listbox model="${quarters}">
      <template name="model">
            <listitem>
                  <custom-attributes master="${each}"/>
                  <listcell>
                        <listbox model="${months[each]}">
                              <template name="model">
                                    <listitem>
                                          <listcell label="${forEachStatus.index}" />
                                          <listcell>${master}</listcell>
                                          <listcell>${each}</listcell>
                                    </listitem>
                              </template>
                        </listbox>
                  </listcell>
            </listitem>
      </template>
</listbox>
```

[1] On the other hand, it returns the previous iteration information when using with the forEach attribute

Template for GroupsModel

When used with GroupsModel (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/GroupsModel.html#), listboxes will use the template called `model:group` for rendering the grouping object. If it is not defined, it will look for the template called `model` instead (i.e., the same template is used for rendering the grouping and non-grouping objects).

```
<listbox mode="${fooGroupsModel}">
   <template name="model:group">
      <listgroup open="${groupingInfo.open}" label="${each}"/>
   </template>
   <template name="model">
      <listitem>....</listitem>
   </template>
   <template name="model:groupfoot">
      <listgroupfoot>....</listgroupfoot>
   </template>
<listbox>
```

- Note the *groupingInfo* is used to get the information of the grouping data. Please refer to GroupingInfo (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/GroupingInfo.html#)

Version History

Version	Date	Content
6.0.0	July 2011	The template feature was introduced.
6.0.0	January 2012	The GroupingInfo statement was introduced.

Grid Template

Similar to Listbox, you can define a customer rendering with a template for a grid:

```
<grid model="${books}">
      <columns>
            <column label="ISBN" sort="auto"/>
            <column label="Name" sort="auto"/>
            <column label="Description"/>
      </columns>
      <template name="model">
            <row>
                  <label value="${each.isbn}"/>
                  <label value="${each.name}"/>
                  <label value="${each.description}"/>
            </row>
      </template>
</grid>
```

where books is assumed as an instance of ListModel [2] that contains a list of the Book instances while each Book instances has at least three getter methods: getIsbn, getName and getDescription.

Notice that the template named `model` must be associated with the grid, i.e., it must be a direct child element of the grid element as shown above. A common mistake is to put it under the rows element. Remember the template is a ZUML fragment telling the grid how to render a row, and the template itself is not a component.

Template for GroupsModel

When used with GroupsModel [2], grids will use the template called `model:grouping` for rendering the grouping object. If it is not defined, it will look for the template called `model` instead (i.e., the same template is used for rendering the grouping and non-grouping objects).

```
<grid mode="${fooGroupsModel}">
   <template name="model:group">
      <group open="${groupingInfo.open}">....</group>
   </template>
   <template name="model">
      <row>....</row>
   </template>
   <template name="model:groupfoot">
      <groupfoot>....</groupfoot>
   </template>
```

```
<grid>
```

- Note the *groupingInfo* is used to get the information of the grouping data. Please refer to GroupingInfo [1]

Version History

Version	Date	Content
6.0.0	July 2011	The template feature was introduced.
6.0.0	January 2012	The GroupingInfo statement was introduced.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/GroupingInfo.html#

Tree Template

Similar to Listbox, you can also define a customer rendering with a template for a tree:

```
<tree model="${files}">
	<treecols>
		<treecol label="Path"/>
		<treecol label="Description"/>
	</treecols>
	<template name="model">
		<treeitem context="menupopup">
			<treerow>
				<treecell label="${each.data.path}"/>
				<treecell label="${each.data.description}"/>
			</treerow>
		</treeitem>
	</template>
</tree>
```

assume files is an instance of DefaultTreeModel [4]. Notice since DefaultTreeModel [4] is used, each references an instance of DefaultTreeNode [5]. Thus, to retrieve the real data, use DefaultTreeNode.getData() [1]

Version History

Version	Date	Content
6.0.0	July 2011	The template feature was introduced.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/DefaultTreeNode.html#getData()

Combobox Template

Similar to Listbox, you can render a combobox with a template:

```
<combobox model="${infos}">
	<template name="model">
		<comboitem label="${each[0]}: ${each[1]}"/>
	</template>
</combobox>
```

where we assume there is a list model (ListModel [2]) called `infos` such as:

```
ListModel infos = new ListModelArray(
	new String[][] {
		{"Apple", "10kg"},
		{"Orange", "20kg"},
		{"Mango", "12kg"}
});
```

Version History

Version	Date	Content
6.0.0	July 2011	The template feature was introduced.

Selectbox Template

Similar to Listbox, you can render a selectbox with a template. However, notice that, unlike other components, selectbox doesn't allow any child component, so you have to render each item as a string. For example,

```
<selectbox model="${users}" onSelect='alert(model.get(event.getData()));'>
      <template name="model">
            Name is ${each}
      </template>
</selectbox>
```

where we assume there is a list model (ListModel [2]) called `users` such as:

```
ListModelList model = new ListModelList(new String[] { "Tony", "Ryan",
"Jumper", "Wing", "Sam" });
```

Version History

Version	Date	Content
6.0.0	November 2011	The selectbox component was introduced.
6.0.0	July 2011	The template feature was introduced.

Renderer

A renderer is a Java class that is used to render the items specified in a data model[1] . The implementation of a renderer depends on the component. For example, the display of Listbox [1] can be customized by an implementation of ListitemRenderer [2], and Grid [3] by RowRenderer [3][4] .

---

[1] If you prefer to define the rendering of each item in the ZUML document, you could use templates instead.

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

[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/RowRenderer.html#

[4] The same model usually can be shared by components having the same *logic model*. For example, ListMode (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListMode.html#) can be used in both Grid (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Grid.html#) and Listbox (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Listbox.html#). However, a renderer is usually specific to a particular component.

Listbox Renderer

Here we describe how to implement a custom renderer for a listbox (ListitemRenderer [2]). For the concepts about component, model and renderer, please refer to the Model-driven Display section.

When a listbox (Listbox [1]) is assigned with a model, a default renderer is assigned too. The default renderer will assume that each tree item has only one column, and it converts the data into a string directly[1] . If you want to display multiple columns or retrieve a particular field of the data, you have to implement ListitemRenderer [2] to handle the rendering.

For example,

```
public class MyRenderer implements ListitemRenderer{
      public void render(Listitem listitem, Object data, int index) {
            Listcell cell = new Listcell();
            listitem.appendChild(cell);
            if (data instanceof String[]){
                  cell.appendChild(new
Label(((String[])data)[0].toString()));
            } else if (data instanceof String){
                  cell.appendChild(new Label(data.toString()));
            } else {
                  cell.appendChild(new
Label("UNKNOW:"+data.toString()));
            }
      }
}
```

[1] If the tree is assigned a template called `model`, then the template will be used to render the tree. For more information, please refer to the Listbox Template section.

Version History

Version	Date	Content
6.0.0	February 2012	The index argument was introduced.

Grid Renderer

When a grid (Grid [3]) is assigned with a model, a default renderer is assigned too[1] . The default renderer will assume that each row has only one column, and it converts the data into a string directly[2] . If you want to display multiple columns or retrieve a particular field of the data, you have to implement RowRenderer [3] to handle the rendering.

For example,

```
public class FoodGroupRenderer implements RowRenderer,
java.io.Serializable {
    public void render(Row row, Object obj, int index) {
        if (row instanceof Group) {
            row.appendChild(new Label(obj.toString()));
        } else {
            User user = (User) obj;
            row.appendChild(new Label(user.getName()));
            row.appendChild(new Label(user.getDescription()));
            row.appendChild(new Label(user.getDomain()));
        }
    }
}
```

[1] For the concept about component, model and renderer, please refer to the Model-driven Display section.

[2] If the grid is assigned a template called `model`, then the template will be used to render the grid. For more information, please refer to the Grid Template section.

Version History

Version	Date	Content
6.0.0	February 2012	The index argument was introduced.

Tree Renderer

When a tree (Tree [2]) is assigned with a model, a default renderer is assigned too[1] . The default renderer will assume that each tree item has only one column, and it converts the data into a string directly[2] . If you want to display multiple columns or retrieve a particular field of the data, you have to implement TreeitemRenderer [3] to handle the rendering.

For example,

```
public class HostTreeRenderer implements TreeitemRenderer {
      public void render(Treeitem treeitem, Object data, int index)
throws Exception {
            Treerow row = treeitem.getTreerow();
            if (row == null) { // tree row not create yet.
                  row = new Treerow();
                  treeitem.appendChild(row);
            }
            if (data instanceof HostTreeModel.FakeGroup) {
                  treeitem.getTreerow().appendChild(new
Treecell(((HostTreeModel.FakeGroup)data).getName()));
            } else if (data instanceof HostTreeModel.FakeHost) {
                  treeitem.getTreerow().appendChild(new
Treecell(((HostTreeModel.FakeHost)data).getName()));
            } else if (data instanceof HostTreeModel.FakeProcess) {
                  treeitem.getTreerow().appendChild(new
Treecell(((HostTreeModel.FakeProcess)data).getName()));
            }
      }
}
```

[1] For the concept about component, model and renderer, please refer to the Model-driven Display section.

[2] If the tree is assigned a template called `model`, then the template will be used to render the tree. For more information, please refer to the Tree Template section.

[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/TreeitemRenderer.html#

Version History

Version	Date	Content
6.0.0	February 2012	The index argument was introduced.

Combobox Renderer

When a combobox (Combobox [2]) is assigned with a model, a default renderer is assigned too[1] . The default renderer will assume that the combobox displays the data as a string[2] . If you want to display more sophisticated information or retrieve a particular field of the data, you have to implement ComboitemRenderer [3] to handle the rendering.

For example,

```
public class MyRenderer implements ComboitemRenderer {
      public void render(Comboitem item, Object data, int index) throws
 Exception {
            item.setLabel(((User)data).getName());
            item.setDescription(((User)data).getDescription());
      }
}
```

[1] For the concept about component, model and renderer, please refer to the Model-driven Display section.

[2] If the tree is assigned a template called `model`, then the template will be used to render the tree. For more information, please refer to the Tree Template section.

[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ComboitemRenderer.html#

Version History

Version	Date	Content
6.0.0	February 2012	The index argument was introduced.

Selectbox Renderer

The implementation of a custom renderer for a Selectbox (ItemRenderer [1]) is straightforward[2] :

```
public class FooItemRenderer implements org.zkoss.zul.ItemRenderer {
    public String render(Component owner, Object data, int index)
throws Exception {
        return data.toString(); //converting data to a string; it
depends on your application's requirement
    }
}
```

Then, if we have a list model (ListModel [2]) called `users`, and an instance of a custom renderer called `userRenderer`, then we can put them together in a ZUML document as follows:

```
<selectbox model="${users}" itemRenderer="${userRenderer}"/>
```

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

[2] For the concept about component, model and renderer, please refer to the Model-driven Display section.

Version History

Version	Date	Content
6.0.0	November 2011	Selectbox was introduced.

Annotations

An annotation is a special form of syntactic metadata that can be added to components. The definitions, properties and components themselves may be annotated. The annotations can be retrieved at the run time.

The annotations have no direct effect on the operation of the components. Rather, they are mainly used for UI designers to annotate metadata, such that it controls how a tool or a utility shall do at run-time. The content and meanings of annotations totally depend on the tools or the utilities the developer uses. For example, ZK Bind examines annotations to know how to load and store the value of a component.

Annotate in ZUML

Annotations can be applied to the declarations of components and properties in ZUML pages.

Annotate Properties

To annotate a property, you could specify an annotation expression as the value of the property. In other words, if the value of the property is an annotation expression, it is considered as an annotation for the property, rather than a value to be assigned.

The format of an annotation expression:

```
@annotation-name ()
@annotation-name ( attr-name1 = attr-value1, attr-name2 = attr-value2 )
@annotation-name ( attr-name1 = { attr-value1-1, attr-value1-2 }, attr-name2 = attr-value2 )
```

As shown, an annotation consists of an annotation name and any number of attributes, and an attribute consists of an attribute name and an attribute value. The name of an annotation must start with a letter ('a' - 'z' or 'A' - 'Z'), an underscore ('_'), or a dollar sign ('$').

If an attribute has multiple values, these values have to be enclosed with the curly braces (as shown in the third format).

For example,

```
<listitem label="@bind(datasource='author',value='selected')"/>
```

where an annotation called `bind` is annotated to the `label` property, and the `bind` annotation has two attributes: `datasource` and `value`.

If the attribute name is not specified, the name is assumed to be `value`. For example, the following two statements are equivalent:

```
<textbox value="@bind(vm.p1.firstName)"/>
<textbox value="@bind(value=vm.p1.firstName)"/>
```

Here is a more complex example.

```
<textbox value="@save(vm.person.firstName,  before={'cmd1', 'cmd2'})"/>
```

where it annotates the `value` property with an annotation named `save`, and the annotation has two attributes: value and before. The value of the before attribute is a two-element array: 'cmd1' and 'cmd2'. Notice that the quotations, `'` and `"`, will be preserved, so they will be retrieved exactly the same as they are specified in the ZUML document.

To annotate the same property with multiple annotations, you could specify them one-by-one and separate them with a space, as shown below.

```
<textbox value="@bind(vm.value1) @validator('validator1')" errorMessage="@bind(vm.lastMes
```

In additions, you could annotate with multiple annotations that have the same name. For example,

```
<textbox value="@bind(vm.first) @bind(vm.second)"/>
```

where two annotations are annotated to the value property.

Annotate Components

To annotate a component, you could specify an annotation expression in a specific attribute called `self` as shown below.

```
<label self="@title(value='Hello World')"/>
```

where `self` is a keyword to denote the annotation which is used to annotate the component declaration, rather than any property.

The annotation Namespace

ZK Loader detects the annotation automatically. However, it may not be what in you expect. Here we discuss how to resolve these conflicts.

Specify both value and annotation

If you'd like to specify both the value and the annotations of a given property, you could specify a namespace called annotation to distinguish them. For example,

```
<textbox value="a property's value" a:value="@save(vm.user)" xmlns:a="annotation"/>
```

Then, the textbox's value property will be assigned with a value, "a property's value", and an annotation, @save(vm.user).

Specify a value that looks like an annotation

If the value of a property looks like an annotation, you could specify a namespace other than annotation to tell ZK Loader not to interpret it as an annotation. For example,

```
<textbox u:value="@value()" xmlns:u="zul"/>
```

Then, @value() will be considered as a value rather an annotation, and assigned to the textbox's value property directly.

Version History

Version	Date	Content
6.0.0	December 2011	The new syntax was introduced. For ZK 5's syntax, please refer to ZK 5's Developer's Reference. Though not recommended, it is OK to use ZK 5's syntax in ZK 6.

Annotate in Java

You could annotate a component or a property in Java by the use of java.lang.String, java.util.Map) ComponentCtrl.addAnnotation(java.lang.String, java.lang.String, java.util.Map) [1].

For example,

```
Listbox listbox = new Listbox();
listbox.addAnnotation(null, "foo", null); //null in the first argument
means to annotate listbox
Label label = new Label();
label.addAnnotation("value", "fun", null); //annotate the value
property of label
```

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/sys/ComponentCtrl.html#addAnnotation(java.lang.String,

Retrieve Annotations

The annotations can be retrieved back at the run-time. They are designed to be used by tools or utilities, such as the data-binding manager, rather than applications. In other words, applications annotate a ZUML page to tell the tools how to handle components for a particular purpose.

The following is an example to dump all annotations of a component:

```
void dump(StringBuffer sb, Component comp) {
   ComponentCtrl compCtrl = (ComponentCtrl)comp;
    sb.append(comp.getId()).append(": ")
      .append(compCtrl .getAnnotations(null)).append('\n');

    for (String prop: compCtrl.getAnnotatedProperties()) {
        sb.append(" with ").append(prop).append(": ")
            .append(compCtrl .getAnnotations(prop)).append('\n');
    }
}
```

Version History

Version	Date	Content

Annotate Component Definitions

In addition to annotating a component or its properties, you could annotate a component definition, such that all its instances will have the annotations.

To annotate a component definition, you have to specify the annotations in a language definition. For example, we could extend the definition of bandbox to add annotations. Please refer to ZK Component Reference/Annotation/Data Binding for detail.

```
<component>
	<component-name>bandbox</component-name>
	<extends>bandbox</extends>
	<annotation>
		<annotation-name>ZKBIND</annotation-name>
		<property-name>value</property-name>
		<attribute>
			<attribute-name>ACCESS</attribute-name>
			<attribute-value>both</attribute-value>
		</attribute>
		<attribute>
			<attribute-name>SAVE_EVENT</attribute-name>
			<attribute-value>onChange</attribute-value>
		</attribute>
		<attribute>
			<attribute-name>LOAD_REPLACEMENT</attribute-name>
			<attribute-value>rawValue</attribute-value>
		</attribute>
		<attribute>
			<attribute-name>LOAD_TYPE</attribute-name>
			<attribute-value>java.lang.String</attribute-value>
		</attribute>
	</annotation>
	<annotation>
		<annotation-name>ZKBIND</annotation-name>
		<property-name>open</property-name>
		<attribute>
			<attribute-name>ACCESS</attribute-name>
			<attribute-value>both</attribute-value>
		</attribute>
		<attribute>
			<attribute-name>SAVE_EVENT</attribute-name>
			<attribute-value>onOpen</attribute-value>
		</attribute>
```

```
      </annotation>
</component>
```

Version History

Version	Date	Content

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

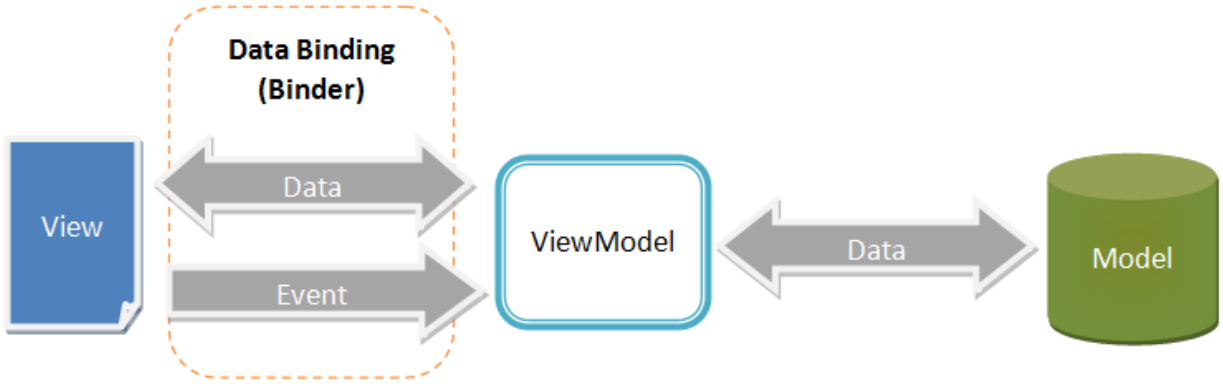
Separation of data and logic from presentation

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.

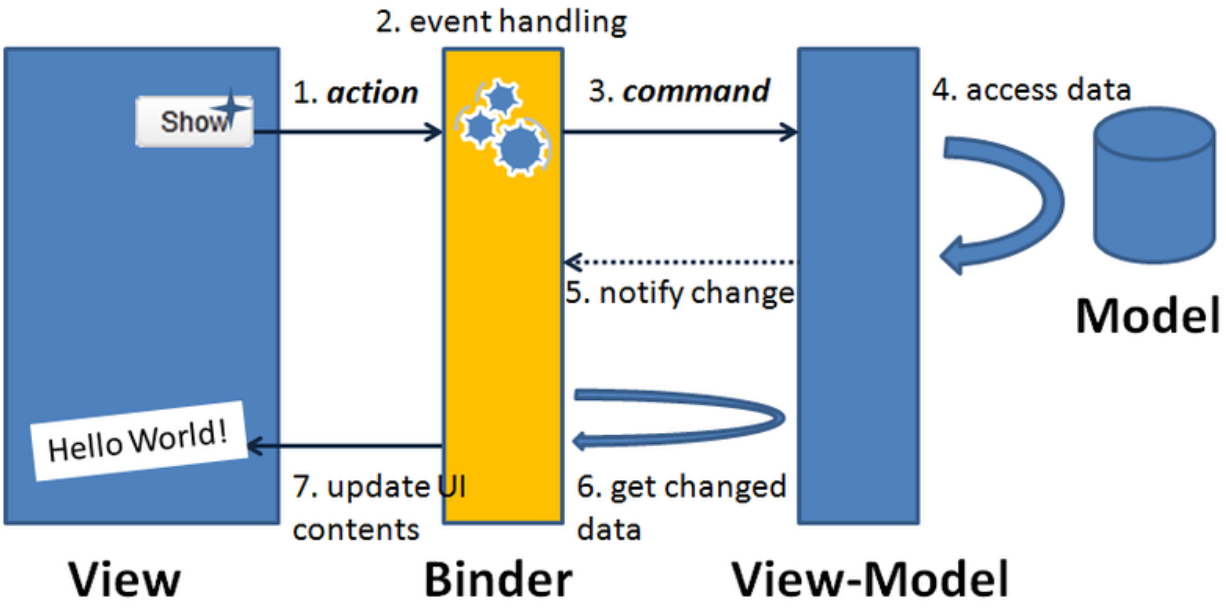
MVVM & ZK Bind

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

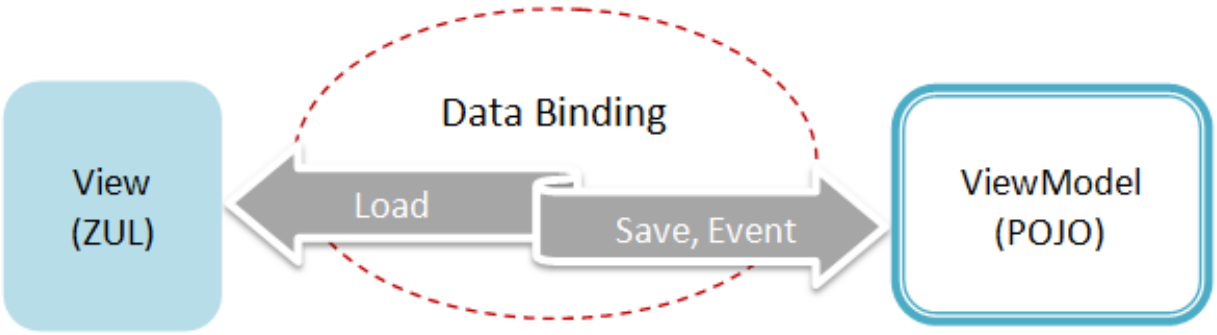
ViewModel

What is ViewModel

ViewModel 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.

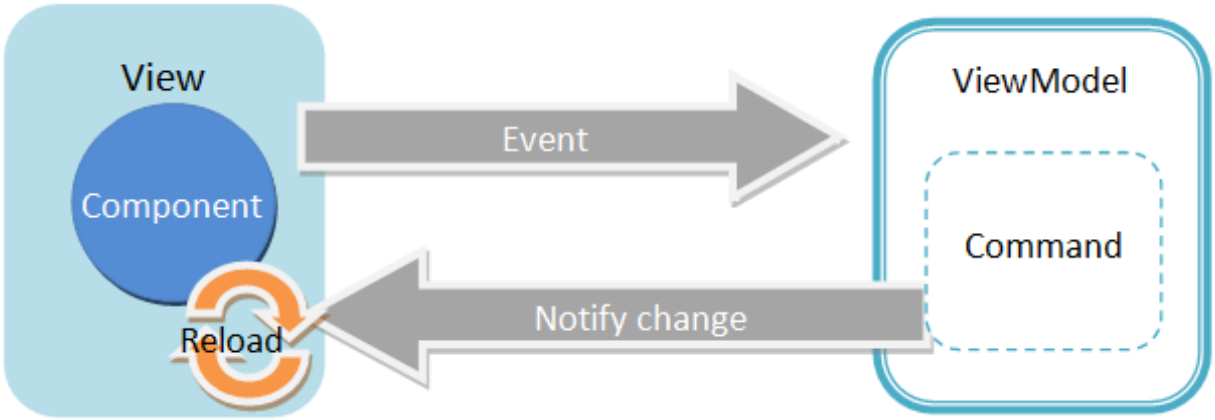
ViewModel is also a **Model of the View**. It contains the View's state (e.g. selection, enablement) that might change during user interaction. For example, if a button is enabled when a user selects an item of a listbox. ViewModel shall store the state of the button's enablement and implement the logic to enable the button.

Although ViewModel stores the View's states, it contains no reference to UI components. It can not access any UI components directly. Hence, there is a data binding mechanism to synchronize data between View and ViewModel. After developers define the binding relationship between View (UI component) and ViewModel, the data binding mechanism synchronize the data automatically. This makes ViewModel and View loose coupled.



ViewModel acts like a Controller in MVC, so the data binding mechanism forwards events to ViewModel's handlers. The handlers are ViewModel's method with specific Java annotation. We call such a method **Command** of the ViewModel. These methods usually manipulate data of the ViewModel, like deleting an item. Data binding mechanism also supports binding a UI component's event with a ViewModel's command. Firing the component's event will trigger the execution of bound command, invoking the annotated method.

During the execution of command, it usually changes the data of ViewModel. The developer should specify what properties change to be notified through Java annotation.



the UI sometimes has to display data in a different format from original one in the Model. The converting logic shall be implemented in ViewModel or developers can adopt a more reusable element **converter** that we will talk about in a later section.

Validation will also be performed before saving data into ViewModel. If validation fails, the data will not be saved to ViewModel.

ViewModel with Annotation

In ZK, ViewModel can be simply a **POJO**, and it knows nothing about View's visual elements which means it should not hold any reference to UI components. The data binding mechanism handles communication and synchronization between View and ViewModel. Developers have to specify command and data dependency of a ViewModel with ZK provided Java annotation, then data binding mechanism will know how to interact with the ViewModel.

Creating a ViewModel is like creating a POJO, and it exposes its properties like JavaBean through setter and getter methods.

```
public class MyViewModel{
	private String name;

	public String getName(){
		return name;
	}
	public voic setName(String name){
		this.name = name;
	}

	@NotifyChange({"selected","orders"})
	@Command
	public void newOrder(){
		//manipulate data
	}
}
```

- We'll describe above annotation in detail at ZK Developer's Reference/MVVM/ViewModel/Notification

Reference a ViewModel in a ZUL

We can bind ZK UI component to a ViewModel by setting its **viewModel** attribute, and that component becomes the **Root View Component** for the ViewModel. All child components of this Root View Component can acces the same ViewModel and its properties. To bind a ViewModel, we have to apply a composer called **org.zkoss.bind.BindComposer**, it will create a binder for the ViewModel and instantiate the ViewModel's class. Then we set **viewModel** attribute with ViewModel's id in `@id` ZK Bind annotation and a ViewModel's full-qualified class name in `@init` . We use the id to reference ViewModel's properties, e.g. vm.name. We'll explain ZK bind syntax in detail at subsections of ZK Developer's Reference/MVVM/DataBinding.

```
<window apply="org.zkoss.bind.BindComposer"       viewModel="@id('vm') @init('foo.MyViewM

    <label value="@bind(vm.name)"/>
    <button onClick="@command('newOrder')" />
</window>
```

Initialization

The binder is responsible for creating and initializing a ViewModel instance. If you want to perform your own initialization in a ViewModel, you can declare your own **initial method** by annotating a method with `@Init` . The Binder will invoke this method when initializing a ViewModel. Each ViewModel can only have one initial method.

```
public class MyViewModel {

      @Init
      public void init(){
            //initialization code
      }
}
```

This annotation has an element "superclass", if you set it to "true". The binder will look for the initial method of ViewModel's parent class and invoke it first if it exists.

```
public class ParentViewModel{

      @Init
      public void init(){
            //initialization code
      }
}


public class ChildViewModel extends ParentViewModel{

      @Init(superclass=true)
      public void init(){
            //initialization code
      }
```

```
}
```

- ParentViewModel's initial method is invoked first then ChildViewModel's.

The initial method can retrieve various context object by applying annotation on its parameter, please refer ZK Developer's Reference/MVVM/Advance/Parameters.

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

Data and Collections

Expose ViewModel's Properties

ViewModel, like a JavaBean, exposes its properties through getter and setter methods. Any property that VIew (ZUL) wants to retrieve through data binding should have a corresponding getter method. The method's return type can be primitive type (int, boolean...), or a JavaBean. If a UI component needs to save its attribute value (it's usually user input) back to ViewModel's property, the ViewModel should provide a corresponding setter method. Therefore, through setter and getter can change the data of both the View and the ViewModel using the data binding mechanism.

Property is Primitive Type

Primitive type property

```
public class PrimitiveViewModel{

	private int index;
	private double price;

	public int getIndex(){
		return index;
	}
	public void setIndex(int index){
		this.index = index;
	}

	public double getPrice(){
		return price;
	}
	public void setPrice(double price){
		this.price = price;
	}

}
```

A zul that uses PrimitiveViewModel

```
<window apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init('foo.PrimitiveView

    <intbox value="@bind(vm.index)"/>

    <doublebox value="@bind(vm.price)"/>

    <!-- other components -->

</window>
```

- @id('vm') sets ViewModel name, then we can use `vm` to reference ViewModel's property in the following components. (line 1)

Property is Object or JavaBean

If a property is a JavaBean, that JavaBean's property can also be accessed through an EL expression.

Object type property

```
public class Address{
      private String city;
      private String street;

      //gettter & setter
}

public class ObjectViewModel{

      private Integer index;
      private String name;
      //JavaBean
      private Address address;

      public int getIndex(){
            return index;
      }
      public void setIndex(Integer index){
            this.index = index;
      }

      public String getName(){
            return name;
      }
      public void setName(String name){
            this.name = name;
      }

      public Item getAddress(){
```

```
            return address;
      }
      public void setAddress(Address address){
            this.address = address;
      }

}
```

A zul that uses ObjectViewModel.

```
    <intbox value="@bind(vm.index)"/>

    <textbox value="@load(vm.name)"/>

    <label value="@load(vm.address.street)"/>
```

- vm.address is a JavaBean, and we can access its property **street** with EL expression vm.address.street. (line 5)

Property is Collection

If the UI component is a collection container like listbox or grid, it should be bound to a property whose type is `Map` or subinterface of `Collection` like `List` or `Set` .

Collection type property

```
public class CollectionViewModel{
      //primitive type
      private int selectedIndex;
      //Collection
      private List itemList;

      public int getSelectedIndex(){
            return selectedIndex;
      }
      public void setSelectedIndex(int index){
            selectedIndex = index;
      }

      public List getItemList(){
            return itemList;
      }
}
```

- As we can't save whole list through data binding, the ViewModel only provide setter for itemList.

A zul that uses CollectionViewModel

```
    <label value="@load(vm.address.street)"/>
    <listbox model="@load(vm.itemList)" selectedIndex="@bind(vm.selectedIndex)">
        <!-- children of listbox -->
    </listbox>
```

- Listbox's model attribute should be bound to a collection object, vm.itemList. (line 4)

Version History

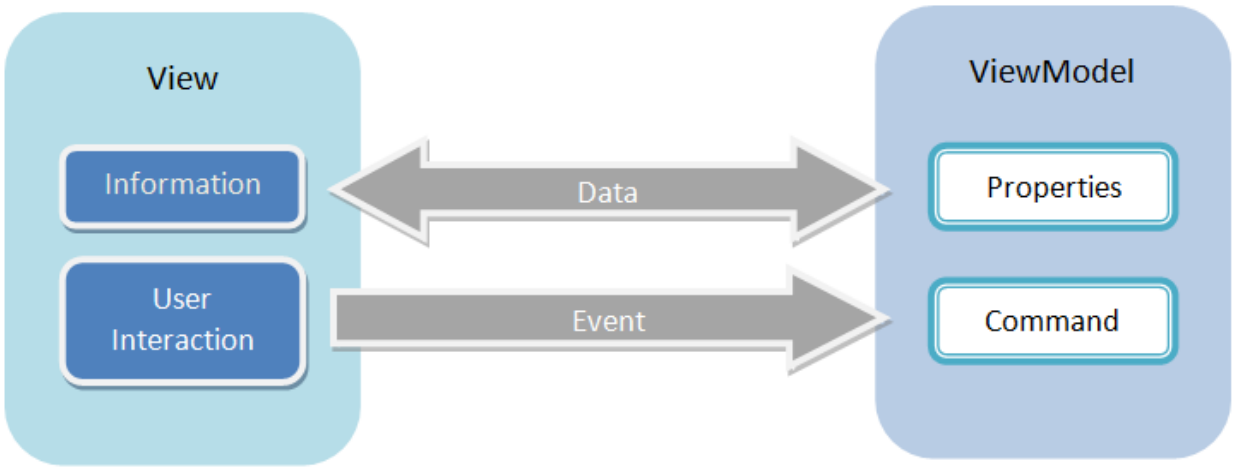
Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

Commands

Introduction

The ViewModel is an abstraction of the View. The View is responsible for displaying information and interacting with users. The information corresponds to ViewModel's property and interaction corresponds to ViewModel's **Command**. The **Command** is an action to manipulate ViewModel's property. Each command provides an action that the View can perform on ViewModel. These actions are also the ways that users can interact with View. For example, a ViewModel provides 2 commands: "save" and "delete". It means users can only perform these 2 actions on the View with the ViewModel. They could perform actions through clicking buttons or menuitems.

As ViewModel acts a role like Controller, developers can bind a UI component's event to a Command by specifying Command's name which is similar to register a event listener. Multiple events can bind to the same Command. When a user interacts with a component (e.g. click a button), the component fire an event then the data binding mechanism triggers the execution of Command. The Command may modify ViewModel's properties and then information displayed on View changes.



The Command is implemented as ViewModel's method. Because ViewModel is a POJO, in order to make data binding mechanism identify which method represent a Command, developers have to annotate the method with ZK provided `@Command` annotation. We'll use the term: **Command method** to depict the special annotated method of a ViewModel in the later section. These methods usually manipulate ViewModel's property, like deleting an item. Firing a component's event triggers the execution of bound command, that is invoking the Command method. During executing the Command, the developer also has to specify what properties change to notify through Java annotation that we will describe in later section.

Declare Commands

Local Command

ViewModel's Command is like an event handler, we can bind an event to a Command. The binding between events and a command is what we call "Event-Command Binding". Before establish this binding, we have to declare a Command with its name in a ViewModel. Be careful that command names in a ViewModel cannot be duplicated, or it will cause run-time exception.

```
public class OrderVM {

	// create and add a new order to a list
	// command name is set to method name by default
	@Command
	public void newOrder(){
		Order order = new Order();
		getOrders().add(order); //add a new order to order list
		selected = order;//select the new one
	}

	// save an order
	// command name is specified
	@Command("save")
	public void saveOrder(){
		orderService.save(selected);
	}

	// delete an order
	// multiple command names
	@Command({"delete", "deleteOrder"})
	public void deleteOrder(){
		//delete order
	}
}
```

- Notice that we can declare a Command without specifying its name, and its name is set to method name by default. (line 5)
- We can also give Command's name by `@Command('userDefinedName')` . (line 14)
- We can even give multiple Command's name with array of String. (line 21)

Then we can bind component's event to the command in the ZUL.

```
	<toolbar>
		<button label="New" onClick="@command('newOrder')" />
		<button label="Save" onClick="@command('save')" />
	</toolbar>
```

We describe the detail of command binding here. This binding allow you to pass parameters to Command method, please refer here.

Global Command

Global Command is also a ViewModel's command and can hook UI component's events to it. The local command can only be triggered by events of a ViewModel's Root View Component and its child components. The main difference of a global command from local command is that the event doesn't have to belong to the ViewModel's root view component or its child component. By default we can bind an event to any ViewModel's global command **within the same desktop**. A method can be both a local command and a global command.

```
@Command("delete") @GlobalCommand("delete")
public void deleteOrder(){
    //...
}
```

We can declare multiple global commands with same name in different ViewModel. When an event triggers a global command, all matched command methods in every ViewModel will be executed.

```
public class MainViewModel {

      @GlobalCommand
      public void show(){
            //...
      }
}


public class ListViewModel {

      @GlobalCommand
      public void show(){
            //...
      }
}
```

- If we trigger global command "show", each binder associated with each ViewModel will execute the `show` global command method but not in any particular order.

Command Execution

A command execution is a mechanism of ZK Bind where it performs a method call on the ViewModel. It binds to a component's event and when a binding event comes, binder will follow the lifecycle to complete the execution. We'll describe this in detail in Command Binding and Global Command Binding.

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

Notification

Overview

The binder is responsible for synchronizing data between the View and ViewModel. Typically the ViewModel is a POJO and has no reference to any UI component. Hence it cannot push data to a UI component. Developers have to specify the dependency of ViewModel's properties with ZK provided Java annotation, then binder knows when to reload which property and to update the UI view.

Notify Change

ZK bind provides a set of Java annotations (`@NotifyChange`, `@DependsOn`, `@NotifyChangeDisabled`) to specify **when to reload which property to UI components**. Developers have to specify it at design time, and the binder will synchronize data at run-time. The binder keeps track of binding relationships between component's attribute and ViewModel's property. After each time it invokes a ViewModel's method (setter, getter, or command method), it looks for corresponding component attribute to reload according to specified properties in the annotation.

Notify on Command

During execution of a command, one or more properties are changed because Command method usually contains business or presentation logic. Developers have to specify which property (or properties) is changed in Java annotation's element, then the data binding mechanism can reload them from ViewModel to View at run-time after executing the Command.

The syntax to notify property change:

One property:

```
@NotifyChange("oneProperty")
```

Multiple properties:

```
@NotifyChange({"property01","property02"})
```

All properties in a ViewModel:

```
@NotifyChange("*")
```

Following is an example of @NotifyChange:

```
public class OrderVM {

	//the order list
	List<Order> orders;
	//the selected order
	Order selected;

	//getter and setter

	@NotifyChange("selected")
	@Command
	public void saveOrder(){
		getService().save(selected);
```

```
	}

	@NotifyChange({"orders","selected"})
	@Command
	public void newOrder(){
		Order order = new Order();
		getOrders().add(order);
		selected = order;//select the new one
	}

	@NotifyChange("*")
	@Command
	public void deleteOrder(){
		getService().delete(selected);//delete selected
		getOrders().remove(selected);
		selected = null; //clean the selected
	}

}
```

- In `newOrder()` , we change property "orders" and "selected", therefore we apply `@NotifyChange` to notify binder to reload 2 changed properties after command execution. (line 16)
- The `deleteOrder()` also change the same 2 properties. Assume that this ViewModel class has only 2 properties, we can use **"*"** (asterisk sign) to notify all properties in current class. (line 24)

Notify on Setter

For similar reason, property might be changed in setter method. After value is saved to ViewModel's property, multiple components or other properties might depend on the changed property. We also have to specify this data dependency by `@NotifyChange` .

Enabled by Default

A setter method usually changes only one property, e.g. setMessage() will set *message* to new value. To save developer's effort, the target property changed by setter method is notified automatically. That is, `@NotifyChange` annotation on a setter is not required unless there are some other properties changed by the setter and need to be explicitly notified.

```
public class MessageViewModel{
	private String message;

	public String getMessage() {
		return this.message;
	}
	public void setMessage(String msg) {
		message = msg;
	}
}
```

- There is no `@NotifyChange` on setter method in `MessageViewModel` .

The ZUL that uses `MessageViewModel` .

```
<textbox id="msgBox" value="@bind(vm.message)"/>

<label id="msg1" value="@load(vm.message)"/>

<label id="msg2" value="@load(vm.message)"/>
```

- When a user input value in msgBox and the value is saved back to ViewModel, the data binding mechanism will synchronize the value to msg1 and msg2 automatically.

Developer can disable this default behavior by adding `@NotifyChangeDisabled` on setter method.

Specify Dependency

For setter method, `@NotifyChange` is used when multiple properties have dependency relationship, e.g. one property's value is calculated from the other properties. Notice that if you apply `@NotifyChange` on a setter method, default notification behavior will be overridden.

@NotifyChange on Setter

```
public class FullnameViewModel{

	//getter and setter

	@NotifyChange("fullname")
	public void setFirstname(String firstname) {
		this.firstname = firstname;
	}

	@NotifyChange({"lastname","fullname"})
	public void setLastname(String lastname) {
		this.lastname = lastname;
	}

	public String getFullname() {
		return (firstname == null ? "" : firstname)	+ " "
			+ (lastname == null ? "" : lastname);
	}
}
```

- By default setLastname() will notify "lastname" property's change. Becuase we apply `NotifyChange` on it, this default notification is overridden. We have to notify it explicitly. (line 10)

In above code snippet, fullname is concatenated by firstname and lastname. When firstname and lastname is changed by setter, the fullname should be reload to reflect its change.

If a property depends on many properties, `@NotifyChange` will distribute in multiple places in a ViewModel class. Developers can use `@DependsOn` for convenience. This annotation defines the dependency relations among properties and can provide same function as explicit `@NotifyChange`, but it's used on getter method. **Example of @DependsOn**

```
public class FullnameViewModel{
```

```
	//getter and setter

	public void setFirstname(String firstname) {
		this.firstname = firstname;
	}

	public void setLastname(String lastname) {
		this.lastname = lastname;
	}

	@DependsOn({"firstname", "lastname"})
	public String getFullname() {
		return (firstname == null ? "" : firstname)	+ " "
				+ (lastname == null ? "" : lastname);
	}
}
```

- The example is functional equivalent to previous one, but written in reversed meaning. It means when any one of 2 properties (firstname, lastname) change, fullname should be reloaded.

Notify Bean Change

If you bind a component's attribute to a bean and want to reload it after the bean's property changed, you should apply the following syntax on the setter of that target property:

`@NotifyChange(".")`

Note a dot(.) is a bit different from an asterisk(*) because an asterisk, `@NotifyChange("*")` , just notify binder to reload those components that bound to the bean's properties, e.g. `@bind(vm.person.firstname)`, `@bind(vm.person.lastname)`, and `@bind(vm.person.fullname)`. The component attribute that bound to the bean itself, i.e. `vm.person` , will **NOT** be reloaded with the @NotifyChange("*"). In such case you have to use `@NotifyChange(".")` to tell binder that the bean itself has changed and the component attribute bound to the bean shall be reloaded.

For what use cases you will need this? Most of the time, we bind one component attribute to only one bean property. However, there are cases that you might want to bind the component attribute to a calculation result of multiple bean properties; yet you don't want to put that calculation logic in the bean. Then you can bind the component attribute to the whole bean and have a converter to convert the data upon the bean's multiple properties.

For example, assume there is a web page in a tour website. A customer has to fill the leave and return day of a trip and the duration of the trip shall be automatically calculated and show on the screen after the user fills either fields. Here we choose to use a converter to do the duration calculation instead of implementing the logic in the Trip class as a property. In such scenario, you will need to annotate @NotifyChange(".") on both leaveDate and returnDate property setters so when either leaveDate or returnDate property changes, the duration attribute can be updated.

Bind to an object example

```
	<hlayout>
		Leave Date: <datebox value="@save(vm.trip.leaveDate)"/>
	</hlayout>
	<hlayout>
		Return Date: <datebox value="@save(vm.trip.returnDate)"/>
	</hlayout>
```

```
      Duration including leave and return(day): <label value="@load(vm.trip) @converter(v
```

- Because durationConverter needs to access 2 properties (leaveDate, returnDate), we have to bind label's value to the trip bean itself not an individual property of the bean. (line 7)

@NotifyChange(".") example

```
public class DurationViewModel  {

      private Trip trip = new Trip();

      public class Trip{

            private Date leaveDate;
            private Date returnDate;

            public Date getLeaveDate() {
                  return leaveDate;
            }
            @NotifyChange(".")
            public void setLeaveDate(Date leaveDate) {
                  this.leaveDate = leaveDate;
            }
            public Date getReturnDate() {
                  return returnDate;
            }
            @NotifyChange(".")
            public void setReturnDate(Date returnDate) {
                  this.returnDate = returnDate;
            }


      }


      public Converter getDurationConverter(){
            return new Converter() {

                  public Object coerceToUi(Object val, Component
component, BindContext ctx) {
                        if (val instanceof Trip){
                              Trip trip = (Trip) val;
                              Date leaveDate = trip.getLeaveDate();
                              Date returnDate = trip.getReturnDate();
                              if (null != leaveDate && null !=
returnDate){
                                    if
(returnDate.compareTo(leaveDate)==0){
                                          return 1;
```

```
                                    }else if
(returnDate.compareTo(leaveDate)>0){
                                        return ((returnDate.getTime()
 - leaveDate.getTime())/1000/60/60/24)+1;
                                    }
                                    return null;
                                }
                            }
                            return null;
                        }

                        public Object coerceToBean(Object val, Component
component, BindContext ctx) {
                            return null;
                        }
                };
        }
}
```

- About how to implement a converter, please refer to ZK Developer's Reference/MVVM/DataBinding/Converter.

Notify Programmatically

Sometimes the changed properties we want to notify depends on value at run-time, so we cannot determine the property's name at design time. In this case, we can use `BindUtils.postNotifyChange()` to notify change dynamically. The underlying mechanism for this notification is binder subscribed event queue that we talk about in the binder section. It uses **desktop scope** event queue by default.

Dynamic Notification

```
        String data;
        // setter & getter

        @Command
        public void cmd(){
                if (data.equal("A")){
                        //other codes...
                        BindUtils.postNotifyChange(null,null,this,"value1");
                }else{
                        //other codes...
                        BindUtils.postNotifyChange(null,null,this,"value2");
                }

        }
```

- The first parameter of postNotifyChange() is queue name and second one is queue scope. You can just leave it null and default queue name ans scope (desktop) will be used.

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

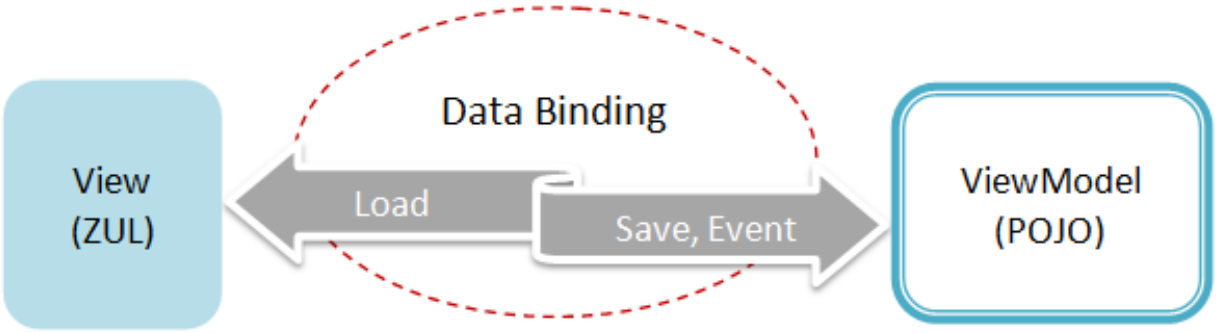
Data Binding

Overview

The data binding mechanism plays key role in synchronizing data between the View and ViewModel when developing a web application using the MVVM pattern.

Data binding is a mechanism to ensure that any change made to the data in a UI components is automatically carried over to the target ViewModel based on a predefined binding relationship. (and vice versa). Application developers only have to define the data binding relationship between UI component's attribute and the target object, it's usually a ViewModel, by data binding annotation expression.

ZK Bind is a whole new data binding mechanism with new specifications and implementations. Based on the experiences learned from data binding 1 and taking into account suggestions and feedback from users and contributors, we have created this easy to use, flexible, feature-rich new data binding system in ZK6.



Developers use ZK bind annotation to define various data binding relationship. The **binding source** is a component's attribute and **binding target** is a ViewModel's property or Command. When we bind a component to a ViewModel, that component becomes the ViewModel's **Root View Component**. Attributes of all child components and root view component itself can be bound to the ViewModel's properties (or command) through ZK bind annotation. This Root View Component doesn't have to be the root component of a page,

For example:

```
<vlayout>
	<vbox apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init('foo.Mai
		<label value="@load(vm.msg)" />
	</vbox>
</vlayout>
```

- The vbox is the Root View Component of ViewModel "foo.MainViewModel"
- The label's value attribute is binding source, ViewModel's msg property is binding target.

Quick Preview

Annotation Expression

- Java style annotation expressions in zul: The new ZK annotation is consistent with Java's annotation style. If you know the Java style, you know the ZK Style.
- A set of collaborated annotations: ZK Bind uses a set of annotations to make the use of data binding as intuitive and clear as possible.
  - **@id(...)**: used to idientify a instance's id , ex a view model or a form.
  - **@init(...)**: used to init a instance
  - **@load(...)**: used to bind data and command along with parameters for loading data to target
  - **@save(...)**: used to bind data and command along with parameters for saving data.
  - **@bind(...)**: used to bind data along with parameters for both loading and saving.
  - **@converter(...)**: used to specify converter along with parameters
  - **@valiator(...)**: used to specify validator along with parameters
  - **@command(...)**: used to bind an event to a command along with parameters.
  - **@global-command(...)**: used to bind an event to a global command along with parameters.
  - **@template(...)**: used to determine the template to render a container component.

EL 2.2 Flexible Expressions

- ZK Bind accepts EL 2.2 syntax expressions in which you can provide flexible operations easily.
- Bind to bean properties, indexed properties, Map keys seamlessly.
- Bind to component custom attributes automatically.
- Bind to Spring, CDI, and Seam managed bean automatically.

```
<image src="@load(vm.person.boy ? 'boy.png' : 'girl.png')"/>
```

```
<button onClick="@command(vm.add ? 'add' : 'update')" label="@load(vm.add ? 'Add' : 'Upda
```

```
<button onClick="@command('subscribe')" disabled="@load(empty vm.symbol)" label="Subscrib
```

One Way Load Only

- Load when bean property changes
- Conditional load after/before executing a command
- Multiple conditional load before/after executing different/same commands

```
<label value="@load(vm.person.fullname)"/>
```

```
<label value="@load(vm.person.firstname, after='update')"/>
```

```
<label value="@load(vm.person.firstname, before='delete')"/>
```

```
<label value="@load(vm.person.firstname, after={'update','delete'})"/>
```

```
<label value="@load(vm.person.firstname, after='update') @load(vm.person.message, after='
```

One Way Save Only

- Save when UI component attribute changes
- Multiple save to property of different target beans
- Conditional save before/after executing a command
- Multiple conditional save before/after executing different/same commands

```
<textbox value="@save(vm.person.firstname)"/>
```

```
<textbox value="@save(vm.person.firstname) @save(vm.tmpperson.firstname)"/>
```

```
<textbox value="@save(vm.person.firstname, before='update')"/>
```

```
<textbox value="@save(vm.person.firstname, after='delete')"/>
```

```
<textbox value="@save(vm.selected.firstname, before={'update','add'})"/>
```

Initial Binding

- Loads when UI components are first added into the binding system

```
<label value="@init(vm.selected.firstname)"/>
```

Two Way Data Binding

- Multiple conditional load and save on different back-end beans before/after executing different/same commands

```
<textbox value="@load(vm.selected.firstname) @save(vm.selected.firstname) @save(vm.newper
```

- Short expression for both save and load bindings without command condition.[1]

```
<textbox value="@bind(vm.person.firstname)"/>
```

[1] @bind(...) is shortcut for "@load(...) @save(...)", and @save is automatically ignored if the property doesn't support it

Bind to Any Attributes

- Bind to all attributes of UI components

```
<textbox value="@bind(vm.symbol)" instant="true"/>
<button disabled="@load(empty vm.symbol)" label="Subscribe" />
```

Event Command Binding

- Bridge ZK event to command
- Automatic event listener registration
- Simple command name invocation

```
<button onClick="@command('subscribe')" disabled="@load(empty vm.symbol)" label="Subscrib
```

Collection Binding

- Binding on Listbox/Grid/Tree/Combobox
- Local variable scope is limited to the container component
- Support index property

```
<listbox width="300px" model="@load(vm.albumList)" selectedItem="@bind(vm.selectedAlbum)"
    <template name="model" var="a">
        <listitem label="@load(a.title)"/>
    </template>
</listbox>
```

Form Binding

- Middle form binding to avoid affecting back-end data beans
- Submit a form in a whole
- Conditional save for different commands

```
<grid form="@id('fx') @load(vm.selected) @save(vm.selected, before='update') @save(vm.new
    <row>Title: <textbox value="@bind(fx.title)"/></row>
    <row>Artist: <textbox value="@bind(fx.artist)"/></row>
    <row><checkbox checked="@bind(fx.classical)"/> Classical</row>
    <row>Composer: <textbox value="@bind(fx.composer)"/></row>
</grid>
<button onClick="@command('add')" label="Add"/>
<button onClick="@command('update')" label="Update"/>
```

Embedded Validation Cycle

- Bind validator by name or by EL expression
- Embedded system Validator: provide commonly used validators in which users can use directly by only specifying the name
- Validate a single property or a form
- Validate upon a command

```
<textbox value="@save(vm.selected.firstname) @validator('beanValidator')"/>
```

```
<grid form="@id('fx') @load(vm.selected) @save(vm.selected, before='update') @validator(v
    <row>username<textbox value="@bind(fx.username)"/></row>
    <row>password<textbox value="@bind(fx.password)" type="password"/></row>
    <row>retype password<textbox value="@bind(fx.retypePassword)" type="password"/></row>
</grid>
```

Enhanced Converter Mechanism

- Bind converter by name or by EL expression
- Embedded system Converters: provide commonly used converters in which users can use directly by only specifying the name

```
<datebox value="@bind(vm.selected.birthday) @converter('formatedDate', format='yyyy/MM/dd
```

Dynamic Template

- Dynamically determine which template to render.

```
<grid model="@bind(vm.nodes) @template(vm.type='foo'?'template1':'template2')">
    <template name="template1">
    <!-- child components -->
    </template>

    <template name="template2">
    <!-- child components -->
    </template>
</grid>
```

EL Expression

EL Expression in Data Binding

In ZK bind annotation, we adopt EL expression to specify a binding target and reference an implicit object. The binding target is mostly a ViewModel's (nested) properties. You can use EL expression in a ZUL which is described in the section ZK Developer's Reference/UI Composing/ZUML/EL Expressions. But using EL in ZK bind annotation is a little bit different in format and evaluation.

Basic Format

All ZK bind annotation has the general format:

@[Annotation](value=[EL-expression], [arbitraryKey]=[EL-expression])

It starts from a "@" symbol followed by an annotation's name like "id" or "bind". Inside parentheses we can write multiple **key-value pairs** separated with a comma. The key is a self-defined name (not an EL expression), and it's like a key in a Map. The value is an EL expression but is **not enclosed with "${" and "}"**. The default key name is **"value"**. If you only write a EL expression without specifying its key name, it's implicit set to key named "value". Hence we usually omit this default key name when writing ZK bind annotation. In most case, we can just write a annotation as follows:

@[Annotation]([EL-expression])

We just need to write one key-value pair and omit default key name. We often use multiple key-value pairs for passing parameters to command, converter, and validator.

Although all ZK bind annotation has the general format, the way how a binder parse and evaluates the EL expression is different among different annotations. We'll describe the differences in the following sections.

Run-time Evaluation

A binder usually evaluates an EL expression each time when it wants to access the target object. Hence The evaluation result might be different from time to time.

Command binding according to run-time value

```
<button label="Cmd" onClick="@command(vm.checked?'command1':'command2')" />

<groupbox visible="@load(not empty vm.selected)" />
```

- When clicking the button, the binder executes a command upon value of "vm.checked".

Indirect reference

```
<label value="@bind(vm.person[vm.currentField])"/>
```

- If evaluation result of vm.currentField is firstName, the binder loads vm.person.firstName. So which property of vm.person that a binder loads depends on vm.currentField's evaluation result.

Call Methods

You can use EL expression to call a method in a ViewModel.

Call concat() of a ViewModel

```
<label value="@load(vm.concat(vm.foo,'postfix'))"/>
```

Tips

As ZK bind annotation is mainly composed of key-value pairs (key=value), **"="** (equal mark) is not allowed to be used in EL expression. You should replace it with "eq" and "!=" with "ne". But you still can use ">" and "<". For example:

```
<image src="@load(vm.picture ne null ? 'images/'.concat(vm.picture) : 'images/NoImage.png

<label value="@bind(vm.age>18?'true':'false')"/>
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

BindComposer

To enable the data binding in the ZUL, you have to apply a BindComposer [1] on a component (said `Root` component of this data binding). The `BindComposer` implements Composer [2] and plays a role to activate data binding mechanism for a component and its children components. It also initializes a Binder and ViewModel and passes ViewModel object's reference to Binder.

Apply BindComposer

To use a ViewModel you have to apply a BindComposer by setting "org.zkoss.bind.BindComposer" to "apply" attribute of a component.

```
<window id="win" apply="org.zkoss.bind.BindComposer">
<!-- other components inside will have data binding ability-->
</window>
```

Initialize a ViewModel

You also have to specify ViewModel's full-qualified class name to initialize it and give it an ID. The typical usage is like:

```
<window id="win" apply="org.zkoss.bind.BindComposer"
        viewModel="@id('vm') @init('foo.MyViewModel')">
    <label value="@load(vm.message)"/>
<!-- other components -->
</window>
```

In above code example, after the target component, window, is composed, the `BindComposer` will try to resolve the string literal inside `@init()` as a `Class` object and instantiate it. If it succeeds, it will store ViewModel object as an attribute of the root component for future use and it stores with the key **vm** which is specified in `@id` . Therefore, any child component of <window> can reference ViewModel by the id **vm**.

If there is no ViewModel attribute specified, the binder itself will become the ViewModel. That means you can apply a ViewModel which inherit `BindComposer` without specifying ViewModel attribute, but we only suggest this usage for experienced ZK user.

Wire Variable Automatically

If a member field is annotated by `@WireVariable` in a ViewModel, The variable (if existed) will be wired into this field automatically before connecting Binder and ViewModel. Read Wire Variable for more detail about wire variable. Following is a example that shows how to wire a messagService variable to the ViewModel.

```
public class OrderVM {
      @WireVariable
      MessageService messageService;
...
}
```

Initialize Binder

If you don't specify a root component's binder attribute, `BindComposer` creates `AnnotateBinder` by default. You usually don't have to specify binder attribute unless you want to use your own binder. You can get the `Binder` object by invoke `getAttribute("binder")` on the root component. In above example, it's `win.getAttribute("binder")` . We only suggest this usage to experienced ZK users.

Initialize Validation Message Holder

The validate message holder is also created by `BindComposer` . It's a container of validation messages that are generated by Validator during validation. Before using validation message holder, we have to give it an id in validationMessages attribute. We can retrieve validation message from it with a component as a key. We'll describe the detail in section.

```
<window title="Order Management" border="normal" width="600px"
      apply="org.zkoss.bind.BindComposer" viewModel="@id('vm')
@init(orderVm)"
      validationMessages="@id('vmsgs')">
            <hlayout>
                  <intbox id="qbox"
                  value="@load(vm.selected.quantity)
@save(vm.selected.quantity, before='saveOrder')
@validator(quantityValidator)"/>
                  <label value="@bind(vmsgs[qbox])" sclass="red" />
            </hlayout>
</window>
```

- Give validation message holder an id in order to reference it. (line **3**)
- We can retrieve validation messages of a input component, it's qubox, with the component as a key. (line **7**)

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

References

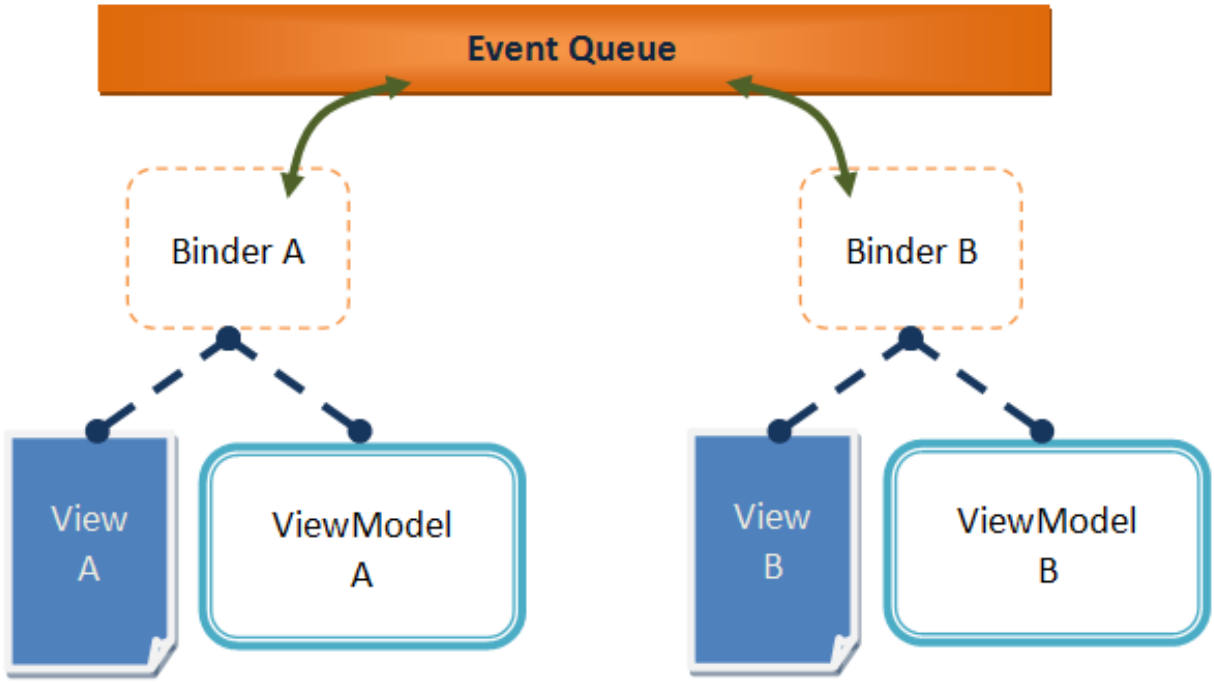
[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/bind/BindComposer.html#

Binder

The Binder [1] is the key in the whole data binding mechanism. When we apply a BindComposer [1] on a component, it will create a binder and pass component's and ViewModel's object reference to the binder.

Binder parses ZK bind annotations on ZULs to establish data binding relationship between components and ViewModel. It also synchronizes data between ZUL(View) and ViewModel based on the binding relationship and forwards events to ViewModel's command method.

Typically ViewModel is just a POJO. As it has no knowledge about others, a ViewModel can only communicate with another one by binder. The binder is like a broker and uses ViewModel's meta data(annotation) and methods to help it communicate with others. By default all binders subscribe to a default **desktop-scoped event queue**, thus this is a common communication mechanism among binders. There are 2 features: global command binding and dynamic change notification that operate based on this mechanism.



ZK also allows you to change default queue name and scope that a binder subscribes. You can separate binders into different groups by changing queue's name and scope upon your requirement.The syntax is as follows:

```
<window apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init('foo.MyViewModel')
      binder="@init(queueName='myqueue')">


<window apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init('foo.MyViewModel')
      binder="@init(queueScope='session')">
```

For available scopes, please refer ZK Developer's Reference/Event Handling/Event Queues.

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/bind/Binder.html#

Initialization

We can initialize any component's attribute by initial binding: `@init` . It loads ViewModel's property once at the beginning then the property won't be synchronized by binder during user interaction. But EL evaluation's result we expect in initial binding annotation is different for attribute "viewModel" and "form".

@init at viewModel Attribute

The first place we usually use this annotation is to assign a ViewModel to a component in viewModel attribute. When using in viewModel attribute, we should specify the **full-qualified class name** in a string literal in `@init` .

```
<window apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init('foo.MyViewModel')
</window>
```

The composer will resolve the string 'foo.MyViewModel' and create a object of it.

@init at Component's Attribute

It's common to use it to initialize a component's attribute with a constant value or a ViewModel's proeprty. The binder loads it once and doesn't synchronize it afterward during following user interaction.

```
    <label value="@init(vm.message)"/>

    <label value="@init(123)"/>

    <checkbox checked="@init(true)"/>
```

@init at Form Binding

We can also use the initial binding on the form binding to fill in predefined value.

```
<div      form="@id('fx') @init(vm.defaultOrder) @load(vm.order) @save(vm.order, before='

</div>
```

We'll describe more advanced usage with `@init` in section ZK Developer's Reference/MVVM/DataBinding/Form Binding.

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

Command Binding

Overview

Command binding is a mechanism for hooking up a UI component's event such as button's onClick to ViewModel's Command[1] without writing code. This binding is established by `@command` with command's name in a ZUL. We can only bind one event to only one Command but can bind multiple events to the same Command.

Command's name is the name of ViewModel's method with Java annotation `@Command` by default. You can also specify a Command's name in `@Command` 's element.

Command method example

```
public class OrderVM {

    // create and add a new order to a list
    // command name is set to method name by default
    @Command
    public void newOrder(){
        Order order = new Order();
        getOrders().add(order);
        selected = order;//select the new one
    }

    // save an order
    // command name is specified
    @Command('save')
    public void saveOrder(){
        orderService.save(selected);
    }
}
```

Command binding example

```
    <toolbar>
        <button label="New" onClick="@command('newOrder')" />
        <button label="Save" onClick="@command('save')" />
    </toolbar>

    <menubar>
        <menu label="Order">
            <menupopup >
                <menuitem label="New"  onClick="@command('newOrder')"/>
```

```
                        <menuitem label="Save"  onClick="@command('save')"/>
                    </menupopup>
                </menu>
            </menubar>
```

- When clicking "Save" button or menuitem, the binder will invoke `saveOrder()` in a ViewModel.

You can pass parameters to a command method. Please refer here.

Empty Command

If we specify command name with an **empty string literal** or evaluation result of EL inside `@command()` is **null**, the binder will ignore that command. It's handy if you want to do nothing in some cases. For example:

Empty command

```
<button onClick="@command(empty vm.selection?'add':'')"/>

<button onClick="@command(empty vm.selection?'save':null"/>
```

Command Execution

A command execution is a mechanism of ZK Bind where it performs a method call on the ViewModel. It binds to a component's event and when a binding event comes, binder will follow the life-cycle to complete the execution. There are 6 phases in the `COMMAND` execution: `VALIDATION`, `SAVE-BEFORE`, `LOAD-BEFORE`, `EXECUTE`, `SAVE-AFTER`, `LOAD-AFTER`.

Saving and Loading in Command Execution

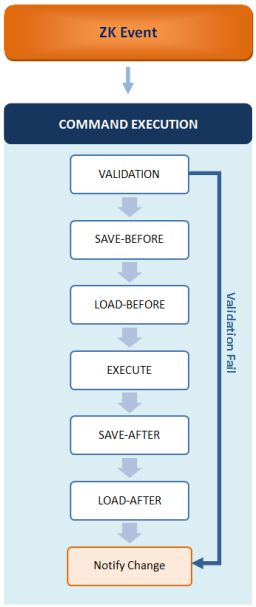
You could save multiple values to ViewModel at the same time before or after the `EXECUTE` phase (i.e., call the ViewModel's command method) by using the `@save(expression, before|after='a-command')` syntax (use this syntax, the value will not be saved immediately after being edited by user). You can also load values to a component before or after the `EXECUTE` phase by using the `@load(expression, before|after='a-command')` syntax.

Validation in Command Execution

Validation is also included in the command execution. It is performed in the `VALIDATION` phase before any other phases. If there are multiple save bindings that depend on the same command, all validators of binding will be called in the `VALIDATION` phase. If a validator said invalid; the execution will be broken, and ignored in the remaining phases.

Phases of Command Execution

Following is the phases of a Command Execution:



- When a bound ZK event enters the binder, the `COMMAND` phase will be invoked and all phases within the `COMMAND` phase will start to execute one by one
- In the `VALIDATION` phase, binder first collects all the properties that needs to be verified. Then, it calls each validator of save-binding that is related to this command. In each call to a validator, binder provides a new `ValidationContext` which contains the main property and other collected properties. This means, you can do dependent validation with collected properties, for example, checking whether the shipping date is larger than the creation date. If any validator reports invalid by calling `ValidationContext.setInvalid ()`, binder ignores all other phases and loads any other properties that has been notified for a change, for example by calling `Binder.notifyChange()`.
- In the `SAVE-BEFORE` phase, binder calls all the save-binding that is relative to the command and mark "before" to save the value to the expression
- In the `LOAD-BEFORE` phase, binder calls all the load-binding that is relative to the command and mark "before" to load the value from expression to the component
- In the `EXECUTE` phase, binder calls the command method of the ViewModel. For example, if the command is "saveOrder", it will try to find a method that has annotation `@Command('saveOrder')` or a method which is named `saveOrder()` with `@Command()` of the viewModel, and then call it. If there is no method to execute, it complains with an exception.
- In the `SAVE-AFTER` phase, binder calls all the save-binding that is relative to the command and mark "after" to save the value to the expression
- In the `LOAD-AFTER` phase, binder calls all the load-binding that is relative to the command and mark "after" to load the value from expression to component

References

[1] ZK Developer's Reference/MVVM/ViewModel/Commands

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

Property Binding

Two Way Data Binding

Property binding makes developers bind any component's attribute to ViewModel's property and specify its access privilege. There are 3 kinds of access privilege: save-load (`@bind`), save only (`@save`) and load only(`@load`).

We usually use save only binding on input components, e.g., textbox, to collect user input into a JavaBean and load only binding when displaying data. If you have both needs, you can use save-load binding. We also call it **Non-conditional Property Binding**, because you cannot specify conditional in this binding.

Save-load binding

```
<window id="window" title="maximize test" border="normal"
                  maximizable="true" maximized="@bind(vm.maximized)">

</window>
```

- In 2 way (save-load) binding, when window's maximized attribute is changed, it will save value back to ViewModel's "maximized" property, and vice versa.

Save only binding

```
<textbox value="@save(vm.person.firstname)"/>
<textbox value="@save(vm.person.lastname)"/>
<intbox value="@save(vm.person.age)"/>
```

- In save-only binding, only textbox's value change will be saved to ViewModel. ViewModel's change won't be loaded to textbox.

Load only binding

```
<label value="@load(vm.welcomeMessage)" />
```

- In load-only binding, only ViewModel's property change will be load to label.

The timing of saving a component attributes' value to ViewModel is **when the attribute related event fires**. For example, textbox's value is saved on the onChange event firing. After you finish your input in a textbox and move the cursor to next input field, the input data is saved to ViewModel at that time. The timing of loading is specified by `@NotifyChange` .

Conditional Binding

Sometimes we need to control the timing of saving or loading instead of depending upon the attribute related event firing. We can specify property `before` or `after` to control the timing upon ViewModel's Command. Therefore, only before or after executing specified Command, the attribute's value is saved (or loading) to a ViewModel.

Assume that there is an order management application with a database. The following zul example is one of its page, listbox displays order list and doublebox is for editing an order detail.

Save before a command

```
<listbox model="@load(vm.orders)" selectedItem="@bind(vm.selected)" hflex="tr
      <template name="model" var="item">
            <listitem >
                  <listcell label="@load(item.id)"/>
                  <listcell label="@load(item.quantity)"/>
                  <listcell label="@load(item.price)" />
            </listitem>
      </template>
</listbox>
<toolbar>
      <button label="New" onClick="@command('newOrder')" />
      <button label="Save" onClick="@command('saveOrder')" disabled="@bind(em
      <!-- show confirm dialog when selected is persisted -->
      <button label="Delete" onClick="@command(empty vm.selected.id?'deleteOr
      disabled="@load(empty vm.selected)" />
</toolbar>
      <grid hflex="true" >
            <columns>
                  <column width="120px"/>
                  <column/>
            </columns>
            <rows>
                  <row>Quantity
                        <intbox value="@load(vm.selected.quantity) @save(vm.s
                  </row>
                  <row>Price
                        <doublebox value="@load(vm.selected.price) @save(vm.s
                        format="###,##0.00"/>
                  </row>
            </rows>
      </grid>
```

- The listcell and doublebox are both bound to `vm.selected.price` . (line 6,26)
- If doublebox's value is saved immediately after user input, listcell's label which is also bound to the same ViewModel's property will also change. This effect might mislead the user that the value has been saved to database.
- To eliminate this misleading effect, developer might hope to batch save all editing result after the user click a "Save" button. We can achieve this by specifying Command's name for property `before` in `@save` . The value of intbox and doublebox are batch-saved when the use clicks "Save" button. (line 23)

Execution Order

If a component has non-conditional property and command binding at the same time. The execution order is :

1. Save binding
2. Execute the command
3. Load binding

A component with multiple binding

```
<textbox value="@bind(vm.filter)" onChange="@command('doSearch')"/>
```

- When onChange event fires, the binder saves vm.filter first, executes command "doSearch", then loads vm.filter.

Multiple Conditions

We also can specify multiple Command's name in an array of string literal for property `before` or `after` like following:

Load after multiple commands

```
<label value="@load(vm.person.firstname, after={'update','delete'})"/>
```

When we use property binding to collect user input, we might need to validate it before saving to a ViewModel. ZK provides a standard validation mechanism through a reusable element called **validator**. We'll describe its detail here.

Collection Binding

We need collection binding when we bind a container component's "model" attribute of , e.g. listbox or grid, to a ViewModel. That target property of ViewModel must be a `Collection` object, e.g. `List` or `Set` . We usually use this binding with <template> and specify its "var" attribute to name the **iteration variable** which represents each object of the collection. ZK automatically set the name of the implicit **iteration status variable** upon the name of iteration variable and this variable stores the index of iteration. E.g. if you set `var="item"` , current iteration index is `itemStatus.index` . If you don't specify iteration variable name in var, their default variable name are **`each`** and **`eachStatus`** .

We'll use the following ViewModel to demonstrate usage of collection binding.

ViewModel for collection binding

```
public class OrderVM {

	//the order list
	private List<Order> orders;

	//the selected order
	private Order selected;

	public List<Order> getOrders() {
		return orders;
	}

	public Order getSelected() {
		return selected;
	}
```

```
	public void setSelected(Order selected) {
		this.selected = selected;
	}
}
```

Binding on Listbox

Collection binding with listbox

```
<listbox model="@load(vm.orders)" selectedItem="@bind(vm.selected)" hflex="true" height="
	<listhead>
		<listheader label="Row Index"/>
		<listheader label="Id"/>
		<listheader label="Quantity" align="center" width="80px" />
		<listheader label="Price" align="center" width="80px" />
		<listheader label="Creation Date" align="center" width="100px" />
		<listheader label="Shipping Date" align="center" width="100px" />
	</listhead>
	<template name="model" var="item">
		<listitem >
			<listcell label="@load(itemStatus.index)"/>
			<listcell label="@load(item.id)"/>
			<listcell label="@load(item.quantity)" style="@load(item.quantity lt 3?
			<listcell label="@load(item.price)"/>
			<listcell label="@load(item.creationDate)"/>
			<listcell label="@load(item.shippingDate)"/>
		</listitem>
	</template>
</listbox>
```

- We bind listbox's model to a Collection type property: `List<Order>` . (line 1)
- We bind "seletedItem" to a ViewModel's property of same object type (`Order`) in a collection. When each time a user select an item of the listbox, binder will save the selected object to ViewModel. Therefore, we can get user's selection by this way. (line 1)
- The value of "var" attribute: item represents an object of the model, so use dot notation to reference its property like `item.quantity` . Its index in the collection can be referenced by `itemStatus.index` . (line 12)
- With power of EL, we can implement simple presentation logic on the ZUL. Here we display quantity number in red color when it's less than 3. (line 14)

Binding on Grid

The usage is similar for grid.

Collection binding usage for grid

```
	<grid model="@load(vm.orders)">
		<columns>
			<column label="Id"/>
			<column label="Quantity"/>
			<column label="Price"/>
```

```
            </columns>
            <template name="model" var="item" >
                  <row>
                        <label value="@bind(item.id)"/>
                        <label value="@bind(item.quantity)"/>
                        <label value="@bind(item.price)"/>
                  </row>
            </template>
      </grid>
```

Dynamic Template

Dynamic template enables developers to specify **which template to apply upon different conditions when rendering a container component**, e.g. grid. It is supported in grid, listbox, combobox, selectbox, and tree. You can use this feature with `@template( [EL-expression] )`. The binder will treat evaluation result of EL expression as a template's name and look for corresponding template to render child components. If the specified template doesn't exist in current component, it looks up parent component's template. If no `@template` assigned, it uses template that named **model** by default.

Default case

```
<grid model="@bind(vm.nodes)">
      <template name="model">
      <!-- child components -->
      </template>
</grid>
```

Template name specified.

```
<grid model="@bind(vm.nodes) @template('myTemplate')">
      <template name="myTemplate">
      <!-- child components -->
      </template>
</grid>
```

You could use EL to decide which template to use.

Conditional

```
<grid model="@bind(vm.nodes) @template(vm.type='foo'?'template1':'template2')">
      <template name="template1">
      <!-- child components -->
      </template>

      <template name="template2">
      <!-- child components -->
      </template>
</grid>
```

It also provides implicit variable: **each** (the instance of item in binding collection) and **eachStatus** (the iteration status) when evaluating the template for each row.

```
<grid model="@bind(vm.nodes) @template(each.type='A'?'template1':'template2')">
      <template name="template1">
      <!-- child components -->
      </template>

      <template name="template2">
      <!-- child components -->
      </template>

</grid>
```

- Assume that the object in binding collection has a property "type". Its value could be A or B. (line 1)

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

Children Binding

Children binding allows us to bind child components to a collection and we can create a group of similar components dynamically upon the collection with <template>. Typically we also can do this by listbox or grid, but they have fixed structure and layout. With this feature we can create child components more flexible, like: a group of checkbox that options comes from a list, or dynamic generated menuitems.

Steps to use this feature:

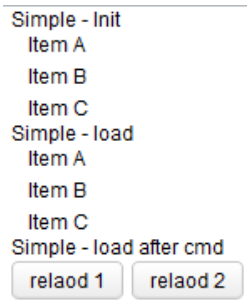
1. Bind a parent component's **children** attribute with `@init` or `@load`
2. Use <template> to enclose child components and set its **name** attribute to **children**. Because children binding chooses the default template with name **children**.
3. Set iteration variable's name in **var** attribute. Then you can use iteration variable to reference each object's property in the collection.

Basic usage example

```
<window apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init('foo.ChildrenSimpl
      Simple - Init
      <vlayout id="init" children="@init(vm.nodes)">
            <template name="children" var="node">
                  <label value="@bind(node.name)" style="padding-left:10px"/>
            </template>
      </vlayout>
      Simple - load
      <vlayout id="load" children="@load(vm.nodes)">
            <template name="children" var="node">
                  <label value="@bind(node.name)" style="padding-left:10px"/>
            </template>
      </vlayout>
```

```
	Simple - load after cmd
	<vlayout id="aftercmd" children="@load(vm.nodes, after='cmd')">
		<template name="children" var="node">
			<label value="@bind(node.name)" style="padding-left:10px"/>
		</template>
	</vlayout>
	<!-- other components -->
</window>
```

Basic usage screenshot



Combine with Dynamic Template

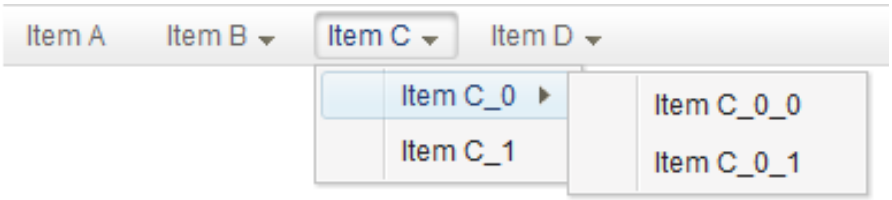
If you combine this feature with dynamic template, you can even render different child components upon different conditions.

Here is an example to create a dynamic menu bar. If a menu item has no sub-menu, we use menuitem or we use menu.

An example of dynamic menu bar

```
	<menubar id="mbar" children="@bind(vm.nodes) @template(empty each.children?'menuite
		<template name="menu" var="node">
			<menu label="@bind(node.name)">
				<menupopup children="@bind(node.children) @template(empty each.ch
			</menu>
		</template>
		<template name="menuitem" var="node">
			<menuitem label="@bind(node.name)" onClick="@command('menuClicked',node
		</template>
	</menubar>
```

Dynamic menu bar screenshot



Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

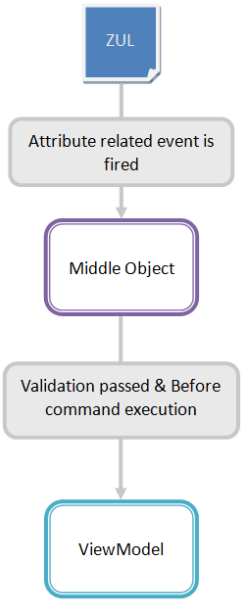
Form Binding

Overview

Form binding is like a cache mechanism. It automatically creates a middle object. Before saving to ViewModel all input data is saved to the middle object. In this way we can keep dirty data from saving into the ViewModel before user confirmation. Assuming a user fills a form in a web application, user input data is directly saved to ViewModel's properties, the target object. Then the user cancel the filling action before submitting the form, thus the data stored in ViewModel is deprecated. That would cause trouble if we further process the deprecated data, so developers might store input data in a middle place first then move to real target object after the user confirms it. Form binding provide a middle object to store unconfirmed input without implementing by yourself.

Form binding can keep target object in ViewModel unchanged before executing a Command for confirmation. Before saving to ViewModel's properties (target object) upon a Command, we can save input in Form binding's middle object. When the command is executed (e.g. button is clicked), input data is really saved to ViewModel's properties. Developers can achieve the whole process easily just by writing ZK bind expression and it reduces developer's burden of cleaning dirty data manually or implementing cached object himself.

The data flow among ZUL, middle object, and the target object is illustrated below:



Steps

1. Give an id to middle object in form attribute with `@id` .

   You can reference the middle object in ZK bind expression with id.
2. Specify ViewModel's property to be loaded with `@load`
3. Specify ViewModel's property to save and before which Command with `@save`

   This means binder will save middle object's properties to ViewModel before a Command
4. Bind component's attribute to middle object's properties like you do in property binding.

   You should use middle object's id specified in `@id` to reference its property.

An example using property binding :

```
<groupbox >
      <grid hflex="true" >
            <columns>
                  <column width="120px"/>
                  <column/>
            </columns>
            <rows>
                  <row>Id
                  <hlayout>
                  <label value="@load(vm.selected.id)"/>
                  </hlayout> </row>
                  <row>Description <textbox value="@bind(vm.selected.description)"/></row
                  <row>Quantity
                        <intbox value="@bind(vm.selected.quantity)"/>
                  </row>
                  <row>Price
                        <doublebox value="@bind(vm.selected.price)" format="###,##0.00" /
                  </row>
            </rows>
      </grid>
</groupbox>
```

- The input data is directly saved to properties of vm.selected. No middle object.

The same example but using form binding:

```
<groupbox form="@id('fx') @load(vm.selected) @save(vm.selected, before='saveOrder')">
      <grid hflex="true" >
            <columns>
                  <column/>
                  <column/>
            </columns>
            <rows>
                  <row>Id
                  <hlayout>
                  <label value="@load(fx.id)"/>
                  </hlayout> </row>
                  <row>Description <textbox value="@bind(fx.description)"/></row>
```

```
                    <row>Quantity
                          <intbox value="@bind(fx.quantity) "/>
                    </row>
                    <row>Price
                          <doublebox value="@bind(fx.price)" format="###,##0.00" />
                    </row>
              </rows>
        </grid>
</groupbox>
```

- We give an id: `fx` to the middle object. (line 1)
- The properties of fx we can access are identical to vm.selected. (line 12,14,17)
- Binder saves input data into fx, middle object, when each time a user triggers an onChange event. When clicking a button bound to Command 'saveOrder', it will save middle object's data to vm.selected.

Middle Object

Form binding automatically creates a middle object for you to store properties from ViewModel's object you specified. But it **only stores those properties which attributes are bound to**. If you still need to access a property that are not stored in the middle object, you can access it from ViewModel's original object. For the example below:

```
<groupbox form="@id('fx') @load(vm.selected) @save(vm.selected, before='saveOrder')">
      <grid hflex="true" >
            <rows>
                  <row>Id <label value="@load(fx.id)"/>
                  </row>
                  <row>Description <textbox value="@bind(fx.description)"/>
                  </row>
                  <row>Quantity <intbox value="@bind(fx.quantity) "/>
                  </row>
            </rows>
      </grid>
</groupbox>
```

- Because we only bind attributes to 3 properties:"id", "description", and "quantity", the middle object only stores these 3 properties. Therefore the "price" property is not stored in middle object, we cannot reference it without binding it first. For example, we cannot pass it as parameter by `@command('cmd', currentPrice=fx.price)` , because `fx.price` doesn't exist in the middle object.

Form Status Variable

Form binding also records middle object's modification status.It's a common requirement for users to know that whether they have modified a form's data (dirty status) or not, developers therefore add a feature that would remind users of this with an UI effect. For example, some text editors appends a star symbol '*' on the title bar to remind users of modified text file. "Form binding" preserves the dirty status in an variable that we can utilize it.

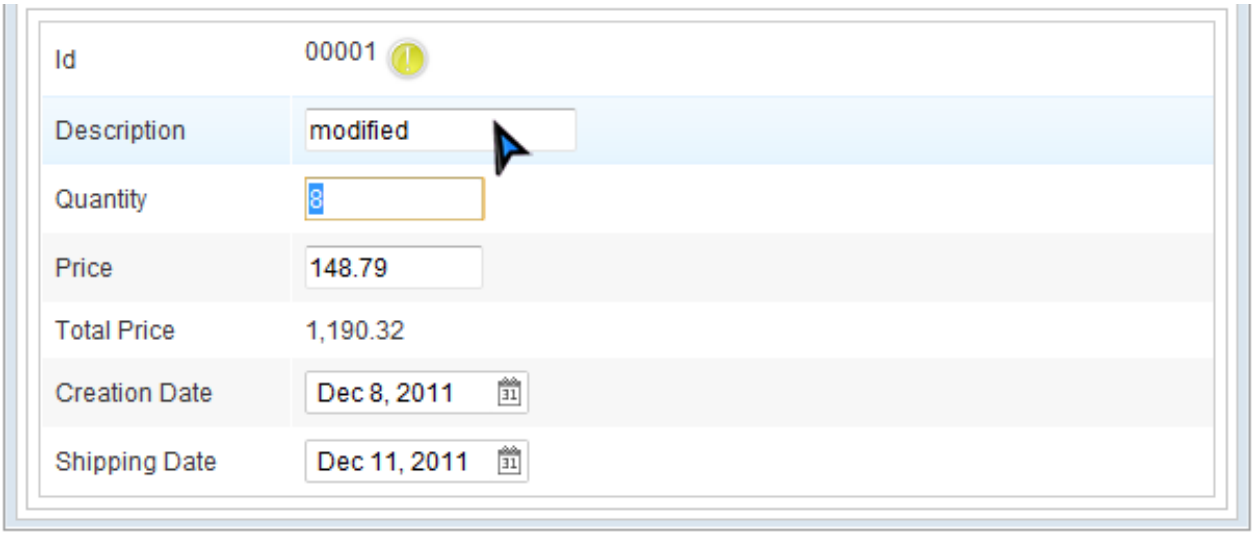
Dirty status is stored in an auto-created **form status variable** with a naming convention of:

`[middleObjectId]Status`

Continue above example, we add an exclamation icon right next to Id value. If users modify any input data, the ex exclamation icon will show up.

```
<groupbox form="@id('fx') @load(vm.selected) @save(vm.selected, before='saveC
      <grid hflex="true" >
            <columns>
                  <column width="120px"/>
                  <column/>
            </columns>
            <rows>
                  <row>
                        Id
                        <hlayout>
                              <label value="@load(fx.id)" />
                              <image src="@load(fxStatus.dirty?'exclamation.p
                        </hlayout>
                  </row>
                  <row>Description <textbox value="@bind(fx.description)"/></
                  <row>Quantity
                        <intbox value="@bind(fx.quantity) "/>
                  </row>
                  <row>Price
                        <doublebox value="@bind(fx.price)" format="###,##0.00
                  </row>
                  <row>Total Price
                        <label value="@load(fx.totalPrice)" />
                  </row>
                  <row>Creation Date
                        <datebox value="@bind(fx.creationDate)"/>
                  </row>
                  <row>Shipping Date
                        <datebox value="@bind(fx.shippingDate)"/>
                  </row>
            </rows>
      </grid>
</groupbox>
```

- In this example, form status variable is `fxStatus` for the form's id is `fx` . Its **dirty** property indicates that whether the form has been modified by users or not.



After users modify a field, an exclamation icon shows up next to “Id” field. If users click “Save” button or change data back to original value, the exclamation icon disappears.

Initialize with Form Object

If you want to gain more control over form binding, e.g. to manipulate the middle object, you can provide an object which implements Form [1] interface (or you can use SimpleForm [2] for convenient) within `@init()` . This will initialize form binding with your own middle object (Form object), then you can do whatever you want like notifying others when middle object's properties change.

Initialize with Form object

```
<groupbox form="@id('fx') @init(vm.myForm) @load(vm.selected) @save(vm.selected, before="
```

Prepare a Form object

```
public class OrderVM {
	Form myForm = new SimpleForm();

	public Form getMyForm() {
		return myForm;
	}

}
```

Form Validation

Before saving data to form's middle object, we also can validate user input with validator. Please refer to ZK Developer's Reference/MVVM/DataBinding/Validator.

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/bind/Form.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/bind/SimpleForm.html#

Converter

Converter performs two way conversion between ViewModel's property and UI component attribute value. It converts data to component attribute when loading ViewModel's property to components and converts back when saving to ViewModel. It's quite common requirement to display data in different format in different context. Using converter developers can achieve this without actually changing the real data in a ViewModel.

Implement a Converter

Developers can create a custom converter by implementing the Converter [1] interface. The method `coerceToUi()` is invoked when loading ViewModel's property to component and its return type should correspond to bound component attribute's value. The `coerceToBean()` is invoked when saving. If you only need to one way conversion, you can leave unused method empty.

The following is how built-in converter 'formatedDate' implement.

```
public class DateFormatConverter implements Converter {
	/**
	 * Convert Date to String.
	 * @param val date to be converted
	 * @param comp associated component
	 * @param ctx bind context for associate Binding and extra
parameter (e.g. format)
	 * @return the converted String
	 */
	public Object coerceToUi(Object val, Component comp, BindContext
ctx) {
		//user sets format in annotation of binding or args when
calling binder.addPropertyBinding()
		final String format = (String)
ctx.getConverterArg("format");
		if(format==null) throw new NullPointerException("format
attribute not found");
		final Date date = (Date) val;
		return date == null ? null : new
SimpleDateFormat(format).format(date);
	}

	/**
	 * Convert String to Date.
	 * @param val date in string form
	 * @param comp associated component
	 * @param ctx bind context for associate Binding and extra
parameter (e.g. format)
```

```
         * @return the converted Date
         */
        public Object coerceToBean(Object val, Component comp,
BindContext ctx) {
              final String format = (String)
ctx.getConverterArg("format");
              if(format==null) throw new NullPointerException("format
attribute not found");
              final String date = (String) val;
              try {
                    return date == null ? null : new
SimpleDateFormat(format).parse(date);
              } catch (ParseException e) {
                    throw UiException.Aide.wrap(e);
              }
        }
}
```

- We retrieve "format" parameter's value by `ctx.getConverterArg("format")` . This allows you to pass in arbitrary parameters.

According to above code, we can pass a "format" parameter to 'formatedDate' converter.

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

Use Custom Converter

The most common way to apply a converter is to bind a component attribute to ViewModel's property which is a custom converter..

Return a converter as a property

```
public class MyViewModel{
      private Converter MyConverter = new MyConverter();

      public Converter getMyConverter(){
            return myConverter;
      }
}
```

Example to use custom converter

```
<label value="@load(vm.message) @converter(vm.myConverter)"/>
```

Use Built-in Converter

ZK have provided some built-in converters that are commonly used. They can be used by **`@converter('converterName', parameterKey='value')`** . Currently, built-in converter we provide are : **formatedNumber, formatedDate**

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

- You should specify number or date pattern in format parameter's value for formatedNumber converter or formatedDate converter.

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/bind/Converter.html#

Validator

User Input Validation

User input validation is an indispensable function of a web application. ZK's **validator** can help developers to accomplish this task. The validator is a reusable element that performs validation and stores validation messages into a validation message holder. When it's applied, it's invoked before saving data to binding target (ViewModel or middle object). When you bind a component's attribute to a validator, binder will use it to **validate attribute's value automatically before saving to a ViewModel or to a middle object**. If validation fails, ViewModel's (or middle object's) properties will be unchanged.

We usually provide custom validator with ViewModel's property.

```
public void class MyViewModel{
	private Validator emailValidator = new EmailValidator();
	public Validator getEmailValidator(){
		return emailValidator;
	}
}
```

Then it can be referenced on a ZUL.

```
<textbox value="@save(vm.account.email) @validator(vm.emailValidator)"/>
```

- Binder use vm.emailValidtor to validate textbox's value before saving to vm.account.email. If validation fails, ViewModel's property: vm.account.email will keep unchanged.

```
<textbox value="@save(vm.account.email, before='save') @validator(vm.emailValidator)"/>
```

- Binder use vm.emailValidtor to validate textbox's value before executing Command 'save'.

Following is a comparison table comparing above two saving syntax:

	Property Binding	Save Before Command
Syntax Example	`@bind(vm.account.email)`	`@load(vm.selected.price)` `@save(vm.account.email, before= 'save')`
Save When	a component's attribute related event fires (e.g. onChange for value)	Before executing a command.
Validation Fails	**Not** save data to ViewModel	**Not** save data to ViewModel & **Not** execute commands
Pros	Immediately validate for single field	Batch save & validate all fields.
Cons	• No validation when executing a command • Save value directly to the bean - might mislead users that an item is persisted	• No immediate validation of single fields after a user input • Lengthy syntax to write for each component

In form binding, you can apply a validator on "form" attribute or an input component. If you apply on both places, you can double validate user input. The first time is when saving data to middle object, this can give a user immediate response after input. The second time is when saving to a ViewModel upon a command, this can validate user input even he doesn't input anything and submit empty data directly.

```
<groupbox form="@id('fx') @load(vm.selected) @save(vm.selected, before='saveOrder') @vali
    <grid hflex="true" >
        <columns>
            <column width="120px"/>
            <column/>
        </columns>
        <rows>
            <row>Id
            <hlayout>
            <label value="@load(fx.id)"/>
            </hlayout> </row>
            <row>Description <textbox value="@bind(fx.description)"/></row>
            <row>Quantity
                <intbox value="@bind(fx.quantity) @validator(vm.quantityValidator)"/>
            </row>
            <row>Price
                <doublebox value="@bind(fx.price) @validator(vm.priceValidator)" format="
            </row>
        </rows>
    </grid>
</groupbox>
```

- You can apply validators on form attribute and each input component respectively. (line 1,14)
- ZK will invoke "vm.formValidator" before executing command **'saveOrder**. (line 1)
- ZK will validate "quantity" when onChange event fires on intbox (when a user blur the focus). (line 14)

Validation Message Holder

ZK provides a standard mechanism to store and display validation message. After performing validation, the validator might store a validation message in **validation message holder**. To use it, you have to initialize it by specifying its id in **validationMessages** attribute with the `@id` . Then you can reference it with this id.

```
<window apply="org.zkoss.bind.BindComposer" viewModel="@id('vm')@init('foo.MyViewModel')"
validationMessages="@id('vmsgs')">
</window>
```

Validation message holder stores messages like a map with key-value pairs. The value is the validation messages that generated by validators after the validation is performed, and the **default key** is the **binding source component (object itself, no id)** that bound to validators. To retrieve and display validation message in a ZUL, you can bind a display component, i.e. label, to validation message holders with a component as the key.

The binder will reload validation messages each time after validation.

Display validation message with default key

```
<window apply="org.zkoss.bind.BindComposer" viewModel="@id('vm')@init('foo.MyViewModel')"
validationMessages="@id('vmsgs')">
<hlayout>Value1:
      <textbox id="tb1" value="@bind(vm.value1) @validator(vm.validator1)" />
      <label id="m1" value="@bind(vmsgs[tb1])"/>
</hlayout>
<hlayout>Value2:
      <intbox id="tb2" value="@bind(vm.value2) @validator(vm.validator2)" />
      <label id="m2" value="@bind(vmsgs[self.previousSibling])"/>
</hlayout>
</window>
```

- You can use component's id to reference a component object. (line 5)
- You can use component's property to reference a component object with relative position that eliminates giving component an id . (line 9)

Implement a Validator

You can create custom validators upon your application's requirement by implementing Validator [1] interface or inheriting AbstractValidator [2] for convenience.

Property Validator

Single Property Validation

We usually need to validate one property at one time, the simplest way is to inherit `AbstractValidator` and override `validate()` to implement a custom validation rule. If validation fails, use `addInvalidMessage()` to store validation messages to be displayed. You have to pass validator bound component object as a key to retrieve this message like we mention in section **Validation Message Holder**.

```
<intbox value="@save(vm.quantity) @validator(vm.rangeValidator)"/>
```

```
public Validator getRangeValidator(){
      return new AbstractValidator() {
```

```
            public void validate(ValidationContext ctx) {
                  Integer val = (Integer)ctx.getProperty().getValue();
                  if(val<10 || val>100){
                        addInvalidMessage(ctx, "value must not < 10 or > 100,
but is "+val);
                  }
            }
      };
}
```

- We can get the user input data from validator2's binding source component by `ctx.getProperty().getValue()` . (line 4)
- addInvalidMessage() will add message into validation message holder. (line 6)

Dependent Property Validation

We sometimes need another property's value to validate the current property. We have to save those values that have dependency among them **upon the same Command** (use `@save(vm.p, before='command' )` , thus binder will pass those properties that are saved upon the same command to `ValidationContext` and we can retrieve them with property's key to process.

Assume shipping date must be at least 3 days later than creation date.

```
<window apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init('eg.ValidationMess
  validationMessages = "@id('vmsgs')">

<grid hflex="true" >
      <columns>
            <column width="120px"/>
            <column/>
      </columns>
      <rows>
            <!-- other input fields -->
            <row>Creation Date
                  <hlayout>
                        <datebox id="cdBox" value="@load(vm.selected.creationDate) @save(
                        @validator(vm.creationDateValidator)"/>
                        <label value="@load(vmsgs[cdBox])" sclass="red" />
                  </hlayout>
            </row>
            <row>Shipping Date
                  <hlayout>
                        <datebox id="sdBox" value="@load(vm.selected.shippingDate) @save(
                        @validator(vm.shippingDateValidator)"/>
                        <label value="@load(vmsgs[sdBox])" sclass="red" />
                  </hlayout>
            </row>
      </rows>
</grid>
</window>
```

- As we save vm.selected.creationDate and vm.selected.shippingDate before Command 'saveOrder', they'll be passed into validation context.

Our custom shipping date validator should get the creation date to compare.

```
public class ShippingDateValidator extends AbstractValidator{

      public void validate(ValidationContext ctx) {
            Date shipping = (Date)ctx.getProperty().getValue();//the
main property
            Date creation =
(Date)ctx.getProperties("creationDate")[0].getValue();//the collected
            //multiple fields dependent validation, shipping date have
to large than creation more than 3 days.
            if(!isDayAfter(creation,shipping,3)){
                  addInvalidMessage(ctx, "must large than creation date
 at least 3 days");
            }
      }

      static public boolean isDayAfter(Date date, Date laterDay , int
day) {
            if(date==null) return false;
            if(laterDay==null) return false;

            Calendar cal = Calendar.getInstance();
            Calendar lc = Calendar.getInstance();

            cal.setTime(date);
            lc.setTime(laterDay);

            int cy = cal.get(Calendar.YEAR);
            int ly = lc.get(Calendar.YEAR);

            int cd = cal.get(Calendar.DAY_OF_YEAR);
            int ld = lc.get(Calendar.DAY_OF_YEAR);

            return (ly*365+ld)-(cy*365+cd) >= day;
      }
}
```

- Shipping date is the **main property** to be validated. The way to retrieve other properties is different the main one. (line 5)

Dependent Property Validator in Form Binding

If you want to validate a property according to another property's value in the form binding, you have to apply a validator in form attribute instead of an input component that bound to a middle object's property. Then you can get all properties from validation context.

The following is the example to demonstrate the same shipping date validator but it's used in form binding.

Using validator in form binding

```
<groupbox form="@id('fx') @load(vm.selected) @save(vm.selected, before='saveOrder') @vali
      <grid hflex="true" >
            <columns>
                  <column width="120px"/>
                  <column/>
            </columns>
            <!-- other components -->
            <rows>
                  <row>Creation Date
                        <hlayout>
                              <datebox id="cdBox" value="@bind(fx.creationDate) @validato
                              <label value="@load(vmsgs[cdBox])" sclass="red" />
                        </hlayout>
                  </row>
                  <row>Shipping Date
                        <hlayout>
                              <datebox id="sdBox" value="@bind(fx.shippingDate)"/>
                              <label value="@load(vmsgs[sdBox])" sclass="red" />
                        </hlayout>
                  </row>
            </rows>
      </grid>
</groupbox>
```

- Apply a validator in form binding, not in individual input component that bound to middle object's property. (line 1)

Validator for form binding

```
      public Validator getShippingDateValidator() {
            return new AbstractValidator(){
                  public void validate(ValidationContext ctx) {
                        Date shipping =
(Date)ctx.getProperties("shippingDate")[0].getValue();
                        Date creation =
(Date)ctx.getProperties("creationDate")[0].getValue();
                        //dependent validation, shipping date have to
later than creation date for more than 3 days.

if(!CaldnearUtil.isDayAfter(creation,shipping,3)){
                              addInvalidMessage(ctx,"must large than
creation date at least 3 days");
```

```
                        }
                    }
                };
        }
```

- There is no main property when applying to a form binding. We should retrieve all properties with its key. (line 4-5)

Pass and Retrieve Parameters

We can pass one or more parameters by EL expression to a validator. The parameters are written in key-value pairs format inside `@validator()` annotation.

Passing a constant value in a ZUL

```
<textbox id="keywordBox"       value="@save(vm.keyword) @validator(vm.maxLengthValidator,
```

Retrieving parameters in a validator

```
      public class MaxLengthValidator implements Validator {

            public void validate(ValidationContext ctx) {
                  Number maxLength =
(Number)ctx.getBindContext().getValidatorArg("length");
                  if (ctx.getProperty().getValue() instanceof String){
                        String value =
(String)ctx.getProperty().getValue();
                        if (value.length() > maxLength.longValue()){
                              ctx.setInvalid();
                        }
                  }else{
                        ctx.setInvalid();
                  }
            }
      }
```

- The parameter's name "length" is user defined.

Passing object in a ZUL

```
<combobox id="upperBoundBox" >
      <comboitem label="90" value="90"/>
      <comboitem label="80" value="80"/>
      <comboitem label="70" value="70"/>
      <comboitem label="60" value="60"/>
      <comboitem label="50" value="50"/>
</combobox>
<intbox value="@save(vm.quantity) @validator(vm.upperBoundValidator, upper=upperBoundBox.
```

- Pass a value from another component's attribute.

Self-defined Validation Message Key

You might want to set the key of a validation message on your own, especially when you use single validator to validate multiple fields. As all validation messages generated by one validator for a component have the same default key (the component object itself), in order to retrieve and display a validation message individually, you have to set different key for each message.

Assume you use only one validator in a form binding to validate all fields.

```
	public Validator getFormValidator(){
		return new AbstractValidator() {

			public void validate(ValidationContext ctx) {
				String val =
(String)ctx.getProperties("value1")[0].getValue();
				if(invalidCondition01(val)){
					addInvalidMessage(ctx, "fkey1", "value1
error");
				}
				val =
(String)ctx.getProperties("value2")[0].getValue();
				if(invalidCondition02(val)){
					addInvalidMessage(ctx, "fkey2", "value2
error");
				}
				val =
(String)ctx.getProperties("value3")[0].getValue();
				if(invalidCondition03(val)){
					addInvalidMessage(ctx, "fkey3", "value3
error");
				}
			}
		};
	}
```

- Because validation messages are stored in the same context, you should use different keys for different messages.

Display validation message aside each component

```
	<vbox form="@id('fx') @load(vm) @save(vm,before='submit') @validator(vm.formValidat
		<hbox><textbox id="t41" value="@bind(fx.value1)"/><label id="141" value="@bin
		<hbox><textbox id="t42" value="@bind(fx.value2)"/><label id="142" value="@bin
		<hbox><textbox id="t43" value="@bind(fx.value3)"/><label id="143" value="@bin
		<button id="submit" label="submit" onClick="@command('submit')" />
	</vbox>
```

Validation in Non-Conditional Property Binding

The execution of a non-conditional property binding is separated from execution of command. If an event triggers both property saving and command execution, they don't block each other.

For example:

```
<textbox  value="@bind(vm.value) @validator(vm.myValidator)" onChange="@command('submit')
```

- When onChange event fires, binder will perform validation before saving vm.value. No matter validation fails or not, it will execute command "submit". Validation failure only stops binder saveing textbox's value into vm.value.

For another opposite example:

```
<textbox id="t1" value="@bind(vm.foo) " onChange="@command('submit')"/>
<textbox id="t2" value="@save(vm.bar, before='submit') @validator(vm.myValidator)"/
```

- When onChange event fires, if t2's value fails in validation, it will stop the binder to execute command "submit". But the binder will still save t1's value to vm.foo.

Use Built-in Validator

ZK provides some built-in validators and you can use it directly without implementing on your own. It's used with syntax `@validator('validatorName')` and you should fill built-in validator's name as a string.

Bean Validator

This validator integrates Java Bean Validation ([3]) framework that defines a metadata model and API [4] to validate JavaBeans. Developers can specify the constraints of a bean's property by Java annotations and validate against the bean by API. By using ZK's bean validator, you only have to specify constraints on bean's properties then bean validator will invoke API to validate for you.

Prepare to Use JSR 303

Required Jars

A known implementation of JSR 303 is Hibernate Validator [5]. The following is a sample dependency list in pom.xml for using Hibernate Validator:

```
<dependency>
	<groupId>org.hibernate</groupId>
	<artifactId>hibernate-validator</artifactId>
	<version>4.0.2.GA</version>
</dependency>
<dependency>
	<groupId>org.slf4j</groupId>
	<artifactId>slf4j-api</artifactId>
	<version>1.6.1</version>
</dependency>
<dependency>
	<groupId>org.slf4j</groupId>
	<artifactId>slf4j-log4j12</artifactId>
	<version>1.6.1</version>
```

```
        </dependency>
```

Hibernate Validator assumes the use of log4j to log information. You can use any log4j implementation you prefer.

Setup

Under project classpath, you need to prepare

1. `log4j.properties`
2. `META-INF/validation.xml`

Here are examples of minimal settings in these files:

`log4j.properties`

```
log4j.rootLogger=info, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] (%F:%L) - %m%n
```

`META-INF/validation.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<validation-config
      xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://jboss.org/xml/ns/javax/validation/configuration">
      <default-provider>org.hibernate.validator.HibernateValidator</default-provider>
      <message-interpolator>org.hibernate.validator.engine.ResourceBundleMessageInterpola

      <traversable-resolver>org.hibernate.validator.engine.resolver.DefaultTraversableRes
      <constraint-validator-factory>org.hibernate.validator.engine.ConstraintValidatorFac
</validation-config>
```

You can customize the setting depending on your requirement.

Usage

Add constraint in JavaBean's property with Java annotation.

```
public static class User{
    private String _lastName = "Chen";

    @NotEmpty(message = "Last name can not be null")
    public String getLastName() {
        return _lastName;
    }

    public void setLastName(String name) {
        _lastName = name;
    }
}
```

Use this validator with its name **beanValidator**.

```
<window id="win" apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init(foo.MyVie
    <textbox id="tb2" value="@bind(vm.user.lastName) @validator('beanValidator')" />
</window>
```

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/bind/Validator.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/bind/validator/AbstractValidator.html#
[3] http://jcp.org/en/jsr/detail?id=303 JSR 303
[4] http://jackson.codehaus.org/javadoc/bean-validation-api/1.0/index.html
[5] http://www.hibernate.org/subprojects/validator.html

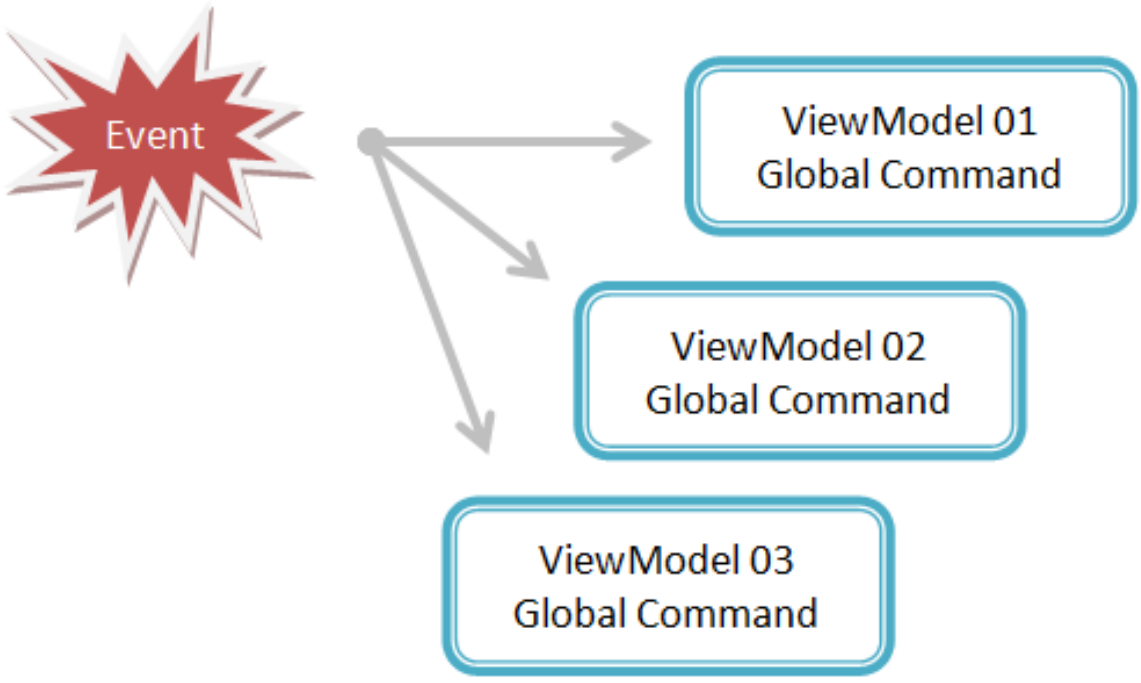
Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

Global Command Binding

Overview

Global Command Binding is similar to command binding but the target becomes a global command. The local command can only be triggered by events of a ViewModel's Root View Component and its child components. The main difference from command binding is the event doesn't have to belong to the ViewModel's root view component or its child component. By default we can bind an event to any ViewModel's global command **within the same desktop**. When we trigger a global command, all matched global command in all ViewModels will be executed.

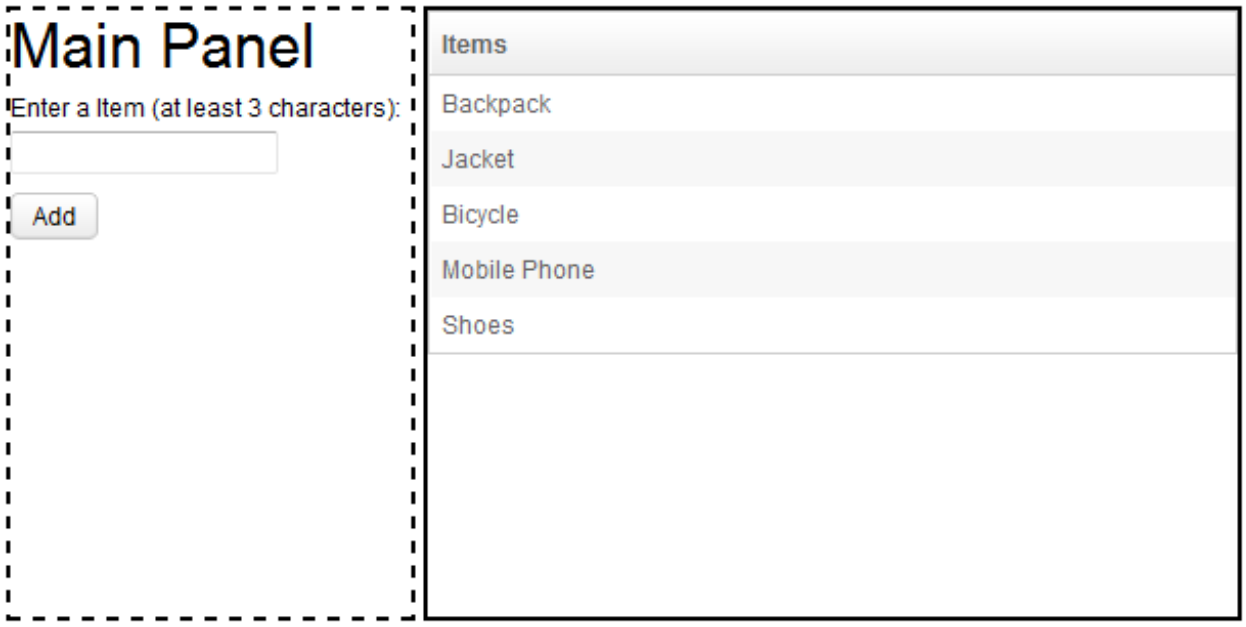


Usage

Notification

Assume we have 2 areas in a page, one is main area for adding items and another is list area for displaying items. Each area is bound to a ViewModel. Two ViewModels can access the same item list from single data source, for simplification we use a static list as data source. The main issue is how to notify another ViewModel to refresh item list after we click the "Add" button (add a new item). We can achieve it by binding "Add" button's onClick event to a global command of list area's ViewModel and in that global command we refresh the item list and notify related properties change to update View.

The interface looks like this:



A zul with 2 ViewModels

```
            <hlayout>
                  <vbox id="mainArea" width="200px" height="300px"
                        style="border:dashed 2px"
visible="@bind(vm.visible)"
                        apply="org.zkoss.bind.BindComposer"
                        viewModel="@id('vm')
@init('org.zkoss.mvvm.examples.globalcommand.AddViewModel')"
                        validationMessages="@id('vmsgs')">
                        <label value="Main Panel" style="font-size:30px" />

                        Enter a Item (at least 3 characters):
                        <textbox id="iBox"
                              value="@load(vm.item)@save(vm.item,
before='add') @validator(vm.itemValidator)" />
                        <label value="@load(vmsgs[iBox])" style="color:red" />
                        <button label="Add"
                              onClick="@command('add')
@global-command('refresh')" />
                        <separator height="20px" />
```

```
                        <label value="@load(vm.msg)" />
                    </vbox>

                    <vbox id="listArea" width="400px" height="300px"
                        visible="@bind(vm.visible)"
apply="org.zkoss.bind.BindComposer"
                        style="border:solid 2px"
                        viewModel="@id('vm')
@init('org.zkoss.mvvm.examples.globalcommand.ListViewModel')">
                        <listbox model="@load(vm.items)">
                            <listhead>
                                <listheader label="Items"/>
                            </listhead>
                            <template name="model">
                                <listitem>
                                    <listcell label="@load(each)"/>
                                </listitem>
                            </template>
                        </listbox>
                        <label value="@load(vm.lastUpdate)" />
                    </vbox>
                </hlayout>
```

- We bind onClick event to a local command "add" and a global command "refresh".

AddViewModel.java

```
public class MainViewModel {

	private String msg;

	@Command @NotifyChange("msg")
	public void add(){
		ItemList.add(item);			//add an item
		msg = "Added "+item;		//update message
	}
	//other code
}
```

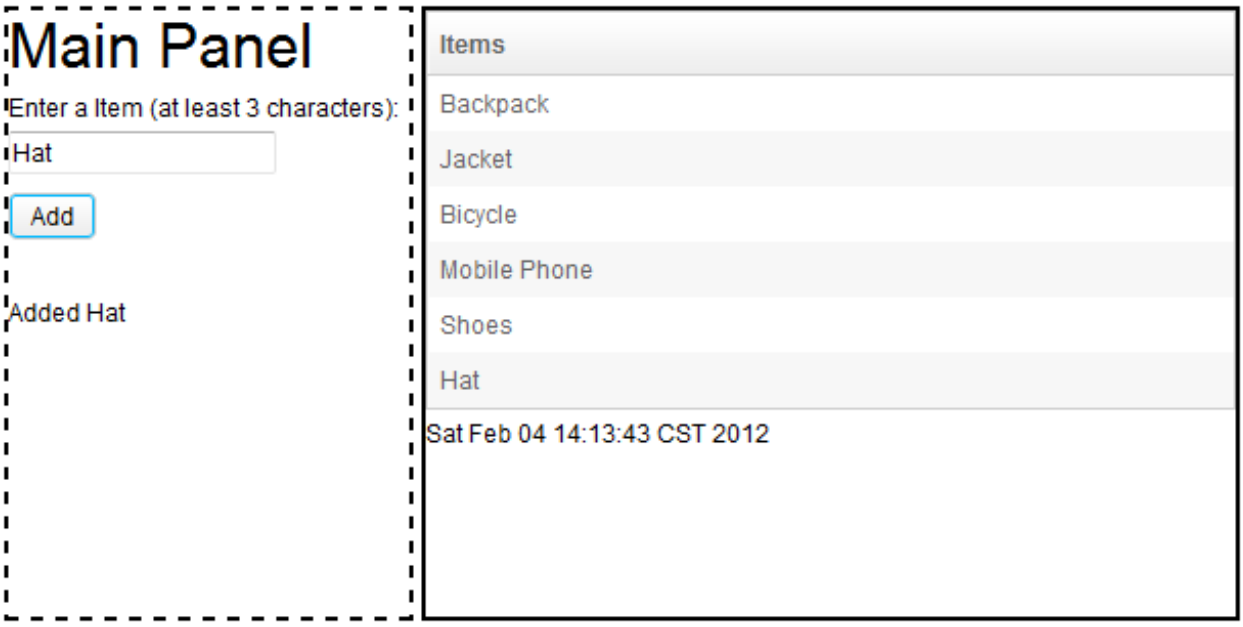
ListViewModel.java

```
public class ListViewModel {

	private List<String> items;
	private Date lastUpdate;

	@GlobalCommand @NotifyChange({"items","lastUpdate"})
	public void refresh(){
		items = ItemList.getList();
		lastUpdate = Calendar.getInstance().getTime();
```

```
        }
        //other code
}
```

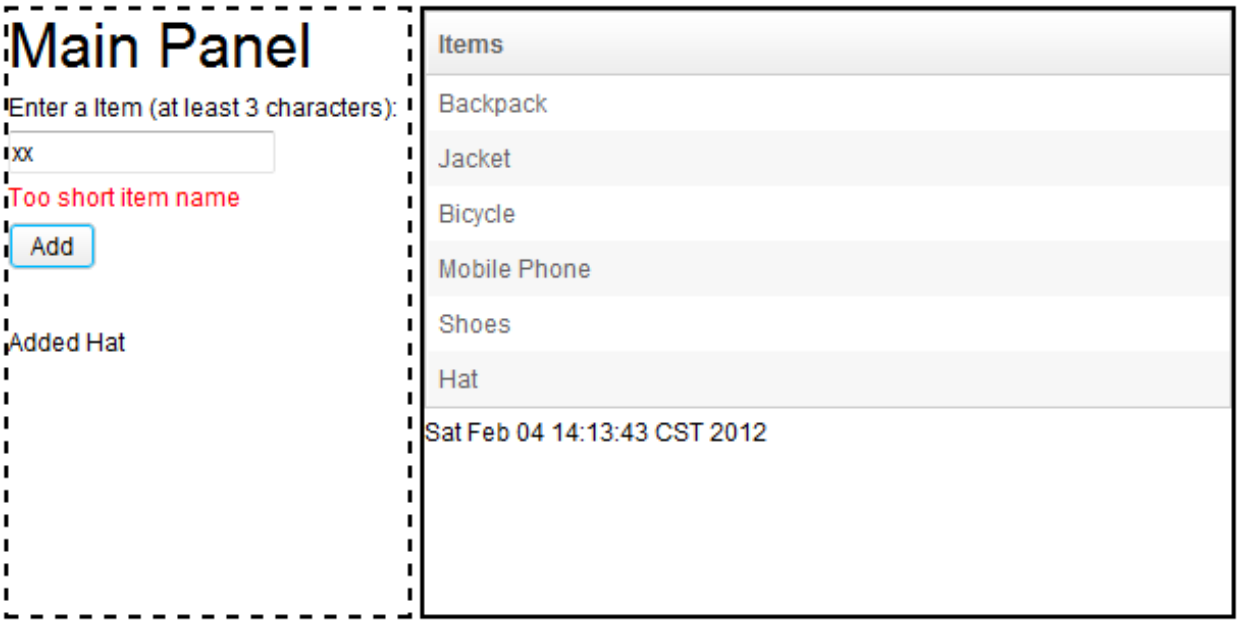
- Declare a global command and notify the change. (line 6)
- The command method reloads items and set current time as last update time. (line 7)

A new item "hat" is added



In above example, we bind onClick event to local and global commands, the global command will always be executed after local command is executed. If the local command is not executed for validation failure, the global command will not be executed. We have a validator to validate item's name, its length can not be less than 3. When we enter a short item name to violate the validation rule, the local command is not executed. So, the global command is not executed, either. We can confirm this behaviour because the message in main area and last update in list area doesn't change.

Validation failure blocks command execution

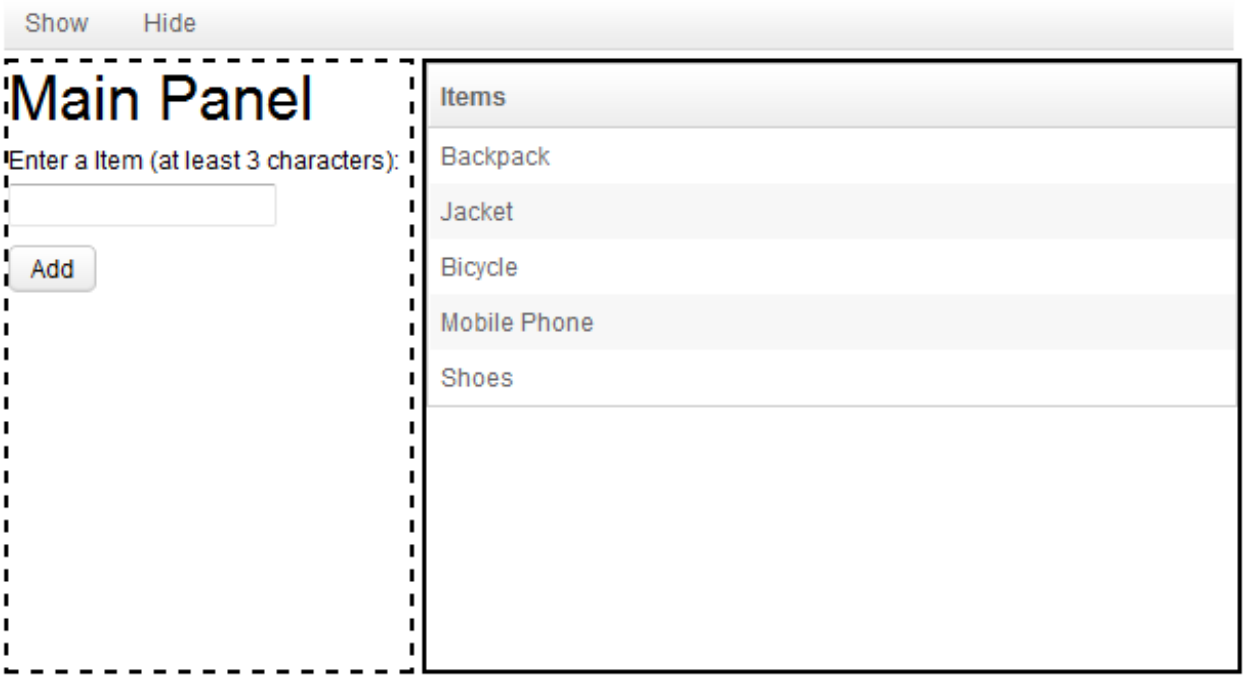


One to Many Communication

We can also use global command binding to communicate multiple ViewModels at one event firing. If an event bound to a global command named "show", all global commands with name "who" in all ViewModels will be executed. Another difference from local command is that you can bind an event to a global command without ensuring its existence. If the global command you bound doesn't exist, it has no response and doesn't throw any exception. The relationship between event and global command is like **publisher-subscriber**. Publisher publishes an event, only those ViewModels who subscribes it execute the corresponding global command.

Based on previous example, assume that we want to hide and show 2 areas simultaneously with menuitems. We can bind a menuitem's onClick event to a global command named "show', and 2 ViewModels implements the "show" command to show itself. So does "hide" command.

The interface is:



One to many communication

```
      <vlayout >
            <menubar width="600px" apply="org.zkoss.bind.BindComposer"
            viewModel="@id('vm')
@init('org.zkoss.mvvm.examples.globalcommand.ControlViewModel')">
                  <menuitem label="Show" onClick="@global-command('show')"></menuitem>
                  <menuitem label="Hide" onClick="@global-command('hide')"></menuitem>
            </menubar>
            <!-- main area-->
            <!-- list area-->
      </vlayout>
```

- Add a menubar with 2 menuitem: show and hide that are bound to global commands.

ViewModel implement show & hide command

```
public class MainViewModel {

      private boolean visible = true;
```

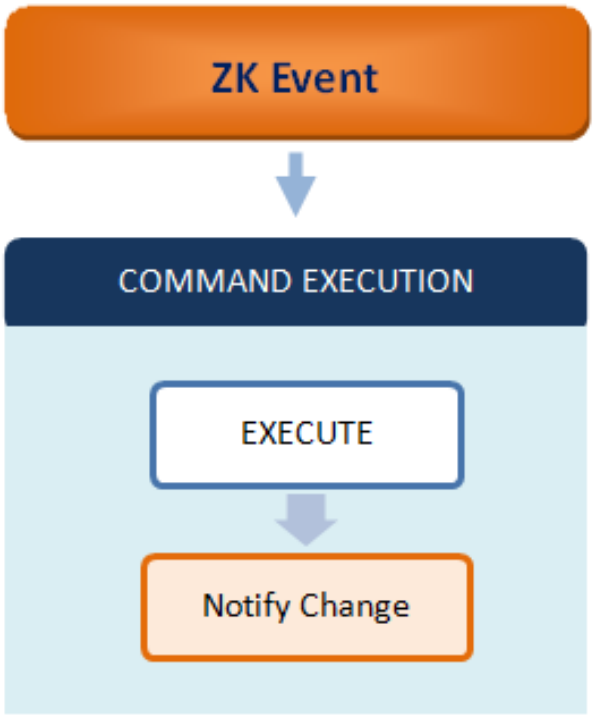
```
	@GlobalCommand @NotifyChange("visible")
	public void show(){
		visible = true;
	}
	@GlobalCommand @NotifyChange("visible")
	public void hide(){
		visible =false;
	}
}
```

- ListViewModel's implementation should have identical command methods.

When clicking hide menuitem, both ViewModel's global command "hide" will be executed thus 2 areas disappear.

Command Execution

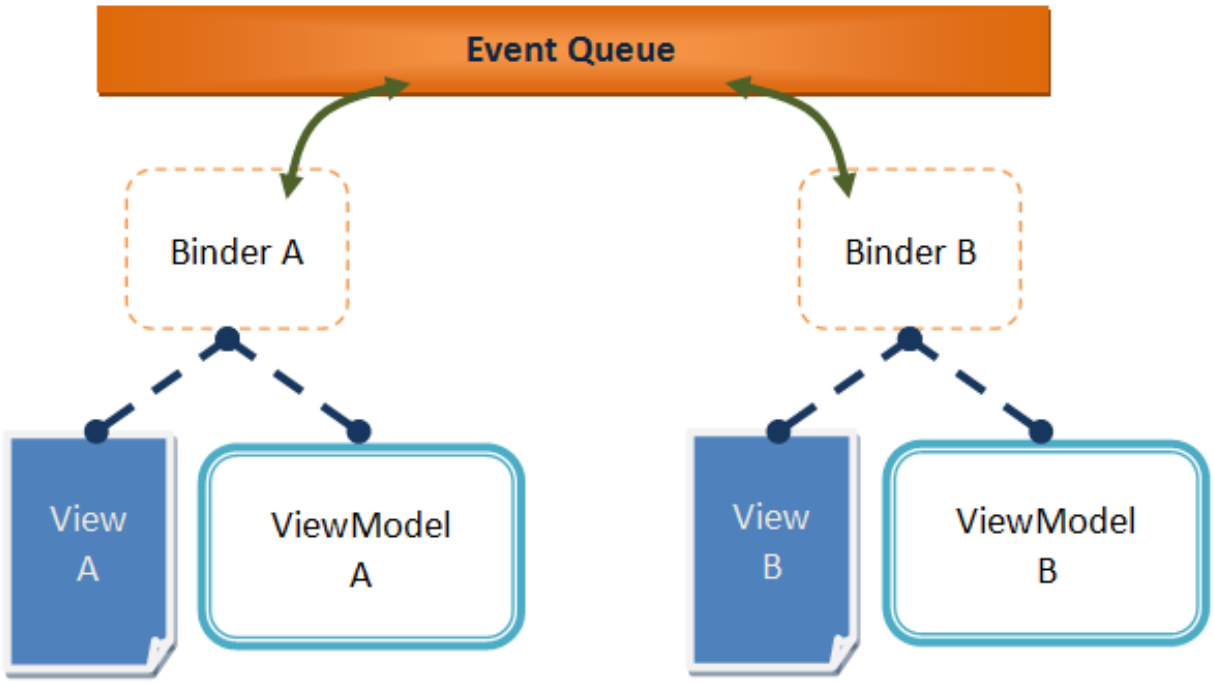
Global command's execution is quite simple. It just executes a command method and then reloads those properties that specified in `@NotifyChange` annotation. But if you bind an event to a local and global command at the same time, binder always **executes local command first**. If any reason blocks the local command's execution, e.g. validation fails, it won't execute the global command.



Background Concept

When we apply a BindComposer [1] to an component, it automatically creates a binder for us. As the ViewModel is a POJO, binder is like a broker to communicate with others. By default all binders subscribe to a default desktop-scoped event queue, thus this is a common communication mechanism among binders.

When we trigger a global command by an event, a binder posts an event to the queue it subscribes. All binders (including the binder that posts the event) that subscribe to the same queue in the same scope will receive this event and try to look up requested global command in their associated ViewModel. If a binder finds a matched global command, it will execute it. Otherwise, it ignores the event without throwing any exception.



In above image, when we trigger a global command "show" in View A, binder A posts an event to the event queue. Binder B which subscribes to the same event queue and binder A itself will both receive the event and try to look up for global command "show" in their associated ViewMoels (ViewModel A and ViewModel B)

ZK allows you to change the name and scope of the event queue a binder subscribes to. Please refer to ZK Developer's Reference/MVVM/DataBinding/Binder.

Trigger a Command Dynamically

Except triggering a global command in a ZUL, we can also do it by calling API. For above example, we can trigger global command in the local command "add" and the code is as follows:

```
@Command @NotifyChange("msg")
public void add(){
    ItemList.add(item);      //add an item
    msg = "Added "+item;     //update message
    BindUtils.postGlobalCommand(null, null, "refresh", null);
}
```

- The first parameter of BindUtils.postGlobalCommand() is queue name, and the second one is queue scope.

You can call this method in a composer, and those pages written in MVC pattern can communicate with ViewModel. A sample code snippet is as follows:

```
public class MyComposer extends SelectorComposer{

	//other codes

	@Listen("onClick=button#postx")
	public void postX(){
		Map<String,Object> args = new HashMap<String,Object>();
		args.put("data", "postX");
		BindUtils.postGlobalCommand("myqueue", EventQueues.DESKTOP,
 "cmdX", args);
	}

}
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

Advance

We'll talk about some advanced topic including how to retrieve and manipulate implicit object or UI components. These approaches certainly give application developers more flexibility, but most of them increase the coupling between ViewModel and View. Therefore they weaken the strength of using MVVM pattern. Hence, we only suggest them to experienced ZK user.

Parameters

Retrieve Binding Parameter

ZK allows you to **pass any object or value that can be referenced by EL** on a ZUL to command method through command binding annotation. Your command method's signature should have a corresponding parameter that is annotated with `@BindingParam` with the same type and key.

The syntax is:

@command('commandName', ***[arbitratyKey]=[EL-expression]*** **)**

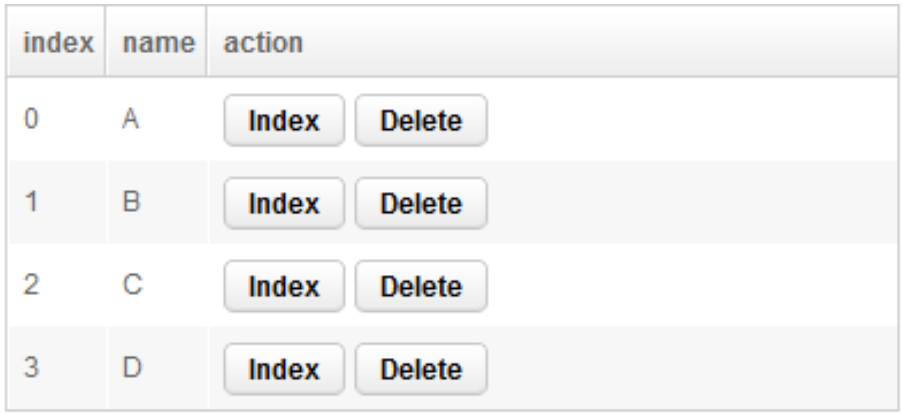
@global-command('commandName', ***[arbitratyKey]=[EL-expression]*** **)**

[arbitratyKey]=[EL-expression]

The expression is optional. It's used only when you want to pass parameters to command method.

A Local Command Example

Assume we have data in grid. It's common to put a button at the end of a row to manipulate it, like delete or update. We usually need index or domain object of a row to perform a action like update.



Here is the zul of above screenshot:

Example to pass parameters

```
<grid id="outergrid" width="700px" model="@bind(vm.items)">
	<columns>
		<column label="index"/>
		<column label="name"/>
		<column label="action" width="300px"/>
	</columns>

	<template name="model" var="item">
		<row>
			<label value="@bind(itemStatus.index)"/>
			<label value="@bind(item.name)"/>
			<hbox>
				<button label="Index" onClick="@command('showIndex', index=
				<button label="Delete" onClick="@command('delete', item=ite
			</hbox>
		</row>
```

```
            </template>
      </grid>
```

- We retrieve row index (Integer class) by iteration status variable and pass it with key "index".
- We retrieve domain object (Item class) by iteration variable and pass it with key "item".

Command methods in the ViewModel

```
      @Command
      public void showIndex(@BindingParam("index") Integer index) {
            message = "item index: " + index;
      }

      @Command
      public void delete(@BindingParam("item") Item item ) {
            int i = items.indexOf(item);
            items.remove(item);
            message = "remove item index " + i;
      }
```

- Command method showIndex() should have a Integer in its argument list. We also have to specify the key "index" in `@BindingParam` (line 2)
- The same as delete(), it should have a parameter with Item class in its argument list. We also have to specify the key "item" in `@BindingParam` (line 7)

A Global Command Example

Passing parameter in global command binding can share data among ViewModels.

The following code passes selected item to another ViewModel.

```
<button label="Submit" onClick="@command('submit') @global-command('detail', name=vm.sele
```

The global command method receives parameter through `@BindingParam` .

```
      @GlobalCommand
      public void detail(@BindingParam("name")String name){
            //...
      }
```

Parameter Default Value

You could choose not to pass parameter for a command method that has parameters. The parameter will become null if you don't pass it, but you can choose to give it a default value with `@Default` . This annotation can be used with other parameter related annotations.

Specify parameter's default value

```
      //getter and setter
      @Command
      public void showIndex(@BindingParam("index") @Default("0")
Integer index) {
      this.index = index;
      }
```

- We set index's default value to 0.

Example to bind above command

```
            <label value="@bind(vm.index)"/>

            <button label="button01" onClick="@command('showIndex', index=9)"/>
            <button label="button02" onClick="@command('showIndex')"/>
```

- Click button01, the label's value is 9.
- Click button02, the label's value is 0.

You can even pass UI components. This resort can make you manipulate UI components directly but also adds a coupling between ViewModel and View which weaken the strength of MVVM pattern.

Example to pass a UI component

```
            <listbox model="@load(vm.items)" selectedItem="@bind(vm.selected)" hflex="tru
                  <listhead>
                        <listheader label="Name"/>
                        <listheader label="Price" align="center" />
                        <listheader label="Quantity" align="center"  />
                  </listhead>
                  <template name="model" var="item">
                        <listitem onMouseOver="@command('popupMessage', target=self, cont
                              <listcell label="@bind(item.name)"/>
                              <listcell label="@bind(item.price)@converter('formatedNumbe
                              <listcell label="@bind(item.quantity)" sclass="@bind(item.q
                        </listitem>
                  </template>
            </listbox>
```

- We pass listitem by implicit object "self".

Command method to receive UI component

```
      @Command
      public void popupMessage(@BindingParam("target")Component target,
@BindingParam("content")String content){
            //...
      }
```

Retrieve Context Object

We can retrieve a value or implicit objects from various context scopes in **initial methods** (methods with **`@Init`**) and **command methods** (methods with **`@Command`**) by applying parameter related annotation on these method's parameters. We list all available HTTP context objects retrieved by parameters annotations related annotations in sections under Parameters

Retrieve HTTP Context Object

Example to get browser information

```
public class HttpParamVM {

	String headerParam;

	@Init
	public void init(@HeaderParam("user-agent")String browser){
		headerParam = browser;

	}
}
```

You can apply multiple parameter related annotation on one method's parameter, and the binder will retrieve the value in multiple context scopes in specified order. It continues to find in next context scope until it retrieves first non-null object.

Multiple context scope retrieval example

```
	@Init
	public void init(@CookieParam("nosuch")
@HeaderParam("user-agent") String guess){
		cookieParam = guess;
	}
```

- In above example, it searches in HTTP request cookie first. If not found a non-null object, it continue to retrieve in HTTP request header.

Retrieve ZK Context Object

You can also receive ZK context object by `@ContextObject` with various `org.zkoss.bind.annotation.ContextType` including Execution [1], Desktop [9], Session [2], BindContext [1], Binder [1], etc. We list all available context objects you can retrieve by `@ContextObject` in ZK Developer's Reference/MVVM/Syntax/ViewModel/Parameters/@ContextParam

We retrieve current binding source component and ViewModel's view component at initial method and command method.

Example to retrieve ZK context object

```
	//getter and setter
	@Init
	public void init(@ContextParam(ContextType.COMPONENT) Component
component,
			@ContextParam(ContextType.VIEW) Component view) {
```

```
            bindComponentId = component.getId();
            bindViewId = view.getId();
        }

        @Command
        public void showId(@ContextParam(ContextType.COMPONENT) Component
 component,
                    @ContextParam(ContextType.VIEW) Component view) {

            bindComponentId = component.getId();
            bindViewId = view.getId();
        }
```

We create 2 labels that bound to current binding component's id and view component's id.

A zul bound to above ViewModel

```
<vbox id="vbox" apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init('eg.Contex

        <label id="componentId" value="@load(vm.bindComponentId)" />
        <label id="viewId" value="@load(vm.bindViewId)" />
        <button id="cmd" label="cmd" onClick="@command('showId')" />
</vbox>
```

- When the page is loaded, component's and view's id are both "vbox". For init(), its current binding source component is vbox. As we apply @init on vbox, it's always the ViewModel's view component.
- After clicking the button, component's id becomes "cmd" because command binding's binding source component is the button and view's id doesn't change.

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/bind/BindContext.html#

Wire Components

Although the original design principle of MVVM pattern is that ViewModel doesn't have any reference to UI components, ZK provide ways to retrieve UI components on a ZUL in a ViewModel. One is passing components as parameters in binding expression. Another is to wire components by Selector. This way enables you to wire components with `@Wire` like you do in a SelectorComposer [3]. Before doing this, you have to call `Selectors.wireComponents()` manually and pass the root component in initial method.

Example to wire components in a ViewModel

```
public class SearchAutowireVM{

	//UI component
	@Wire("#msgPopup")
	Popup popup;
	@Wire("#msg")
	Label msg;

	@Init
	public void init(@ContextParam(ContextType.VIEW) Component view){
		Selectors.wireComponents(view, this, false);
	}
}
```

- Use `@ContextParam(ContextType.VIEW)` to retrieve root component. (line 11)

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

Access Arguments

When you load a ZUL page using `Executions.createComponents("mypage.zul", args)` or `<include>` and pass arguments. ZK bind annotation EL expression can not reference those arguments directly. The simplest solution is to add an custom attribute to hold arguments for later reference. Let's see an example.

outer.zul

```
<include id="inc" type="typeValue" src="inner.zul"/>
```

- Here we pass an argument named "type" to an included ZUL.

inner.zul

```
      <custom-attributes type="${arg.type}"/>
      <vbox apply="org.zkoss.bind.BindComposer"
            viewModel="@id('vm') @init('foo.ExecutionParamVM')">

            <button id="cmd1" label="cmd1" onClick="@command('cmd1', mytype=type)" />
      </vbox>
```

- We should use a custom attribute (line 1) to hold the argument for later use (line 5).

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

Avoid Tracking

When we create a property binding to a ViewModel's property, a **tracker** creates a corresponding tracking record to maintain this binding relationship. Thus when a binder reads properties from annotation `@NotifyChange` , it knows which attributes to reload upon the tracking records. This tracking task consumes time and memory and impacts application performance. If we have an object whose properties never change during whole application running, it's unnecessary to keep track of this immutable object's properties. We can apply `@Immutable` annotation on this immutable object to reduce the cost of tracking. If we bind a attribute to an immutable object's property, the tracker won't create a corresponding tracking record for it.

Immutable object

```
@Immutable
public class SysDefaultConfig{

}
```

Reference an immutable object

```
<label value="@load(vm.sysDefaultConfig.size)"/>
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

Syntax

In following sections we'll list all syntaxes that can be used in implementing a ViewModel and applying ZK bind annotation.

ViewModel

We'll list all Java annotation that are used in ViewModel class and give the description and example in the following sections.

@Init

Syntax

```
@Init

@Init(superclass=true)
```

Description

Target: method

Purpose: Marker annotation to identify a initial method.

Binder calls the method with this annotation when initializing a ViewModel. **Only one or none** init method is allowed in a ViewModel class. If you set annotation element **superclass** to **true**, the ViewModel's parent class's ini method will be invoked first then child's, and this logic repeat on super class. If a class has no method with @Init, no method will be called (including the super class's).

For example, in class hierarchy A(has init) <- B(has init) <- C(no init) <- D (has init, superclass true). D is the last child class.

- When binder initializes D as a view model, it will call D's initial method.
- When binder initializes C, no method will be called.
- When binder initializes B , it will call As' then B's.
- When binder initializes A, it will call A's.

Note that, if you override parent class's initial method e.g. Parent.m1() <- Child.m1(). Because of Java's limitation, binder still call Child.m1(), and Child.m1() will be called twice. To avoid this, you should set superclass to false of Child.m1() and call super.m1() inside it.

We also can use parameter related annotation on initial method's parameters, please refer to subsections of ZK Developer's Reference/MVVM/Syntax/ViewModel/Parameters.

Example

```
public class MyViewModel{
	@Init
	public void initialize(){
		//initializing
	}
}
```

```
public class FooViewModel{
	@Init(superclass=true)
	public void initialize(){
		//initial method of super class will be called first.
	}
}
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@NotifyChange

Syntax

```
@NotifyChange("anotherProperty")

@NotifyChange({"secondProperty","thirdProperty"})

@NotifyChange("*")

@NotifyChange(".")
```

Description

Target: method (setter or command method)

Purpose: To notify binder one or more properties change.

By default, a property set by binder through setter method will notify this property changed without this annotation. You could use this annotation on setter method to override default notification target. You could also add this annotation on a command method to notify properties that are changed after command execution. To avoid the default notification, use `@NotifyChangeDisabled` on setter method solely. Giving "*" in annotation element means notify all properties in a ViewModel.

Use "." to enforce reloading the instance of the class in where the annotation locates, not an instance's property.

Example

```
public class OrderVM {

	//other code...

	@NotifyChange({"selected","messages"})
	public void setSelected(Order selected) {
		this.selected = selected;
	}

	//action command
	@NotifyChange({"selected","orders","messages"})
	@Command
	public void newOrder(){
		Order order = new Order();
		getOrders().add(order); //add new order to order list
		selected = order;//select the new one
	}
}
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@NotifyChangeDisabled

Syntax

```
@NotifyChangeDisabled
```

Description

Target: setter method

Purpose: Marker annotation to disable default notification behavior.

To disable the default notification when binder sets a property.

Example

```
public class OrderVM {

    //other code...

    @NotifyChangeDisabled
    public void setSelected(Order selected) {
        this.selected = selected;
    }

}
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@DependsOn

Syntax

```
@DependsOn
```

Description

Target: getter method

Purpose: To notify change upon property's dependency.

It has the same function as `@NotifyChange` but inverse meaning. It's used to notify binder that getter method's target property is changed because one or more properties it depends on are changed. It can eliminate distributed `@NotifyChange` annotations in a ViewModel when a calculated property depends on multiple properties.

Example

```
public class FullnameViewModel{
	private String firstname;
	private String lastname;
	public String getFirstname() {
		return firstname;
	}
	public void setFirstname(String firstname) {
		this.firstname = firstname;
	}

	public String getLastname() {
		return lastname;
	}
	public void setLastname(String lastname) {
		this.lastname = lastname;
	}

	@DependsOn({"firstname", "lastname"})
	public String getFullname() {
		return (firstname == null ? "" : firstname)	+ " "
				+ (lastname == null ? "" : lastname);
	}
}
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@Command

Syntax

```
@Command()

@Command("commanName")

@Command({"commanName1", "commandName2"})
```

Description

Target: method

Purpose: To identify a Command method.

The optional annotation's element is a String for command's name and that name is referenced in a ZUL with event-command binding. If it's not provided, the method name is set as the command name by default.

We also can use parameter related annotation on initial method's parameters, please refer to subsections of Parameters for more information.

Example

Method name as command name

```
	@Command
	public void search(){
		items = new ListModelList<Item>();
		items.addAll(getSearchService().search(filter));
		selected = null;
	}
```

Specify command name

```
	@Command("delete")
	public void deleteOrder(){
		getService().delete(selected);		//delete selected
		getOrders().remove(selected);
		selected = null;				//clean the selected
	}
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@GlobalCommand

Syntax

```
@GlobalCommand

@GlobalCommand("commanName")

@GlobalCommand({"commanName1", "commandName2"})
```

Description

Target: method

Purpose: To identify a global command method.

The optional annotation's element is a String for command's name and that name is referenced in a ZUL with global command binding. If it's not provided, method name is set as the command name by default.

We can use parameter related annotation on command method's parameters, please refer to subsections of Parameters for more information.

Example

Method name as command name

```
	@GlobalCommand
	public void show(){
		//...
	}
```

Specify command name

```
	@Command("delete") @GlobalCommand("delete")
	public void deleteOrder(){
		//...
	}
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@Immutable

Syntax

```
@Immutable
```

Description

Target: class

Purpose: Marker annotation to indicate an immutable class.

The properties of an immutable class won't be tracked and thus reduce the resources needed in the application.

Example

```
@Immutable
public class SysConfiguration{

}
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

Parameters

You can retrieve value or implicit objects from various context scopes in **initial methods** (methods with `@Init`) and **command methods** (methods with `@Command`) by applying parameter related annotation on these method's parameters. We'll list all parameter related annotations in the following sections.

@BindingParam

Syntax

```
@BindingParam("keyString")
```

Description

Target: Command method's parameter

Purpose: Tell binder to retrieve this parameter with specified key from binding argument on the ZUL.

The annotation is applied to command method's parameter. It declares the applied parameter should come from binding argument written on the ZUL with specified key.

Example

Command binding that pass parameters

```
            <listbox model="@load(vm.items)" selectedItem="@bind(vm.selected)" hflex="tru
                  <listhead>
                        <listheader label="Name"/>
                        <listheader label="Price" align="center" />
                        <listheader label="Quantity" align="center"  />
                  </listhead>
                  <template name="model" var="item">
                        <listitem onMouseOver="@command('popupMessage', myKey='myValue',
                              <listcell label="@bind(item.name)"/>
                              <listcell label="@bind(item.price)"/>
                              <listcell label="@bind(item.quantity)"/>
                        </listitem>
                  </template>
            </listbox>
```

Command method in ViewModel with binding parameter

```
      @Command
      public void popupMessage(@BindingParam("myKey")String target,
@BindingParam("content")String content){
            //...
      }
```

- The `target` 's value is "myValue", and `content`'s is object item's description property.

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@QueryParam

Syntax

```
@QueryParam("keyString")
```

Description

Target: A method's parameter (initial method and command method)

Purpose: Tell binder to retrieve this parameter with specified key from HTTP request parameters.

The annotation is applied to initial method's (or command method's) parameter. It declare the applied parameter should come from HTTP request parameters with specified key.

Example

Http request parameters is appended at URL like: http://localhost:8080/zkbinddemo/httpparam.zul?param1=abc

```
public class HttpParamVM {

	String queryParam;

	@Init
	public void init(@QueryParam("param1") String parm1){
		queryParam = parm1;

	}
```

- In this example, binder will pass "abc" to parm1.

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@HeaderParam

Syntax

```
@HeaderParam("keyString")
```

Description

Target: A method's parameter (applied on initial and command methods)

Purpose: Tell binder to retrieve this parameter with specified key from the HTTP request header.

The annotation is applied to the initial (or command) method's parameter. It declare the applied parameter should come from HTTP request header with specified key.

Example

```
public class HttpParamVM {

	String headerParam;

	@Init
	public void init(@HeaderParam("user-agent")String browser){
		headerParam = browser;
	}
}
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@CookieParam

Syntax

```
@CookieParam("keyString")
```

Description

Target: A method's parameter (for initial and command methods)

Purpose: Tell binder to retrieve this parameter with specified key from the HTTP request cookie.

The annotation is applied to initial (or command) method's parameter. It declare the applied parameter should come from the HTTP request cookie with specified key.

Example

```
public class HttpParamVM {

	String cookieParam;

	@Init
	public void init(@CookieParam("nosuch") String guess){
		cookieParam = guess;
}
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@ExecutionParam

Syntax

```
@ExecutionParam("keyString")
```

Description

Target: A method's parameter (initial method and command method)

Purpose: Tell binder to retrieve this parameter with specified key from the current execution's attribute.

The annotation is applied to initial (or command) method's parameter. It declare the applied parameter should come from the current execution's attribute with specified key.

Example

Assume we want to pass an object by execution's attribute to an included ZUL that uses a ViewModel: `ExecutionParamVM`.

outer ZUL

```
	<window id="w2">
		<zscript>
			void doClick(){
				org.zkoss.zk.ui.Execution ex =
org.zkoss.zk.ui.Executions.getCurrent();
				ex.setAttribute("param1","abc");

				inc.src = "executionparam-inner.zul";
			}
		</zscript>
		<button label="do include" onClick="doClick()"/>
		<include id="inc" />
	</window>
```

We use annotation to retrieve execution's attribute with key "param1".

```
public class ExecutionParamVM {


	private String param1;

	@Init
	public void init(@ExecutionParam("param1") String param1){
		this.param1 = param1;
	}
	//setter, getter, and others
}
```

executionparam-inner.zul

```
	<vbox apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init('foo.Execution

		<label value="@load(vm.param1)"/>
	</vbox>
```

- The label will display "abc".

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@ExecutionArgParam

Syntax

```
@ExecutionArgParam("keyString")
```

Description

Target: A method's parameter (for initial and command methods)

Purpose: Tell binder to retrieve this parameter with specified key from the current execution's argument.

The annotation is applied to initial (or command) method's parameter. It declare the applied parameter should come from the current execution's argument with specified key.

Example

Assume we want to pass an argument to an included ZUL that uses a ViewModel: `ExecutionParamVM`.

outer ZUL

```
<window >
	<include arg1="foo" src="executionparam-inner.zul"/>
</window>
```

We use annotation to retrieve execution's argument with key "arg1".

```
public class ExecutionParamVM {


	private String arg1;

	@Init
	public void init(@ExecutionArgParam("arg1") String arg1){
		this.arg1 = arg1;
	}
	//setter, getter, and others
```

```
}
```

executionparam-inner.zul

```
      <vbox apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init('foo.Execution

            <label value="@load(vm.arg1)"/>
      </vbox>
```

- The label will display "foo".

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@ScopeParam

Syntax

```
@ScopeParam("keyString")

@ScopeParam(scopes=Scope.APPLICATION, value="keyString")
```

A list of all scopes enumeration:

```
enum Scope {
    COMPONENT, SPACE, PAGE, DESKTOP, SESSION, APPLICATION, // single
scope
    AUTO //find by comp.getAttribute(name,true)
}
```

Description

Target: A method's parameter (for initial and command methods)

Purpose: Tell binder to retrieve a value with specified scope

The default scope: **AUTO** means searching the value from COMPONENT to SPACE, PAGE, DESKTOP, SESSION, APPLICATION one by one automatically until find a non-null value. If you specified the **scopes** element, binder will search the only scope you specified.

Example

```
public class ScopeParamVM {

      @Init
      public void init(@ScopeParam(scopes=Scope.APPLICATION ,
value="config") String sysConfig,
```

```
                    @ScopeParam(scopes=Scope.SESSION,value="user") String
userCredential){
      }
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@SelectorParam

Syntax

```
@SelectorParam("#componentId")

@SelectorParam("tagName")

@SelectorParam(".className")

@SelectorParam(":root")

@SelectorParam("button[label='Submit']")

@SelectorParam("window > button")
```

For selector syntax, please refer: SelectorComposer [3]

Description

Target: A method's parameter (for initial and command methods)

Purpose: To identify that a method's parameter should be retrieved from view component of the binder.

The **value** element is the selector to find components. It uses Selectors [1] to select the components. The base component of the selector is the view component of the binder, the component which uses ViewModel.

If the parameter type is a Collection, binder passes the result directly. Otherwise it passes the first result or null if no result.

Example

```
      <vbox apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init('foo.SelectorP

            <hbox><label id="message" /></hbox>
            <hbox><label /></hbox>
            <hbox><label /></hbox>
            <hbox><label /></hbox>
```

```
            <button id="cmd" label="cmd" onClick="@command('cmd')" />
      </vbox>
```

Example to pass components by selector

```
public class SelectorParamVM {

      @Command
      public void cmd(@SelectorParam("label") LinkedList<Label> labels,
@SelectorParam("#message") Label msg) {
            for (int i = 0; i < labels.size(); i++) {
                  labels.get(i).setValue("Command " + i);
            }
            msg.setValue("msg in command");
      }
}
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

References

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

@ContextParam

Syntax

```
@ContextParam(ContextType.PAGE)
```

Enumeration of all context

```
enum ContextType {
    BIND_CONTEXT,       //BindContext instance
    BINDER,                  //Binder instance
    EXECUTION,             //Execution instance
    COMPONENT,             //Component instance of current binding
    SPACE_OWNER,       //IdSpance instance of spaceOwner of current
component
    VIEW,                    //the view component of binder
    PAGE,                    //Page instance of current component
    DESKTOP,              //Desktop instance of current component
    SESSION,              //Session instance
    APPLICATION,       //Application instance
}
```

Description

Target: A method's parameter (for initial and command methods)

Purpose: Tell binder to pass the context object with specified type.

The annotation is applied to initial (or command) method's parameter. Methods can get various ZK context object like: Page or Desktop by applying annotation on parameters.

Example

Retrieve various context object in a ViewModel

```
    @Init
    public void init(@ContextParam(ContextType.APPLICATION_SCOPE) Map<?, ?>
applicationScope,
            @ContextParam(ContextType.SESSION_SCOPE) Map<?, ?> sessionScope,
            @ContextParam(ContextType.DESKTOP_SCOPE) Map<?, ?> desktopScope,
            @ContextParam(ContextType.PAGE_SCOPE) Map<?, ?> pageScope,
            @ContextParam(ContextType.SPACE_SCOPE) Map<?, ?> spaceScope,
            @ContextParam(ContextType.REQUEST_SCOPE) Map<?, ?> requestScope,
            @ContextParam(ContextType.COMPONENT_SCOPE) Map<?, ?>
componentScope,

            @ContextParam(ContextType.EXECUTION) Execution execution,
            @ContextParam(ContextType.COMPONENT) Component component,
            @ContextParam(ContextType.SPACE_OWNER) IdSpace spaceOwner,
            @ContextParam(ContextType.PAGE) Page page,
```

```
            @ContextParam(ContextType.DESKTOP) Desktop desktop,
            @ContextParam(ContextType.SESSION) Session session,
            @ContextParam(ContextType.APPLICATION) WebApp application,

            @ContextParam(ContextType.BIND_CONTEXT) BindContext
bindContext,
            @ContextParam(ContextType.BINDER) Binder binder) {..}
```

The following is another example.

```
<vbox id="vbox" apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init('eg.Contex

<button id="cmd" label="cmd" onClick="@command('cmd')" >
</vbox>
```

A ViewModel used by above zul

```
public class ContextParamVM{
      @Command
      public void cmd(@ContextParam(ContextType.COMPONENT) Component
component,
                  @ContextParam(ContextType.VIEW) Component view) {
      }
}
```

- In above example, the variable component is a Button object and view is a Vbox object.

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@Default

Syntax

```
@Default("defaultValue")
```

Description

Target: A method's parameter (for initial and command methods)

Purpose: Assign a binding parameter's default value when it's null.

You give annotation element's value with a String, and ZK will cast to the corresponding type of the parameter. You can apply this annotation after other parameter related annotation e.g., @BindingParam. If a parameter retrieved by first annotation is null, it uses the default value specified in this annotation.

Example

Pass parameter from a zul

```
<button id="first" onClick="@command('cmd', arg2=100)" />
<button id="second" onClick="@command('cmd')" />
```

Example to assign default value

```
	@Command
	public void cmd(@Default("false") Boolean arg1,
@BindingParam("arg1") @Default("3") Integer arg2){
		//...
	}
```

- According to above example, if we click first button, binder will pass 100 to arg2. If we click second button, arg2 will be 3.

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

Data Binding

In the following sections, we'll cover syntax of all ZK Bind's annotations used on a ZUL. They are all used in **component's attribute** and have similar format:

Basically, the syntax consists of an annotation with comma-separated key-value pairs. The binder treats value of a key as an EL expression, and sets an EL expression without a key to default key's value. All annotations requires at least default key's value.

- **@*[Annotation]*(*[EvaluateOnce EL-expression]*)**

[Annotation]

It could be one of [**id | init**]

[EvaluateOnce EL-expression]

The EL expression will only be evaluated once when parsing.

When using with @id, we strongly suggest you to use string literal to give ViewModel's id.

- **@*[Annotation]*(*[EL-expression]*)**

[Annotation]

It could be one of [**bind | template**]

[EL-expression]

It could be any ZK allowed EL expression (an internal link to EL) without `${ }` enclosed.

- **@*[Annotation]*(*[EL-expression], [conditionKeyword]=[EvaluateOnce EL-expression]*)**

[Annotation]

It could be one of [**load | save**]

[conditionKeyword]=[EvaluateOnce EL-expression]

This expression is optional unless you want to save or load upon a command.

[conditionKeyword]

It could be one of [**before | after**]

[EvaluateOnce EL-expression]

The evaluation result must be one or more command name.

Command name must correspond to the name specified in Java annotation `@Command` in a ViewModel.

- **@*[Annotation]*(*[EL-expression], [arbitraryKey]=[EL-expression]*)**

[Annotation]

It could be one of [**command | global-command | validator | converter**]

[arbitraryKey]=[EL-expression]

It's key-value pairs basically. You can write multiple key-value pairs with different key names.

An EL expression without key is set to a default key named **"value"** implicitly.

Due to each annotation has different functions, some annotations may ignore key-value pair expression other than default key, e.g.@id.

[arbitraryKey]

It could be any name, it's used as a key for parameter related Java annotation in a ViewModel.

@id

Syntax

`@id(` *[EvaluateOnce EL-expression]* `)`

Description

Target Attribute: viewModel, form, validationMessages

Purpose: To give an id to current binding target which can be used to reference its properties in the binding annotation of child components.

We suggest you to use a string literal in EL expression. Because binder only evaluate this annotation's EL expression once to determine ViewModel's id, and this EL expresion is also used in other child component's ZK bind annotation. If it's not a fixed value, it will cause incorrect evaluation result.

When we use it in "validationMessages" attribute, it gives an id to reference validation message holder.

When we use it in "form" attribute, it gives an id to reference form's middle object. Form status variable's naming is: `[middleObjectId]Status`

Example

Usage in viewModel attribute

```
<window apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init('foo.ChildrenMenuV
  validationMessages = "@id('vmsgs')" >
</window>
```

Usage in form attribute

```
<grid form="@id('fx') @load(vm.user) @save(vm.user,before='register')">
</grid>
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@init

Syntax

`@init(` *[EvaluateOnce EL-expression]* `)`

Description

Target Attribute: any

Purpose: Initialize an attribute's value and no reload during user interaction.

The EL expression in this annotation is only evaluated once when binder parses it. When you use it in "viewModel" attribute, binder will try to resolve the EL expression as a class and create an instance of it.

In "form" attribute, the result of evaluating the EL expression should be a Form [1] object.

For other attributes, binder initializes their value with EL evaluation result.

Example

Usage example in viewModel attribute

```
<window apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init('foo.ChildrenMenuV
</window>
```

Usage example in form attribute'

```
<vbox form="@id('fx') @init(vm.myForm) @load(vm.person) @save(vm.person, before='save')">
</vbox>
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@load

Syntax

`@load(` *[EL-expression], [conditionKeyword]=[EvaluateOnce EL-expression]* `)`

Description

Target Attribute: any (except viewModel, validationMessages)

Purpose: Restrict binder to load data from ViewModel only, not save back

For some attributes that don't save data back to the ViewModel like listbox's model or label's value, you can also write `@bind` or `@load` .

[conditionKeyword]=[EvaluateOnce EL-expression]

This expression is optional unless you want to save or load upon a command.

[conditionKeyword]

It could be one of [**before | after**]

[EvaluateOnce EL-expression]

The evaluation result must be one or more command name.

Command name must correspond to the name specified in Java annotation `@Command` in a ViewModel.

Example

```
<label value="@load(vm.user.id)" />

<label value="@load(vm.user.permission, after='showPermission')" />

<label value="@load(vm.user.permission, after={'showPermission', 'showAll'})" />

<label value="@load(vm.user.action, before='process')" />
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@save

Syntax

`@save(` *[EL-expression], [conditionKeyword]=[EvaluateOnce EL-expression]* `)`

Description

Target Attribute: any save-allowed attributes (except viewModel, validationMessages)

Purpose: Restrict binder to save data to ViewModel only, no loading.

You usually use this syntax when you want to save and load data in different conditions, you should write both `@save` and `@load` in an attribute. You have to use it in form binding to save upon a command.

[conditionKeyword]=[EvaluateOnce EL-expression]

This expression is optional unless you want to save or load upon a command.

[conditionKeyword]

It could be one of [**before | after**]

[EvaluateOnce EL-expression]

The evaluation result must be one or more command name.

Command name must correspond to the name specified in Java annotation `@Command` in a ViewModel.

Example

Basic usage

```
<textbox value="@load(vm.person.name) @save(vm.person.name, before='save')"/>

<textbox value="@load(vm.person.name) @save(vm.person.name, before={'save', 'update'})"/>
```

Saving and loading form attribute'

```
<textbox value="@save(vm.number) @load(vm.number, after='cmd')" />
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@bind

Syntax

`@bind(` *[EL-expression]* `)`

Description

Target Attribute: all (except viewModel, validationMessages, form)

Purpose: Specify that binder can both save and load data

It's like a shortcut annotation that combines both `@save` and `@load` . When your saving and loading don't depend on different condition, it's suggested to use this annotation.

Example

```
<textbox id="t2" value="@bind(vm.user.account)" />

<doublebox id="pbox" value="@bind(vm.selected.price)"/>

<datebox id="sdbox" value="@bind(vm.selected.shippingDate)"/>
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@command

Syntax

`@command(` *[EL-expression], [arbitraryKey]=[EL-expression]* `)`

Description

Target Attribute: event attributes (e.g. onClick, onOK)

Purpose: Specify which command to execute when the event fires

You can pass arbitrary arguments in key-value pairs with comma separated.

[arbitraryKey]=[EL-expression]

> It's key-value pairs basically. You can write multiple key-value pairs with different key names.
>
> An EL expression without key is set to a default key named **"value"** implicitly.
>
> Due to each annotation has different functions, some annotations may ignore key-value pair expression other than default key, e.g.@id.

[arbitraryKey]

> It could be any name, it's used as a key for parameter related Java annotation in a ViewModel.

Example

```
<button label="Save" onClick="@command('saveOrder')" />

<button label="Delete" onClick="@command(empty vm.selected.id?'deleteOrder':'confirmDelet

<button label="Index" onClick="@command('showIndex', index=10, keyword='myKeyword')"/>
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@global-command

Syntax

`@global-command(` *[EL-expression], [arbitraryKey]=[EL-expression]* `)`

Description

Target Attribute: event attributes (e.g. onClick, onOK)

Purpose: Specify which global command to execute when the event fires

If you use this binding with a local command binding, remember that local command is always executed first.

You can pass arbitrary arguments in key-value pairs with comma separated.

[arbitraryKey]=[EL-expression]

It's key-value pairs basically. You can write multiple key-value pairs with different key names.

An EL expression without key is set to a default key named **"value"** implicitly.

Due to each annotation has different functions, some annotations may ignore key-value pair expression other than default key, e.g.@id.

[arbitraryKey]

It could be any name, it's used as a key for parameter related Java annotation in a ViewModel.

Example

```
<button label="Save" onClick="@command('saveOrder') @global-command('refresh')" />

<button label="ShowAll" onClick="@global-command('show')" />

<button label="Index" onClick="@command('showIndex') @global-command('showIndex', index=1
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@converter

Syntax

`@converter(` *[EL-expression], [arbitraryKey]=[EL-expression]* `)`

Description

Target Attribute: any (except viewModel, validationMessages, form, and even attributes)

Purpose: It should be used with `@bind` , `@load` , `@save` . It applies a converter to convert data during transformation between UI components and ViewModel.

The evaluation result of EL expression should be a Converter [1] object. You can append arbitrary arguments in key-value pair with comma separated to pass it to the Converter object. Built-in Converter is referenced by a string literal as its name.

[arbitraryKey]=[EL-expression]

It's key-value pairs basically. You can write multiple key-value pairs with different key names.

An EL expression without key is set to a default key named **"value"** implicitly.

Due to each annotation has different functions, some annotations may ignore key-value pair expression other than default key, e.g.@id.

[arbitraryKey]

It could be any name, it's used as a key for parameter related Java annotation in a ViewModel.

Example

Use built-in converter named formatedNumber

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

Use custom converter

```
<label value="@load(vm.selected.totalPrice) @converter(vm.totalPriceConverter)"/>
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@validator

Syntax

`@validator(` *[EL-expression], [arbitraryKey]=[EL-expression]* `)`

Description

Target Attribute: any (except viewModel, validationMessages, form, and event attributes)

Purpose: It should be used with `@bind` , `@load` , `@save` . It applies a validator to validate data when saving to ViewModel.

The evaluation result of EL expression should be a Validator [1] object. You can append arbitrary arguments in key-value pair with comma separated to pass it to the Validator object. Built-in Validator is referenced by a string literal as its name.

[arbitraryKey]=[EL-expression]

It's key-value pairs basically. You can write multiple key-value pairs with different key names.

An EL expression without key is set to a default key named **"value"** implicitly.

Due to each annotation has different functions, some annotations may ignore key-value pair expression other than default key, e.g.@id.

[arbitraryKey]

It could be any name, it's used as a key for parameter related Java annotation in a ViewModel.

Example

Use built-in validator named beanValidator

```
<window id="win" apply="org.zkoss.bind.BindComposer" viewModel="@id('vm') @init(foo.MyVie
    <textbox value="@bind(vm.user.lastName) @validator('beanValidator')" />
</window>
```

Use custom validator

```
<datebox id="cdbox" value="@bind(fx.creationDate) @validator(vm.creationDateValidator)"/>
```

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

@template

Syntax

`@template(` *[EL-expression]* `)`

Description

Target Attribute: model, children

Purpose: It should be used with `@bind` , `@load` . It determines which template to be used to render child components.

The evaluation result of EL expression should be the name of a template which is used to render child components. The result template's name can be any template defined in the same id space.

Example

Dynamic template upon iteration status variable

```
<combobox model="@bind(item.options ) @template(eachStatus.index eq 0 or eachStatus.index
      <template name="model1" var="option" >
            <comboitem label="@bind(optionStatus)" description="@bind(option)"/>
      </template>
      <template name="model2" var="option" >
            <comboitem label="@bind(optionStatus)" description="@bind(option)"/>
      </template>
</combobox>
```

Recursive usage

```
<vlayout id="vlayout" children="@load(vm.nodes) @template('greenBox')">
      <template name="greenBox" var="node">
            <vlayout style="padding-left:10px; border:2px solid green;" >
                  <label value="@bind(node.name)" />
                  <vlayout
                        children="@load(node.children)
@template('greenBox')" />
            </vlayout>
      </template>
</vlayout>
```

- vm.nodes has a tree-like structure, and node.children is a collection of node.

Version History

Version	Date	Content
6.0.0	February 2012	The MVVM was introduced.

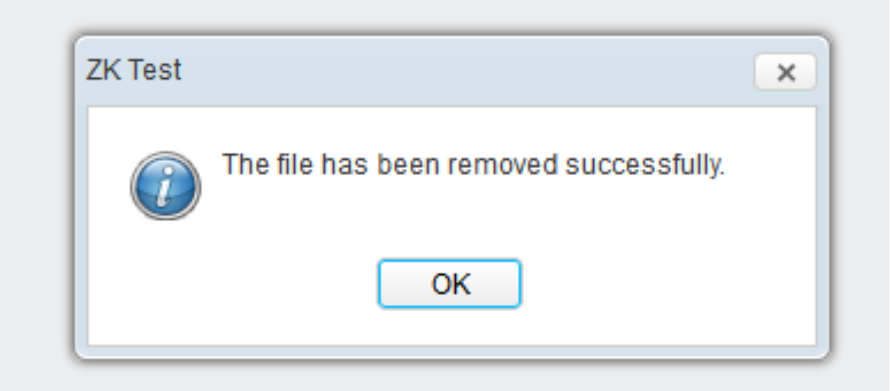
UI Patterns

This section describes feature-specific UI handling topics. For introductory concepts, please refer to the UI Composing section. For detailed information of individual components, please refer to ZK Component Reference.

Message Box

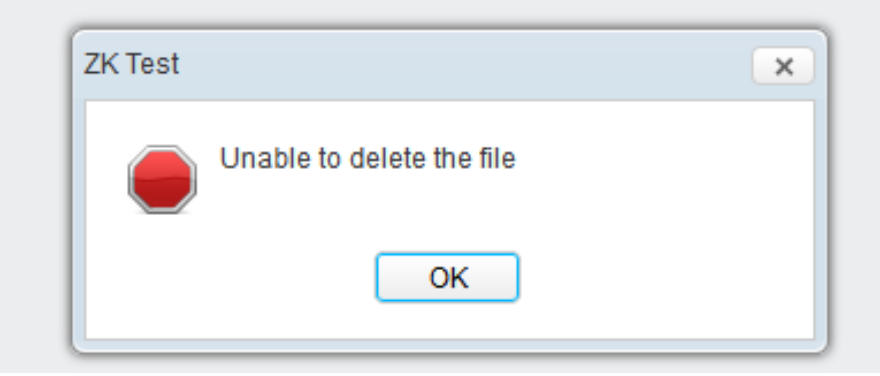
In additions to composing your own window for displaying a message, ZK provide a simple utility: Messagebox [1][2] . For example,

```
Messagebox.show("The file has been removed successfully.");
```



If you could specify a different icon for difference scenario, such as alerting an error:

```
Messagebox.show("Unable to delete the file", null, 0, Messagebox.ERROR);
```



Another typical use is to confirm the users for a decision, such as

```
Messagebox.show("Are you sure you want to remove the file?", null,
   Messagebox.YES+Messagebox.NO, Messagebox.QUESTION,
   new EventListener<MouseEvent>() {
      public void onEvent(MouseEvent event) {
          if (Messagebox.ON_YES.equals(event.getName()))
```

```
                ;//delete the file
            }
        });
```

Notice that the invocation of `show` is returned immediately without waiting for user's clicks[3] .

There are a lot of more utilities, such as the button's order and label. Please refer to ZK Component Reference: Messagebox and Messagebox [1] for more information.

---

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

[2] If you are using zscript, there is a shortcut called alert as follows

```
<button label="Show" onClick='alert("The file has been removed successfully.")'/>
```

[3] If you turned on the use of event thread, the invocation will be stalled until the user clicks a button. It is easier but threading is not cheap. For more information, please refer to the Event Threads section.

Version History

Version	Date	Content

Layouts and Containers

Layouts are components used to partition the display area it owns into several sub-areas for its child components, while containers *group* its child components into the display area it owns.

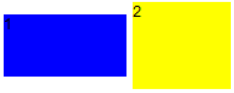
Users are allowed to nest one from another to create desired UI.

Layouts

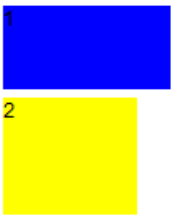
This section provides brief introductions for some of the layout components in ZK. For detailed information and the complete list of layouts, please refer to ZK Component Reference: Layouts.

Hlayout and Vlayout

Hlayout and Vlayout are simple and light-weighted layout components that they arrange its children to be displayed horizontally and vertically respectively. Also, they are easily customizable as they are made up of HTML DIVs.



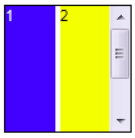

```
<hlayout>
  <div width="100px" height="50px" style="background:blue">1</div>
  <div width="80px" height="70px" style="background:yellow">2</div>
</hlayout>
```



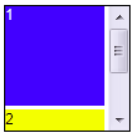

```
<vlayout>
  <div width="100px" height="50px" style="background:blue">1</div>
  <div width="80px" height="70px" style="background:yellow">2</div>
</vlayout>
```

Scrolling

- To make Hlayout and Vlayout scrollable, specify "overflow:auto;" to "style" .
- The height of Hlayout and Vlayout depends on the size of their children, therefore, in order to keep the height of Hlayout and Vlayout constant for the scroll bar to appear, specify a fixed height to Hlayout and Vlayout or place them into a fixed height container, EX: "<window height="100px"...".




```
<font size="8.25">
<hlayout width="100px" height="100px" style="border:1px solid black;overflow:auto;">
      <div width="40px" height="150px" style="background:blue;color:white;">1</div>
      <div width="40px" height="150px" style="background:yellow;">2</div>
</hlayout>
</font>
```




```
<font size="8.25">
<vlayout width="100px" height="100px" style="border:1px solid black;overflow:auto;">
      <div width="80px" height="80px" style="background:blue;color:white;">1</div>
      <div width="80px" height="80px" style="background:yellow;">2</div>
</vlayout>
</font>
```

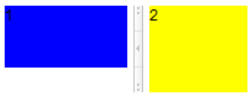
Alignment

Users are allowed to change sclass to control alignment.

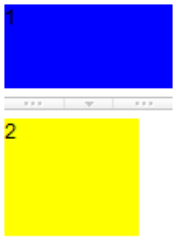

```
<font size="7.99">
<zk>
      <hlayout sclass="z-valign-top">
            <label value="Text:"/>
            <textbox/>
            <window width="50px" height="50px" title="win" border="normal"/>
      </hlayout>
      <separator/>
      <hlayout>
            <label value="Text:"/>
            <textbox/>
            <window width="50px" height="50px" title="win" border="normal"/>
      </hlayout>
      <separator/>
      <hlayout sclass="z-valign-bottom">
            <label value="Text:"/>
            <textbox/>
            <window width="50px" height="50px" title="win" border="normal"/>
      </hlayout>
</zk>
</font>
```

Hbox and Vbox

Similar to Hlayout and Vlayout, Hbox and Vbox arrange its children to be displayed horizontally and vertically respectively. However, they do provide more functionalities such as splitter, align and pack, but their **performance is slower**.



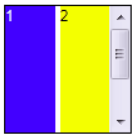
```
<hbox>
  <div width="100px" height="50px" style="background:blue">1</div>
  <splitter collapse="before"/>
  <div width="80px" height="70px" style="background:yellow">2</div>
</hbox>
```



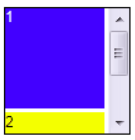
```
<vbox>
  <div width="100px" height="50px" style="background:blue">1</div>
  <splitter collapse="after"/>
  <div width="80px" height="70px" style="background:yellow">2</div>
</vbox>
```

Scrolling

- Hbox and Vbox are created by a table, however, HTML tables are not able to show scroll bars. Hence, to achieve this, users will need to place them in a scrolling container.



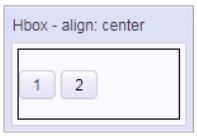
```
<font size="7.84">
<div width="100px" height="100px" style="border:1px solid black;overflow:auto;">
      <hbox>
            <div width="40px" height="150px" style="background:blue;color:white;">1</div
            <div width="40px" height="150px" style="background:yellow;">2</div>
      </hbox>
</div>
</font>
```



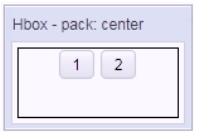
```
<font size="7.84">
<div width="100px" height="100px" style="border:1px solid black;overflow:auto;">
      <vbox>
            <div width="80px" height="80px" style="background:blue;color:white;">1</div>
            <div width="80px" height="80px" style="background:yellow;">2</div>
      </vbox>
</div>
</font>
```

Alignment

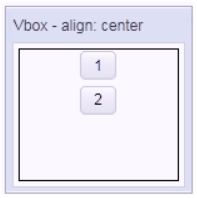
- Users are also allowed to specify align and pack to control alignment.



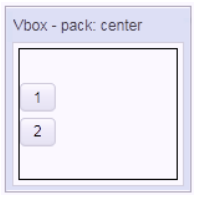

```
<window title="Hbox" border="normal" width="150px" height="100px">
      <caption label="align: center" />
      <hbox width="100%" height="100%" style="border:1px solid black;"
            align="center">
            <button label="1" />
            <button label="2" />
      </hbox>
</window>
```




```
<window title="Hbox" border="normal" width="150px" height="100px">
      <caption label="pack: center" />
      <hbox width="100%" height="100%" style="border:1px solid black;"
            pack="center">
            <button label="1" />
            <button label="2" />
      </hbox>
</window>
```




```
<window title="Vbox" border="normal" width="150px" height="150px">
      <caption label="align: center" />
      <vbox width="100%" height="100%" style="border:1px solid black;"
            align="center">
            <button label="1" />
            <button label="2" />
      </vbox>
</window>
```




```
<window title="Vbox" border="normal" width="150px" height="150px">
      <caption label="pack: center" />
      <vbox width="100%" height="100%" style="border:1px solid black;"
            pack="center">
            <button label="1" />
            <button label="2" />
      </vbox>
</window>
```

For more detailed information, please refer to Hbox and Vbox.

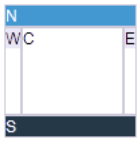
- Users are also allowed to use "cell" to control each cell's alignment.


```
<hbox width="500px">
      <cell style="border:1px solid black;">
            <button label="Help"/>
      </cell>
      <cell style="border:1px solid black;"
            hflex="6" align="center">
            <button label="Add"/>
            <button label="Reomve"/>
            <button label="Update"/>
      </cell>
      <cell style="border:1px solid black;"
            hflex="4" align="right">
            <button label="OK"/>
            <button label="Cancel"/>
      </cell>
</hbox>
```


```
<vbox width="300px" align="stretch">
      <cell style="border:1px solid black;">
            <button label="Help"/>
      </cell>
      <cell style="border:1px solid black;"
            align="center">
            <button label="Add"/>
            <button label="Reomve"/>
            <button label="Update"/>
      </cell>
      <cell style="border:1px solid black;"
            align="right">
            <button label="OK"/>
            <button label="Cancel"/>
      </cell>
</vbox>
```

Borderlayout

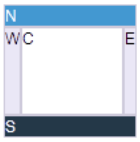
Borderlayout divides its child components into to five areas: North, South, East, West and Center. The heights of North and South are firstly decided, the remainder space is then given to Center as its height. Note that East and West also takes on the height of Center.




```
<borderlayout width="100px" height="100px">
      <north>
            <div style="background:#008db7;color:white;">N</div>
      </north>
      <south>
            <div style="background:#112f37;color:white;">S</div>
      </south>
      <center>
            <div>C</div>
      </center>
      <east>
            <div style="background:#f2f2f2;">E</div>
      </east>
      <west>
            <div style="background:#f2f2f2;">W</div>
      </west>
</borderlayout>
```

flex

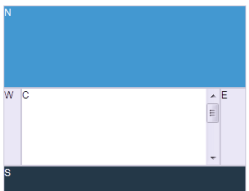
Layout region shares the height of Borderlayout with a distributing sequence of: North, South and Center while the heights of East and West takes on the height of Center. In the previous sample, the div in the layout region does not take up all of layout region's space. In order for the child to occupy the whole area, specify flex="true" in the layout region.




```
<borderlayout width="100px" height="100px">
      <north>
            <div style="background:#008db7;color:white;">N</div>
      </north>
      <south>
            <div style="background:#112f37;color:white;">S</div>
      </south>
      <center>
            <div>C</div>
      </center>
      <east flex="true">
            <div style="background:#f2f2f2;">E</div>
      </east>
      <west flex="true">
            <div style="background:#f2f2f2;">W</div>
      </west>
</borderlayout>
```

Scrolling

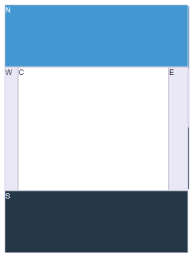
- The height of Center depends on Borderlayout but not on its child, therefore, the height of Center will not be expanded by the growing size of its child components. If Center's height is too short for it's child, Center will cut out the contents of it's child, hence, to avoid this, specify autoscroll="true" to Center in order to assign Center to handle the scrolling.



```
<font size="7.76">
<borderlayout width="300px" height="300px">
      <north>
            <div height="100px"  style="background:#008db7;color:white;">N</div
      </north>
      <south>
            <div height="100px"  style="background:#112f37;color:white;">S</div
      </south>
      <center autoscroll="true">
            <div height="200px">C</div>
      </center>
      <east flex="true">
            <div width="30px" style="background:#f2f2f2;">E</div>
      </east>
      <west flex="true">
            <div width="20px" style="background:#f2f2f2;">W</div>
      </west>
</borderlayout>
</font>
```

Grown by children

- To make Borderlayout dependable on the size of its child components, vflex feature is applied. Specify vflex="min" to each layout region and Borderlayout.



```
<font size="8.19">
<borderlayout width="300px" vflex="min">
      <north vflex="min">
            <div height="100px"  style="background:#008db7;color:white;">N</div>
      </north>
      <south vflex="min">
            <div height="100px"  style="background:#112f37;color:white;">S</div>
      </south>
      <center vflex="min">
            <div height="200px">C</div>
      </center>
      <east flex="true">
            <div width="30px" style="background:#f2f2f2;">E</div>
      </east>
      <west flex="true">
            <div width="20px" style="background:#f2f2f2;">W</div>
      </west>
</borderlayout>
</font>
```

Borderlayout in a container

- Almost all containers' heights depend on their children components, however, the height of Borderlayout does not expand accordingly to the sizes of its children components, therefore, when placing Borderlayout in a container, users have to specify a fixed height in order for Borderlayout to be visible.

```
<zk>
      <window title="win" border="normal">
            <borderlayout height="200px">
                  <north>
                        <div style="background:blue">N</div>
                  </north>
                  <south>
                        <div style="background:blue">S</div>
                  </south>
                  <center>
                        <div>C</div>
                  </center>
                  <east>
                        <div style="background:yellow">E</div>
                  </east>
                  <west>
                        <div style="background:yellow">W</div>
                  </west>
            </borderlayout>
      </window>
</zk>
```

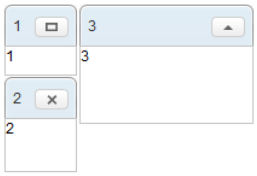
- The default height of Borderlayout is dependent on its parent component, therefore, users can also put Borderlayout in a container with a fixed height.

```
<zk>
	<window title="win" border="normal" height="200px">
		<borderlayout>
			<north>
				<div style="background:blue">N</div>
			</north>
			<south>
				<div style="background:blue">S</div>
			</south>
			<center>
				<div>C</div>
			</center>
			<east>
				<div style="background:yellow">E</div>
			</east>
			<west>
				<div style="background:yellow">W</div>
			</west>
		</borderlayout>
	</window>
</zk>
```

Columnlayout

Columnlayout places its child components into multiple columns while each column allows any numbers of child components placed vertically with different heights (but with the same widths). Unlike portallayout, Columnlayout does *not allow* end users the ability to move child components to different locations at will (although of course, developers are allowed to use the ZK application to re-arrange the order of children components).

Available in ZK PE and EE only [1]

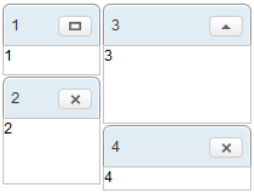



```
<font size="8.46">
<columnlayout>
  <columnchildren width="30%" style="padding: 5px 1px">
    <panel height="60px" title="1" border="normal" maximizable="true">
      <panelchildren>1</panelchildren>
    </panel>
    <panel height="80px" title="2" border="normal" closable="true">
      <panelchildren>2</panelchildren>
    </panel>
  </columnchildren>
  <columnchildren width="70%" style="padding: 5px 1px">
    <panel height="100px" title="3" border="normal" collapsible="true">
      <panelchildren>3</panelchildren>
    </panel>
  </columnchildren>
</columnlayout>
</font>
```

Portallayout

Portallayout places its child components into multiple columns while each column can allow any numbers of child components to be placed vertically with different heights (but with the same widths). Users are also allowed to move any of them to any area desired like that of a portal.

Available in ZK EE only [1]

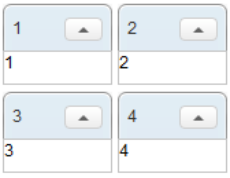



```
<font size="8.44">
<portallayout>
  <portalchildren width="40%" style="padding: 5px 1px">
    <panel height="60px" title="1" border="normal" maximizable="true">
      <panelchildren>1</panelchildren>
    </panel>
    <panel height="90px" title="2" border="normal" closable="true">
      <panelchildren>2</panelchildren>
    </panel>
  </portalchildren>
  <portalchildren width="60%" style="padding: 5px 1px">
    <panel height="100px" title="3" border="normal" collapsible="true">
      <panelchildren>3</panelchildren>
    </panel>
    <panel height="55px" title="4" border="normal" closable="true">
      <panelchildren>4</panelchildren>
    </panel>
  </portalchildren>
</portallayout>
</font>
```

Tablelayout

Tablelayout places its child components in a table. Ths implementation is based on a HTML TABLE tag.

Available in ZK EE only [1]



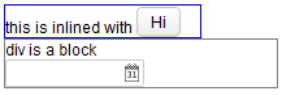

```
<tablelayout columns="2">
  <tablechildren>
    <panel title="1" border="normal"
      collapsible="true" width="80px" height="60px">
      <panelchildren>1</panelchildren>
    </panel>
  </tablechildren>
  <tablechildren>
    <panel title="2" border="normal"
      collapsible="true" width="80px" height="60px">
      <panelchildren>2</panelchildren>
    </panel>
  </tablechildren>
  <tablechildren>
    <panel title="3" border="normal"
      collapsible="true" width="80px" height="60px">
      <panelchildren>3</panelchildren>
    </panel>
  </tablechildren>
  <tablechildren>
    <panel title="4" border="normal"
      collapsible="true" width="80px" height="60px">
      <panelchildren>4</panelchildren>
    </panel>
  </tablechildren>
</tablelayout>
```

Containers

This section provides a brief introduction for some of the container components in ZK. For detailed information and a complete list of containers, please refer to ZK Component Reference: Containers.

Div and Span

Div and span are the most light-weighted containers to group child components. They work the same way as HTML DIV and SPAN tags respectively. Div is a block element that would cause line break for the following sibling i.e. the child and its sibling won't be on the same line (horizontal position). On the other hand, span is an *inline* element which would place the child component and its siblings on the same line (horizontal position).




```
<div style="border: 1px solid blue" width="150px">
  this is
  <span>inlined with <button label="Hi"/></span>
</div>
<div style="border: 1px solid grey">
      <div>div is a block</div>
      <datebox/>
</div>
```

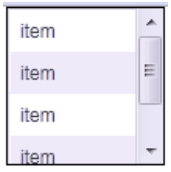
Scrolling

Span:

- Span is an inline element that is not scrollable.

Div:

- To make Div scrollable, specify "overflow:auto;" to "style".
- The height of Div depends on the size of its children, therefore, in order to keep the height of Div constant for the scroll bar to appear, specify a fixed height to Div.




```
<div height="100px" width="100px"
      style="border:1px solid black;overflow:auto;">
      <grid>
            <rows>
                  <row>item</row>
                  <row>item</row>
                  <row>item</row>
                  <row>item</row>
                  <row>item</row>
            </rows>
      </grid>
</div>
```

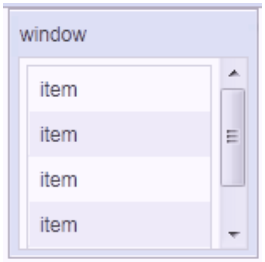
Window

Window is a container providing captioning, bordering, overlapping, draggable, closable, sizable, and many other features. Window is also the owner of an ID space, such that each child component and its IDs are in one independent window so as to avoid the IDs of each child components conflicting with one another.


```
<window title="A" closable="true" sizable="true"
 border="normal" mode="overlapped">
   <div style="background: yellow">1</div>
   <combobox/>
</window>
```

Scrolling

- To make Window scrollable, specify "overflow:auto;" from "contentStyle".
- The height of Window is dependent on the size of its children, therefore, in order to keep the height of Window constant for the scroll bar to appear, specify a fixed height to Window.




```
<window title="window" border="normal"
      height="150px" width="150px"
      contentStyle="overflow:auto;">
      <grid>
            <rows>
                  <row>item</row>
                  <row>item</row>
                  <row>item</row>
                  <row>item</row>
                  <row>item</row>
            </rows>
      </grid>
</window>
```

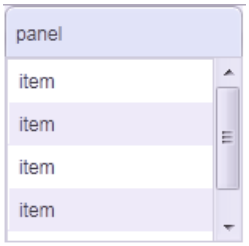
Panel

Like Window, panel is another powerful container supporting captioning, bordering, overlapping and many other features. However, IdSpace [1] is not implemented by this component, therefore, all of its children belongs to the same ID space of its parent.


```
<panel title="A" framable="true" border="normal"
 maximizable="true" collapsible="true">
   <panelchildren>
      <div style="background: yellow">1</div>
      <combobox/>
   </panelchildren>
</panel>
```

Scrolling

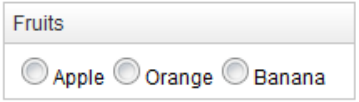
- To make Panel scrollable, specify "overflow:auto;" to "style" of "panelchildren".
- The height of Panel is dependent on the size of its children, therefore, in order to keep the height of the Panel constant for the scroll bar to appear, specify a fixed height to Panel.




```
<panel title="panel" border="normal"
      height="150px" width="150px">
      <panelchildren style="overflow:auto;">
            <grid>
                  <rows>
                        <row>item</row>
                        <row>item</row>
                        <row>item</row>
                        <row>item</row>
                        <row>item</row>
                  </rows>
            </grid>
      </panelchildren>
</panel>
```

Groupbox

Groupbox is a light-weighted way to group child components together. It supports "caption" and "border", however, it does not support overlapping or resizing. Like Panel, IdSpace [1] is not implemented by this component either.

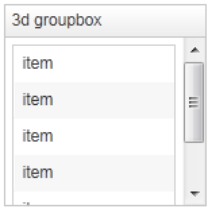



```
<groupbox mold="3d">
  <caption label="Fruits"/>
  <radiogroup>
    <radio label="Apple"/>
    <radio label="Orange"/>
    <radio label="Banana"/>
  </radiogroup>
</groupbox>
```

Scrolling

```
3d mold only
```

- To make Groupbox scrollable, specify "overflow:auto" to "contentStyle".
- The height of the Groupbox depends on the size of its children, therefore, in order to keep the height of the Groupbox constant for the scroll bar to appear, specify a fixed height to Groupbox.




```
<groupbox mold="3d" height="150px" width="150px"
      contentStyle="overflow:auto;">
      <caption label="3d groupbox" />
      <grid>
            <rows>
                  <row forEach="1,2,3,4,5,6">item</row>
            </rows>
      </grid>
</groupbox>
```

Tabbox

Tabbox is a container used to display a set of tabbed groups of components. A row of tabs can be displayed at the top (or left) of the tabbox; users can switch in between each tab group by a simple click. IdSpace [1] is not implemented by this component either.


```
<tabbox height="80px">
  <tabs>
    <tab label="Tab 1"/>
    <tab label="Tab 2"/>
  </tabs>
  <tabpanels>
    <tabpanel>This is panel 1</tabpanel>
    <tabpanel>This is panel 2</tabpanel>
  </tabpanels>
</tabbox>
```

Scrolling

- To make Tabpanel scrollable, specify "overflow:auto;" to "style".
- The height of Tabpanel is dependent on the size of its children, therefore, in order to keep the height of the Tabpanel constant for the scroll bar to appear, specify a fixed height to Tabbox.


```
<tabbox height="100px" width="150px">
      <tabs>
            <tab label="tab" />
      </tabs>
      <tabpanels>
            <tabpanel style="overflow:auto;">
                  <grid>
                        <rows>
                              <row forEach="1,2,3,4,5,6">item</row>
                        </rows>
                  </grid>
            </tabpanel>
      </tabpanels>
</tabbox>
```

Version History

Version	Date	Content

Hflex and Vflex

Hflex (HtmlBasedComponent.setHflex(java.lang.String) [1]) and vflex (HtmlBasedComponent.setVflex(java.lang.String) [2]) indicate the flexibility of the component, which indicates how a component's parent distributes the remaining empty space among its children. Hflex controls the flexibility in the horizontal direction, while vflex in the vertical direction.

Flexible components grow and shrink to fit their given space. Components with larger flex values will be made larger than components with lower flex values, at the ratio determined by the two components. The actual value is not relevant unless there are other flexible components within the same container. Once the default sizes of components in a box are calculated, the remaining space in the box is divided among the flexible components, according to their flex ratios. Specifying a flex value of 0 has the same effect as leaving the flex attribute out entirely.

Fit-the-Rest Flexibility

The simplest use of flex is to have one component to take the rest of the space of its parent (or the page, if it is the root component). For example,

```
<zk>
  <datebox/>
  <div vflex="1" style="background: yellow"/>
</zk>
```

And, the result

Here is another example that we'd like to grow the tabbox to fit the rest of the space:

```
<zk>
   <datebox/>
   <tabbox vflex="1">
      <tabs>
         <tab label="Home"/>
         <tab label="Direction"/>
      </tabs>
      <tabpanels>
         <tabpanel style="overflow: auto">
         <div height="500px" width="100%" style="background: yellow"/>
         </tabpanel>
         <tabpanel>
         </tabpanel>
```

```
      </tabpanels>
   </tabbox>
</zk>
```

Notice you could specify `style="overflow: auto"` in the tabpanel such that the scrollbar will be inside the tabbox rather than the browser window, if the content is too large to fit.

Parent Requires Width/Height

Notice that, if the parent has no predefined height (i.e., its height is decided by this children), the flexible component won't take any space. For example, the inner div (with vflex) in the following example takes no space:

```
<div><!--Wrong! Height required since it is default to be minimal height-->
  <datebox/>
  <div vflex="1" style="background: yellow"/><!--height will be zero since height not spe
</div>
```

To solve it, you have to specify the height in the outer div, such as , , or .

Proportional Flexibility

The absolute value of the vflex/hflex is not that important. It is used to determine the proportion among flexible components. That is, you can give different integers to differentiate child components so they will take space proportionally per the given vflex/hflex value. For example,

```
<div width="200px" height="50px">
      <div style="background: blue" vflex="1" hflex="1"/>
      <div style="background: yellow" vflex="2" hflex="1"/>
</div>
```

And, the result is

Here is another example (hflex):

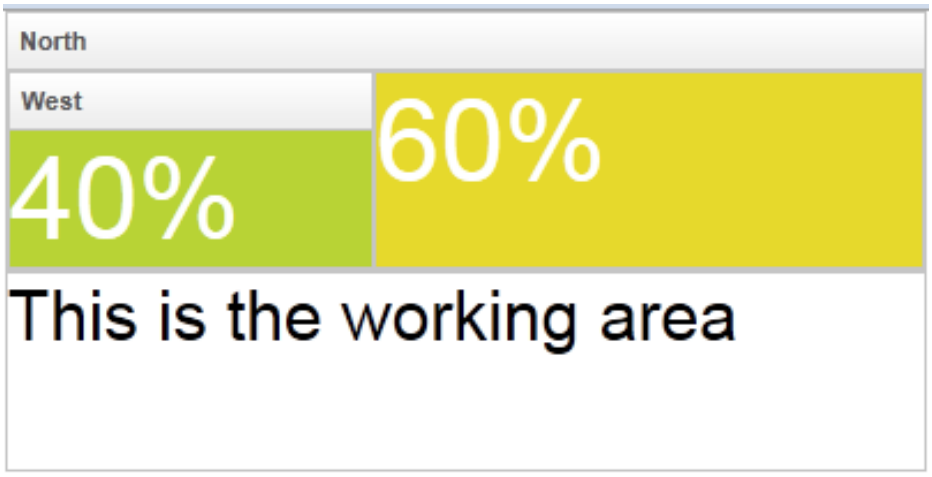
```
<hlayout width="200px">
      <div style="background: blue" hflex="1">1</div>
      <div style="background: yellow" hflex="2">2</div>
```

```
</hlayout>
```

Minimum Flexibility

Sometimes, you might wish that the parent component's size is determined by its children. Or I shall say, the size of the parent component is just high/wide enough to hold all of its child components. We also support that. Just specify vflex/hflex="min".

```
<borderlayout height="200px" width="400px">
      <north title="North" vflex="min">
            <borderlayout vflex="min">
                  <west title="West" size="40%" flex="true" vflex="min">
                        <div style="background:#B8D335">
                              <label value="40%" style="color:white;font-size:50px"/>
                        </div>
                  </west>
                  <center flex="true" vflex="min">
                        <div style="background:#E6D92C">
                              <label value="60%" style="color:white;font-size:50px"/>
                        </div>
                  </center>
            </borderlayout>
      </north>
      <center>
            <label value="This is the working area"
                  style="font-size:30px" />
      </center>
</borderlayout>
```



As you can see, the height of the north region of the outer borderlayout is determined by its child borderlayout. And the height of the inner borderlayout, in this example, is determined by the height of its west child region.

Also notice that the flex property (LayoutRegion.setFlex(boolean) [3]) is unique to borderlayout (north and others). Don't confuse it with hflex or vflex.

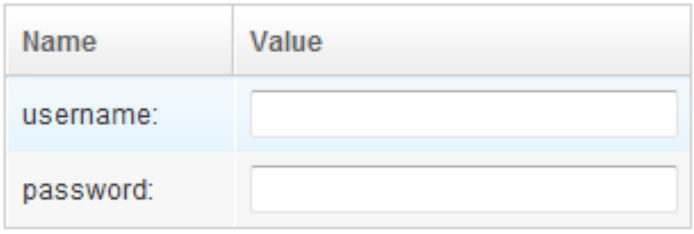
Grid's Column and Flexibility

If hflex is specified in the header of grid, listbox and tree, it is applied to the whole column (including the header and contents).

For example, we could assign 33% to the first column and 66% to the second as follows.

```
<grid width="300px">
	<columns>
		<column label="Name" hflex="1"/>
		<column label="Value" hflex="2"/>
	</columns>
	<rows>
		<row>username:<textbox hflex="1"/></row>
		<row>password:<textbox hflex="1"/></row>
	</rows>
</grid>
```

The result is



Notice that we also specify `hflex="1"` to the textbox, so it will take up the whole space.

Alignment

When we create a form, we will put some input elements in a Grid. We can set hflex="min" to Grid and each Column for keep Grid with minimal size.

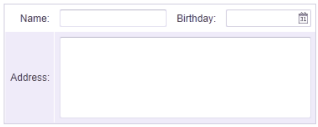

```
<grid hflex="min">
	<columns>
		<column hflex="min" align="right"/>
		<column hflex="min"/>
	</columns>
	<rows>
		<row>
			<label value="Name:"/>
			<textbox/>
		</row>
		<row>
			<label value="Birthday:"/>
			<datebox/>
		</row>
	</rows>
</grid>
```

If we need the Datebox's width the same with Textbox, we can specify hflex="1" to Datebox.


```
<grid hflex="min">
	<columns>
		<column hflex="min" align="right"/>
		<column hflex="min"/>
	</columns>
	<rows>
		<row>
			<label value="Name:"/>
			<textbox/>
		</row>
		<row>
			<label value="Birthday:"/>
			<datebox hflex="1"/>
		</row>
	</rows>
</grid>
```

Cell colspan

Sometimes we need to put some elements in cross column, we can put it in a Cell and set hflex="1" to the element.




```
<grid hflex="min">
	<columns>
		<column hflex="min" align="right" />
		<column hflex="min" />
		<column hflex="min" align="right" />
		<column hflex="min" />
	</columns>
	<rows>
		<row>
			<label value="Name:" />
			<textbox/>
			<label value="Birthday:" />
			<datebox/>
		</row>
		<row>
			<label value="Address:" />
			<cell colspan="3">
				<textbox rows="5" hflex="1"/>
			</cell>
		</row>
	</rows>
</grid>
```

For a complete list of controls that you could apply to the columns of grid, listbox and tree, please refer to ZK Developer's Reference/UI Patterns/Grid's Columns and Hflex.

Flexibility versus Percentage

The use of hflex and vflex is similar to the use of percentage in width and height. For example,

```
<div width="200px" height="200px">
      <div height="33%" style="background: blue">1</div>
      <div height="66%" style="background: yellow">2</div>
</div>
```

The advantage of percentage is that the performance will be a little better, since it is done by the browser. However, hflex and vflex are recommended because of the following issues:

- The use of 100% will cause overflow (and then scrollbar appears if overflow:auto), if padding is not zero. Moreover, some browsers might show mysterious scrollbars or overflow the parent's space even if padding is zero.
- The percentage does *not* work, if any of the parent DOM element does not specify the width or height.
- The percentage does *not* support *take-the-rest-space*. For example, the following doesn't work:

```
<!-- a vertical scrollbar appear (not as expected) -->
<div height="100%">
      <datebox/>
      <div height="100%"/>
</div>
```

Body Height and Padding

By default, ZK's theme configures the document's BODY tag as follows.

```
body {
    height: 100%;
    padding: 0 5px;
}
```

Sometimes you might prefer to add some padding vertically, but it *cannot* be done by changing BODY's styling as follows.

```
body {
    height: 100%;
    padding: 5px; /* WRONG! It causes vertical scrollbar to appear
since the 100% height is used with vertical padding */
}
```

As described in the previous section, a vertical scrollbar will appear, since both the vertical padding and the 100% height are specified.

Solution: you shall *not* change the default CSS styling of BODY. Rather, you could enclose the content with the div component, and then specify vflex="1" and the padding to the div component. For example,

```
<div style="padding: 5px 0" vflex="1">
      <grid>
            <rows>
                  <row>aaa</row>
```

```
            </rows>
      </grid>
</div>
```

Flexibility and Resizing

Vflex and hflex support resizing. If the parent component or the browser window changes its size to increase or decrease the extra space, the child components with vflex/hflex will recalculate themselves to accommodate the new size.

```
<zk>
      <zscript><![CDATA[
            int[] str = new int[100];
            for(int i=0;i<100;i++){
                  str[i]=i;
            }
      ]]></zscript>

      <div height="100%" width="300px">
            Top of the Tree
            <tree vflex="1">
                  <treechildren>
                        <treeitem forEach="${str}" label="item${each}"/>
                  </treechildren>
            </tree>
            <tree vflex="2">
                  <treechildren>
                        <treeitem forEach="${str}" label="item${each}"/>
                  </treechildren>
            </tree>
            Bottom of the Tree
      </div>
</zk>
```

Note that the height proportion between the two trees are always 1 : 2, when we change the browser height.

Limitations

Span Ignores Width and Height

Span ignores the width and height, so hflex and vflex has no effect on them (unless you specify display:block [4] -- but it makes it div eventually).

```
<!-- this example does not work -->
<div width="200px">
      <span style="background: blue" hflex="1">1</span>
      <span style="background: yellow" hflex="2">2</span>
</div>
```

And, the result is as follows - the width has no effect:

This limitation can be solved by the use of hlayout and div as follows.

```
<!-- this is correct -->
<hlayout width="200px">
      <div style="background: blue" hflex="1">1</div>
      <div style="background: yellow" hflex="2">2</div>
</hlayout>
```

Hflex Must Align Correctly

Hflex will be wrong if a component is not aligned in the same *row* with its siblings. For example,

```
<div width="200px">
  <div style="background: blue" hflex="1">1</div><!-- not work since it won't be aligned
  <div style="background: yellow" hflex="2">2</div>
</div>
```

As shown below, the second div is not aligned vertically with the first div, so is the width not as expected:

This limitation can be solved by use of hlayout and div as show in the previous subsection.

Input elements have incorrect margin values in WebKit browsers

In WebKit browsers (Chrome, Safari), the left and right margin values of an input element are considered 2px by browsers, where they are really 0px on screen. This may cause hflex wrongly handle InputElements like textbox, intbox, etc. For example, in the following case the Textbox does not occupy the entire Div width in Chrome:

```
      <div width="300px" style="border: 1px solid green">
            <textbox hflex="1" />
      </div>
```

You can work around this by specifying Textbox margin to be 0:

```
      <style>
            input.nomargin {
                  margin-left: 0;
                  margin-right: 0;
            }
      </style>
      <div width="300px" style="border: 1px solid green">
            <textbox sclass="nomargin" hflex="1" />
      </div>
```

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlBasedComponent.html#setHflex(java.lang.String)
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlBasedComponent.html#setVflex(java.lang.String)
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/LayoutRegion.html#setFlex(boolean)
[4] http://www.quirksmode.org/css/display.html

Grid's Columns and Hflex

This section introduce the usage of ZK auto sizing APIs. All of these can apply to following components :

• Listbox
  • Listhead
  • Listheader
	• Grid
	  • Columns
	  • Column
		• Tree
		  • Treecols
		  • Treecol

Hflex and Width

There are basically two approaches to control the width of a column: width and hflex. They could be specified in the column's header, such as column and listheader.).

While width (HtmlBasedComponent.setWidth(java.lang.String) [1]) specifies the width in precise number (such as width="100px"), hflex (HtmlBasedComponent.setHflex(java.lang.String) [1]) specifies the proportional width (such as hflex="1") or the minimal width (such as hflex="min").

Span

If the total width of all columns is smaller than the grid, there will be some whitespace shown at the right side of the grid.

If you prefer to make each column's width a bit wider to cover the whole grid, you could specify span="true" in grid/listbox/tree (such as Grid.setSpan(java.lang.String) [2]). If you want to make a particular column larger to cover the whole grid, you could specify a number, such as span="2" to expand the third column.

sizedByContent

If you want to make each column as minimal as possible, you could specify hflex="min" for each column. Alternatively, you could specify sizedByContent="true" in the grid/listbox/tree (Grid.setSizedByContent(boolean) [3]). In other words, it implies the width of a column that is *not* assigned with width and hflex shall be minimal.

In general, you will specify span="true" too to cover the whole grid/listbox/tree.

Use Cases

Here we take listbox with listheaders as an example to show some different use cases.

Data Length : Short

```
<zk>
       <zscript><![CDATA[
       String[] msgs2 = {
                    "Application Developer's Perspective",
                    "Server+client Fusion architecture",
                    "Component Developer's Perspective",
                    "Execution Flow of Loading a Page",
                   "Execution Flow of Serving an Ajax Request",
                    "When to Send an Ajax Request"
       };
       ]]></zscript>
      <listbox width="800px">
            <listhead>
                  <listheader label="Product" />
                  <listheader label="Description" />
                  <listheader label="Comment" />
            </listhead>
            <listitem>
                 <listcell><label value="${msgs2[0]}"></label></listcell>
                  <listcell><label value="${msgs2[1]}"></label></listcell>
                  <listcell><label value="${msgs2[2]}"></label></listcell>
            </listitem>
            <listitem>
                  <listcell><label value="${msgs2[3]}"></label></listcell>
                  <listcell><label value="${msgs2[4]}"></label></listcell>
                  <listcell><label value="${msgs2[5]}"></label></listcell>
            </listitem>
      </listbox>
</zk>
```

- **Default** : Data component will show the data correctly, with no width specification. (the exactly width of columns are rendered by browser automatically)

Product	Description	Comment
Application Developer's Perspective	Server+client Fusion architecture	Component Developer's Perspective
Execution Flow of Loading a Page	Execution Flow of Serving an Ajax Request	When to Send an Ajax Request

Proportional Width

Sure you can use the **hflex** we have mentioned in previous section.

```
      <listhead>
                  <listheader label="Product" hflex="1"/>
                  <listheader label="Description" hflex="2"/>
                  <listheader label="Comment" hflex="1" />
       </listhead>
```

Product	Description	Comment
Application Developer's Perspective	Server+client Fusion architecture	Component Developer's Perspective
Execution Flow of Loading a Page	Execution Flow of Serving an Ajax Request	When to Send an Ajax Request

Minimum Flexibility

In the case of **hflex=min**, column's width will be just fitted the contents. As you can see, there might be blank space on the right of the listbox.

```
        <zscript><![CDATA[
       String[] msgs2 = {
                    "Application Developer's Perspective",
                    "Very Short Text",
                   "Server+client Fusion architecture",
                   "Execution Flow of Serving an Ajax Request",
                   "Very Short Text",
                   "When to Send an Ajax Request (Addition Text )"
        };
        ]]></zscript>
    <listbox width="800px">
      <listhead>
                   <listheader label="Product" hflex="min"/>
                   <listheader label="Description" hflex="min"/>
                   <listheader label="Comment" hflex="min" />
       </listhead>
```

Product	Description	Comment	
Application Developer's Perspective	Very Short Text	Server+client Fusion architecture	(Blank Here)
Execution Flow of Serving an Ajax Request	Very Short Text	When to Send an Ajax Request (Addition Text)	

- If you want your contents fill the whole grid to eliminate the blank space, you can set **span=true** to make it proportionally expanded.

```
    <listbox width="800px" span='true'>
      <listhead>
                   <listheader label="Product" hflex="min"/>
                   <listheader label="Description" hflex="min"/>
                   <listheader label="Comment" hflex="min" />
       </listhead>
```

Product	Description	Comment
Application Developer's Perspective	Very Short Text	Server+client Fusion architecture
Execution Flow of Serving an Ajax Request	Very Short Text	When to Send an Ajax Request (Addition Text)

- If you want the rest of space to be assigned to one of the columns, set **span** to a number. The number is 0-based index of columns.

```
    <listbox width="800px" span='0'>
      <listhead>
```

```
                    <listheader label="Product" hflex="min" />
                    <listheader label="Description" hflex="min" />
                    <listheader label="Comment" hflex="min" />
        </listhead>
```

Product	Description	Comment
Application Developer's Perspective	Very Short Text	Server+client Fusion architecture
Execution Flow of Serving an Ajax Request	Very Short Text	When to Send an Ajax Request (Addition Text)

- If you want the size of the Listbox determined by it content, assign **hflex=min** on the Grid, and make sure all the Listheaders either have **hflex=min** or has a fixed **width**.

```
    <listbox width="800px" hflex='min'>
      <listhead>
                    <listheader label="Product" hflex="min" />
                    <listheader label="Description" hflex="min" />
                    <listheader label="Comment" hflex="min" />
       </listhead>
```

Product	Description	Comment
Application Developer's Perspective	Very Short Text	Server+client Fusion architecture
Execution Flow of Serving an Ajax Request	Very Short Text	When to Send an Ajax Request (Addition Text)

Data Length : Long

```
        <zscript><![CDATA[
        String[] msgs2 = {
                    "Application Developer's Perspective",
                  "Very Long Long Long Long Long Long Long Long Long
Long Text",
                   "Server+client Fusion architecture",
                   "Execution Flow of Serving an Ajax Request",
                   "Very Long Long Long Long Long Long Long Long Long
Long Text",
                   "When to Send an Ajax Request"
        };
        ]]></zscript>
```

- **Default** : ZK data component will wrap the text to fit the width of column.

Product	Description	Comment
Application Developer's Perspective	Very Long Long Long Long Long Long Long Long Long Long Long Long Text	Server+client Fusion architecture
Execution Flow of Serving an Ajax Request	Very Normal Normal Text	When to Send an Ajax Request (Addition Text)

Scrollbar

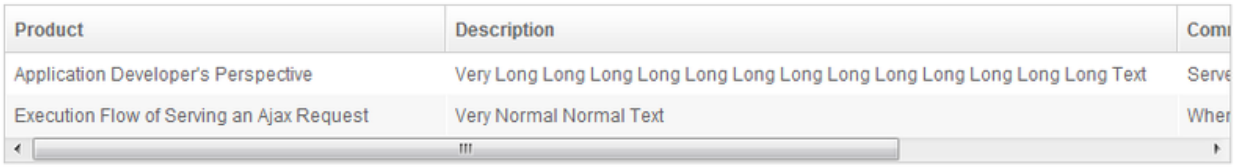
When the sum of content width is larger than Grid width. The scroll appears **if and only if**

1. The Columns and Column component are presented.
2. Each of the Column components is given an as **hflex** or **width** value.

Specify Width

This is simple way to avoid text wrapped by given proper width. However, it can be difficult if you don't know the content length beforehand.

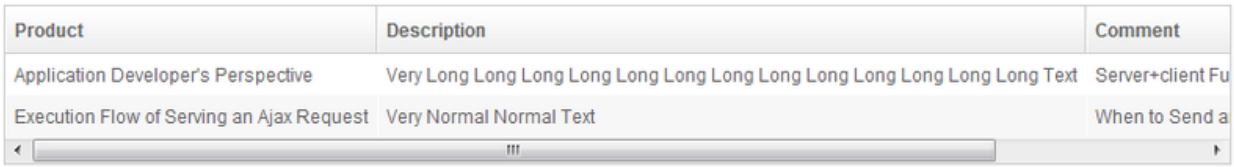
```
<listhead>
            <listheader label="Product" width="250px"/>
            <listheader label="Description" width="470px"/>
            <listheader label="Comment" width="280px" />
 </listhead>
```



Minimum Flexibility

- Set **hflex=min** and ZK will calculate the length of content and set proper width to the column accordingly.
- **Notes**: Remember to set every column with **hflex=min** or specify a specific **width**; otherwise those columns without setting minimum hflex or specifying a width could disappear if not enough space in the listbox.

```
<listhead>
            <listheader label="Product" hflex="min" />
            <listheader label="Description" hflex="min" />
            <listheader label="Comment" hflex="min" />
 </listhead>
```



Mixed Flexibility and width

Width + Hflex proportion

```
<listhead>
            <listheader label="Product" width="120px" />
            <listheader label="Description" hflex="2" />
            <listheader label="Comment" hflex="1" />
 </listhead>
```

Product	Description	Comment
Application Developer's Perspective	Very Long Long Long Long Long Long Long Long Long Long Long Long Text	Server+client Fusion architecture
Execution Flow of Serving an Ajax Request	Very Normal Normal Text	When to Send an Ajax Request (Addition Text)

Width + Hflex min

```
<listhead>
        <listheader label="Product" width="150px" />
        <listheader label="Description" hflex="min" />
        <listheader label="Comment" hflex="min" />
 </listhead>
```

Product	Description	Comment
Application Developer's Perspective	Very Long Long Long Long Long Long Long Long Long Long Long Long Text	Server+client Fusion architecture
Execution Flow of Serving an Ajax Request	Very Normal Normal Text	When to Send an Ajax Request (A

Width + Hflex min + Hflex proportion

```
<listhead>
        <listheader label="Product" width="120px" />
        <listheader label="Description" hflex="min" />
        <listheader label="Comment" hflex="1" />
 </listhead>
```

Product	Description	Comment
Application Developer's Perspective	Very Long Long Long Long Long Long Long Long Long Long Long Long Text	Server+client Fusion architecture
Execution Flow of Serving an Ajax Request	Very Long Long Long Long Long Long Long Long Long Long Long Long Text	When to Send an Ajax Request

Data Length : Dynamic

If users can't control the data size, that means, you should take both the short data and the long data situation into consideration.

Now we have several attributes, rules, and situations, please choose the right solution for your case.

- **(Blank)** : Doesn't matter
- **V** : Must set
- **X** : Must not set
- **!** : Acceptable but something need notice.

Specification	Span=true	Hflex proportion	Hflex=min	Specific Width
Fill whole Grid	One of below attribute was set		X	!
No Scrollbar Grid		V	X	!
No Content Wrapping Column			V	!
Fill whole Grid + No Scrollbar Grid		V	X	!
Fill whole Grid + No Content Wrapping	V	All Column Component Should have one of these attributes		

- ! : Specific width must not more than grid's width.

Version History

Version	Date	Content
5.0.6	February 2011	New specification of hflex and span.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlBasedComponent.html#setWidth(java.lang.String)
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Grid.html#setSpan(java.lang.String)
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Grid.html#setSizedByContent(boolean)

Tooltips, Context Menus and Popups

The support of tooltips, context menus and popups are generic. Any components inherited from XulElement [1] can handle them in the same way.

To enable any of them, you have to prepare a component representing the customized look, and then specify this component or its ID in the corresponding properties (such as XulElement.setTooltip(java.lang.String) [2]) in the target component. Then, when the user triggers it (such as moving the mouse over the target component), the component representing the customized look is shown up.

The component representing the customized look must be a Popup component or one of derives, such as Menupopup, while the target component (which causes the customized look to show up) can be any kind of component.

What	Condition	API
Tooltips[3]	When the user moves the mouse point over the target component for a while	XulElement.setTooltip(java.lang.String) [2] or XulElement.setTooltip(org.zkoss.zul.Popup) [4]
Context Menus	When the user clicks the *right* button on the target component	XulElement.setContext(java.lang.String) [5] or XulElement.setContext(org.zkoss.zul.Popup) [6]
Popups	When the user clicks the *left* button on the target component	XulElement.setPopup(java.lang.String) [7] or XulElement.setPopup(org.zkoss.zul.Popup) [8]

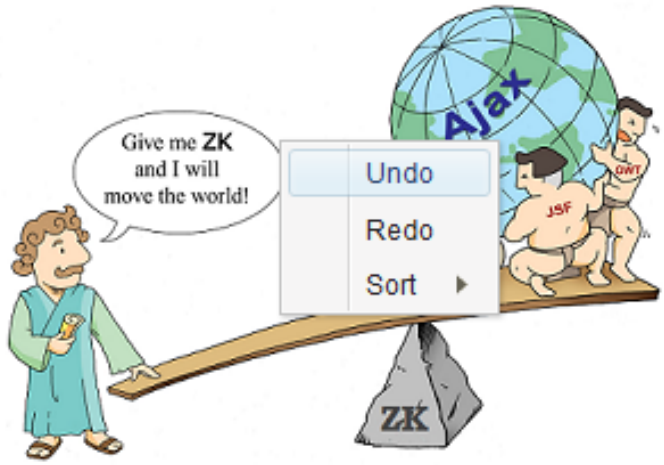
[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/impl/XulElement.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/impl/XulElement.html#setTooltip(java.lang.String)
[3] Notice that if you'd like to have different text for the tooltip (rather than a fully customized look), you shall use HtmlBasedComponent.setTooltiptext(java.lang.String) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlBasedComponent.html#setTooltiptext(java.lang.String)) instead (which is easier to use).
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/impl/XulElement.html#setTooltip(org.zkoss.zul.Popup)
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/impl/XulElement.html#setContext(java.lang.String)
[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/impl/XulElement.html#setContext(org.zkoss.zul.Popup)
[7] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/impl/XulElement.html#setPopup(java.lang.String)
[8] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/impl/XulElement.html#setPopup(org.zkoss.zul.Popup)

Tooltips

To provide a custom tooltip, you could specify the ID of the custom tooltip in the target component's `tooltip` (XulElement.setTooltip(java.lang.String) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zul/ impl/ XulElement. html#setTooltip(java. lang. String)) or XulElement.setTooltip(org.zkoss.zul.Popup) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zul/ impl/ XulElement. html#setTooltip(org. zkoss. zul. Popup))). For example,

```
<zk>
    <image src="/img/earth.png" tooltip="msg"/>

    <menupopup id="msg">
        <menuitem label="Undo"/>
        <menuitem label="Redo"/>
        <menu label="Sort">
            <menupopup>
                <menuitem label="Sort by Name" autocheck="true"/>
                <menuitem label="Sort by Date" autocheck="true"/>
            </menupopup>
        </menu>
    </menupopup>
</zk>
```

Then, when the user moves the mouse pointer over the image for a while, the menupopup will be shown up as shown below.

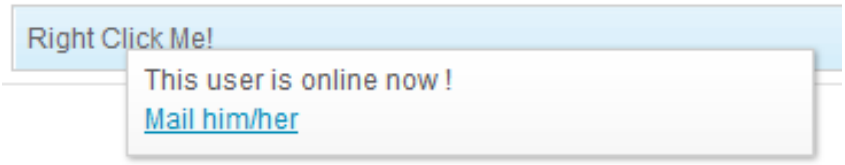


The time to wait before showing up the tooltip can be configured. Please refer to ZK Configuration Reference for more information.

Context Menus

Providing a customized context menu is the same, except it uses the `context` property instead. For example,

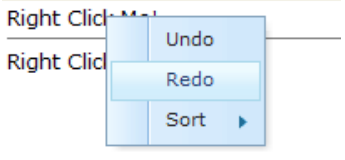
```
<zk>
    <listbox>
        <listitem label="Right Click Me!" context="status"/>
    </listbox>

    <popup id="status" width="300px">
        <vlayout>
            This user is online now !
            <a label="Mail him/her"/>
        </vlayout>
    </popup>
</zk>
```



As shown above, you could use Popup (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Popup.html#) so the context menu is not limited to a menupopup.

Here is another example: context menus versus right clicks.

```
<zk>
      <menupopup id="editPopup">
          <menuitem label="Undo"/>
          <menuitem label="Redo"/>

          <menu label="Sort">
              <menupopup>
                  <menuitem label="Sort by Name" autocheck="true"/>
                  <menuitem label="Sort by Date" autocheck="true"/>
              </menupopup>
          </menu>
      </menupopup>

      <label value="Right Click Me!" context="editPopup"/>
      <separator bar="true"/>
      <label value="Right Click Me!" onRightClick="alert(self.value)"/>
</zk>
```



Notice that the `menupopup` is not visible until a user right-clicks on a component that is associated with its' ID.

Tip: If you want to disable browser's default context menu, you can set the `context` attribute of a component to a non-existent ID.

The `popup` component is generic popup and you are able to place any kind of components inside of popup. For example,

```
<zk>
      <label value="Right Click Me!" context="any"/>
</zk>
```

```
<zk>
      <label value="Right Click Me!" context="any"/>

      <popup id="any" width="300px">
          <vbox>
               It can be anything.
              <toolbarbutton label="ZK" href="http://zk1.sourceforge.net"/>
          </vbox>
      </popup>
</zk>
```

Popups

Providing a customized popup is the same, except it uses the `popup` property instead. For example,

```
<zk>
    <label value="Click Me!" popup="status"/>

    <popup id="status" width="300px">
        <vlayout>
            This user is online now !
            <a label="Mail him/her"/>
        </vlayout>
    </popup>
</zk>
```

Limitation of iOS

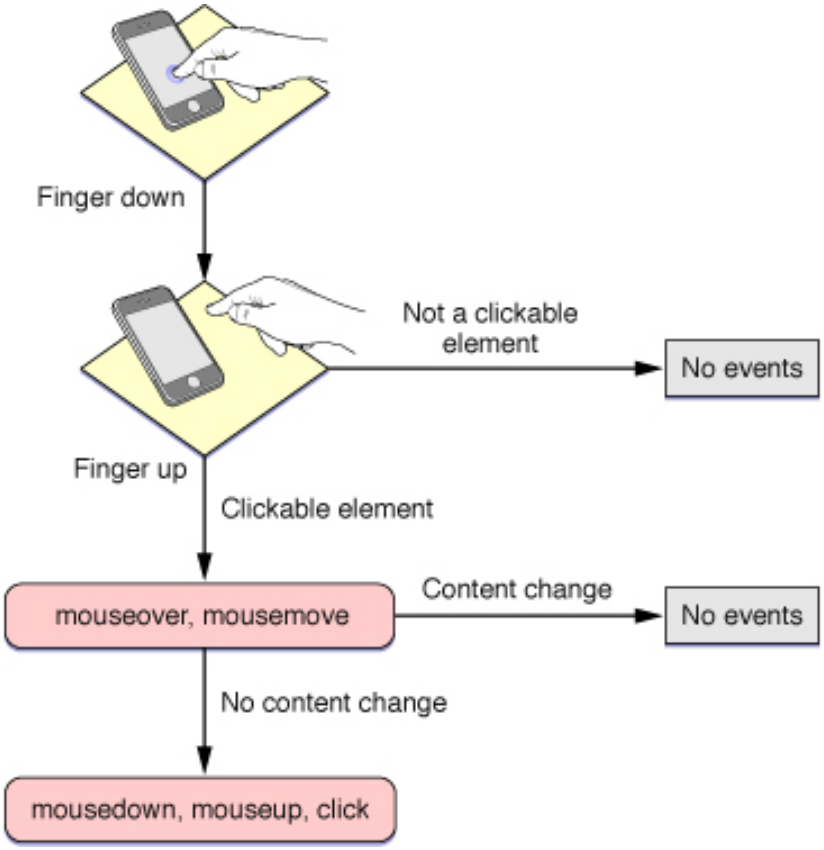
Tooltips

Since there is no mouse move event in iOS, the tooltip won't be shown.

Context Menu

[since 5.0.7]

ZK Client Engine will simulate the context menu, if the user touches the DOM element, associated with the contextmenu property, and holds a while.



Popup

Like onClick, ZK Client Engine simulates the click, if the user touches the DOM element associated with the popup property.

For more information, please refer to Safari Developer Library (http:/ / developer. apple. com/ library/ safari/ #documentation/ AppleApplications/ Reference/ SafariWebContent/ HandlingEvents/ HandlingEvents. html#/ / apple_ref/doc/uid/TP40006511-SW1).

Version History

Version	Date	Content
5.0.7	May 2011	Context Menus supported with iOS

Keystroke Handling

Keystroke handling is generic. Any component inherited from XulElement [1] can handle the key event in the same way.

ENTER and ESC

To handle ENTER, you could listen to the onOK event (notice O and K are both in upper case). To handle ESC, you could listen to the onCancel event. For example,

```
<grid id="form" apply="foo.Login">
    <rows>
        <row>Username: <textbox id="username"/></row>
        <row>Password: <textbox id="password" type="password"/></row>
        <row><button label="Login" forward="form.onOK"/><button label="Reset" forward="fo
    </rows>
</grid>
```

Then, you could implement a composer as follows.

```
package foo;
import org.zkoss.zul.*;
public class Login extends org.zkoss.zk.ui.util.GenericForwardComposer {
    Textbox username;
    Textbox password;
    public void onOK() {
        //handle login
    }
    public void onCancel() {
        username.setValue("");
        password.setValue("");
    }
}
```

Notice that the `onOK` and `onCancel` events are sent to the nearest ancestor of the component that has the focus. In other words, if you press ENTER in a textbox, then ZK will look up the textbox, its parent, its parent's parent and so on to see if any of them has been registered a listener for `onOK`. If found, the event is sent to it. If not found, nothing will happen.

Also notice that, if a button gains the focus, ENTER will be intercepted by the browser and interpreted as pressed. For example, if you move the focus to the Reset button and press ENTER, you will receive `onCancel` rather than `onOK` (since `onClick` will be fired and it is converted to `onCancel` because of the forward attribute specified).

Control Keys

To handle the control keys, you have to specify the keystrokes you want to handle with XulElement.setCtrlKeys(java.lang.String) [1]. Then, if any child component gains the focus and the user presses a keystroke matches the combination, the `onCtrlKey` will be sent to the component with an instance of KeyEvent [2].

Like ENTER and ESC, you could specify the listener and the `ctrlKeys` property in one of the ancestors. ZK will search the component having the focus, its parent, its parent's parent and so on to find if any of them specifies the `ctrlKeys` property that matches the keystroke.

For example,

```
<vlayout ctrlKeys="@c^a#f10^#f3" onCtrlKey="doSomething(event.getKeyCode())">
    <textbox/>
    <datebox/>
</vlayout>
```

As shown, you could use KeyEvent.getKeyCode() [3] to know which key was pressed.

Allowed Control Keys

<table>
<tr><th>Key</th><th>Description</th></tr>
<tr><td>^k</td><td>The control key, i.e., <code>Ctrl+k</code>, where <code>k</code> can be a~z, 0~9, #n and ~n.</td></tr>
<tr><td>@k</td><td>The alt key, i.e., <code>Alt+k</code>, where <code>k</code> can be a~z, 0~9, #n and ~n.</td></tr>
<tr><td>$k</td><td>The shift key, i.e., <code>Shift+k</code>, can <code>k</code> could be #n and ~n.</td></tr>
<tr><td>#n</td><td>A special key as follows.
<table>
<tr><td>#home</td><td>Home</td><td>#end</td><td>End</td><td>#ins</td><td>Insert</td></tr>
<tr><td>#del</td><td>Delete</td><td>#left</td><td>←</td><td>#right</td><td>→</td></tr>
<tr><td>#up</td><td>↑</td><td>#down</td><td>↓</td><td>#pgup</td><td>PgUp</td></tr>
<tr><td>#pgdn</td><td>PgDn</td><td>#bak</td><td>Backspace</td><td></td><td></td></tr>
<tr><td>#fn</td><td colspan="5">A function key. #f1, #f2, ... #f12 for F1, F2,... F12.</td></tr>
</table>
</td></tr>
</table>

Document-level Keystrokes

`[5.0.6]`

If there is no widget gaining a focus when the end user presses a keystroke, ZK will try to locate the first root component and then forward the event to it. For example, when visiting the following page, the <mp>div</mp> component will receive the <mp>onOK</mp> event.

```
<div onOK="doSomething()">
...
</div>
```

In other words, doSomething() will be called if the user presses ENTER, even though no widget ever gains the focus.

Nested Components

Keystrokes are propagated up from the widget gaining the focus to the first ancestor widget that handles the keystroke. For example,

```
<div onOK="doFirst()">
   <textbox id="t1"/>
   <div onOK="doSecond()">
       <textbox id="t2"/>
   </div>
</div>
```

Then, doSecond() is called if `t2` is the current focus, and doFirst() is called if `t1` has the focus.

Version History

Version	Date	Content
5.0.6	January 2011	Document-level keystroke handling was introduced.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/impl/XulElement.html#setCtrlKeys(java.lang.String)
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/KeyEvent.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/KeyEvent.html#getKeyCode()

Drag and Drop

ZK allows a user to drag particular components around the user interface. For example, dragging an image representing a file onto a tree represents a directory, or dragging a listitem representing a product onto a listbox represents a shopping cart.

A component is droppable, if a user could drop a draggable component to it.

Notice that ZK does not assume any behavior about what should take place after dropping. It is up to application developers to implement the `onDrop` event listener.

If an application doesn't do anything, the dragged component is simply moved back where it is originated from.

The draggable and droppable Properties

With ZK, you could make a component draggable by assigning any value, other than `"false"`, to the draggable property by the use of HtmlBasedComponent.setDraggable(java.lang.String) [1]. To disable it, assign it with `"false"`.

```
<image draggable="true"/>
```

Similarly, you could make a component droppable by assigning `"true"` to the droppable property by the use of HtmlBasedComponent.setDroppable(java.lang.String) [2].

```
<hbox droppable="true"/>
```

Then, the user could drag a draggable component, and then drop it to a droppable component.

Since the draggable and droppable properties are implemented in HtmlBasedComponent [3], almost all the components can become draggable or droppable.

The onDrop Event

Once a user has dragged a component and dropped it to another component, the component that the user dropped the component to will be notified by the `onDrop` event. The `onDrop` event is an instance of DropEvent [4]. By calling DropEvent.getDragged() [5], you could retrieve what has been dragged (and dropped).

Notice that the target of the `onDrop` event is the droppable component, not the component being dragged.

The following is a simple example that allows users to reorder list items by drag-and-drop.

```
<window title="Reorder by Drag-and-Drop" border="normal">
    Unique Visitors of ZK:
    <listbox id="src" multiple="true" width="300px">
        <listhead>
            <listheader label="Country/Area"/>
            <listheader align="right" label="Visits"/>
            <listheader align="right" label="%"/>
        </listhead>
        <listitem draggable="true" droppable="true" onDrop="move(event.dragged)">
            <listcell label="United States"/>
            <listcell label="5,093"/>
            <listcell label="19.39%"/>
        </listitem>
        <listitem draggable="true" droppable="true" onDrop="move(event.dragged)">
            <listcell label="China"/>
            <listcell label="4,274"/>
            <listcell label="16.27%"/>
        </listitem>
        <listitem draggable="true" droppable="true" onDrop="move(event.dragged)">
            <listcell label="France"/>
            <listcell label="1,892"/>
            <listcell label="7.20%"/>
        </listitem>
        <listitem draggable="true" droppable="true" onDrop="move(event.dragged)">
            <listcell label="Germany"/>
            <listcell label="1,846"/>
            <listcell label="7.03%"/>
        </listitem>
        <listitem draggable="true" droppable="true" onDrop="move(event.dragged)">
            <listcell label="(other)"/>
            <listcell label="13,162"/>
            <listcell label="50.01%"/>
        </listitem>
        <listfoot>
            <listfooter label="Total 132"/>
            <listfooter label="26,267"/>
            <listfooter label="100.00%"/>
```

```
        </listfoot>
    </listbox>
    <zscript>
            void move(Component dragged) {
                  self.parent.insertBefore(dragged, self);
            }
    </zscript>
</window>
```

Dragging with Multiple Selections

When a user drag-and-drops a list item or a tree item, the selection status of these items won't be changed. Usually only the dragged item is moved, but you can handle all the selected items at once by looking up the set of all selected items as depicted below.

```
public void onDrop(DropEvent evt) {
      Set selected =
((Listitem)evt.getDragged()).getListbox().getSelectedItems();
      //then, you can handle the whole set at once
}
```

Notice that the dragged item may not be selected. Thus, you may prefer to change the selection to the dragged item for this case, as shown below.

```
Listitem li = (Listitem)evt.getDragged();
      if (li.isSelected()) {
            Set selected =
((Listitem)evt.getDragged()).getListbox().getSelectedItems();
            //then, you can handle the whole set at once
      } else {
            li.setSelected(true);
            //handle li only
}
```

Multiple Types of Draggable Components

It is common that a droppable component doesn't accept all draggable components. For example, an e-mail folder accepts only e-mails and it rejects contacts or others. You could silently ignore non-acceptable components or alert an message, when `onDrop` is invoked.

To have better visual effect, you could identify each type of draggable components with an identifier, and then assign the identifier to the `draggable` property.

```
<listitem draggable="email"/>
 ...
<listitem draggable="contact"/>
```

Then, you could specify a list of identifiers to the `droppable` property to limit what can be dropped. For example, the following image accepts only `email` and `contact`.

```
<image src="/img/send.png" droppable="email, contact" onDrop="send(event.dragged)"/>
```

To accept any kind of draggable components, you could specify `"true"` to the `droppable` property. For example, the following image accepts any kind of draggable components.

```
<image src="/img/trash.png" droppable="true" onDrop="remove(event.dragged)"/>
```

On the other hand, if the `draggable` property is `"true"`, it means the component belongs to anonymous type. Furthermore, only components with the `droppable` property assigned to `"true"` could accept it.

Drag-and-Drop Effect Customization

The effects of drag-and-drop can be customized. It requires some client-side programming. Please refer to ZK Client-side Reference/Customization/Drag-and-Drop Effects for more information.

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlBasedComponent.html#setDraggable(java.lang.String)
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlBasedComponent.html#setDroppable(java.lang.String)
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlBasedComponent.html#
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/DropEvent.html#
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/DropEvent.html#getDragged()

Page Initialization

Sometimes it is helpful to run some code before ZK Loader instantiates any component. For example, check if the user has the authority to access, initialize some data, or prepare some variables for EL expressions.

This can be done easily by implementing Initiator [1], and then specifying it with the init directive.

```
<?init class="com.foo.MyInitial"?>
```

Exception Handling

The initiator can be used to handle the exception when ZK Loader renders a page by implementing Initiator.doCatch(java.lang.Throwable) [2]

> **Notice** that it does not cover the exception thrown in an event listener, which could be handled by the use of ExecutionCleanup [3].

```
import org.zkoss.zk.ui.Page;
import org.zkoss.zk.ui.util.Initiator;

public class ErrorHandler implements Initiator {
    public void doInit(Page page, Map args) throws Exception {
    }
    public void doAfterCompose(Page page) throws Exception { //nothing
to do
```

```
    }
    public boolean doCatch(Throwable ex) throws Exception {
        //handle exception here
        return shallIgnore(ex); //return true if the exception is safe
to ignore
    }
    public void doFinally() throws Exception {
        //the finally cleanup
    }
}
```

Initiator and EL

To prepare a variable for EL expression in an initiator, you could store the variable in the page's attributes.

> **Notice** that the provision of variables for EL expression is generally better to be done with VariableResolver [2] (and then specified it with the variable-resolver directive).

For example, suppose we have a class, CustomerManager, that can be used to load all customers, then we could prepare a variable to store all customers as follows.

```
import org.zkoss.zk.ui.Page;
import org.zkoss.zk.ui.util.Initiator;

public class AllCustomerFinder implements Initiator {
    public void doInit(Page page, Map args) throws Exception {
        String name = (String)args.get("name");
        page.setAttribute(name != null ? name: "customers",
CustomerManager.findAll());
    }
    public void doAfterCompose(Page page) throws Exception { //nothing
to do
    }
    public boolean doCatch(Throwable ex) throws Exception { //nothing
to do
        return false;
    }
    public void doFinally() throws Exception { //nothing to do
    }
}
```

Then, we could use the initiator in a ZUML document as follows.

```
<?init class="my.AllCustomerFinder" name="customers"?>

<listbox id="personList" width="800px" rows="5">
    <listhead>
        <listheader label="Name"/>
        <listheader label="Surname"/>
        <listheader label="Due Amount"/>
```

```
    </listhead>
    <listitem value="${each.id}" forEach="${pageScope.customers}">
        <listcell label="${each.name}"/>
        <listcell label="${each.surname}"/>
        <listcell label="${each.due}"/>
    </listitem>
</listbox>
```

System-level Initiator

```
[since 5.0.7]
```

If you have an initiator that shall be invoked for each page, you could register a system-level initiator, rather than specifying it on every page.

It could be done by specifying the initiator you implemented in WEB-INF/zk.xml as follows. For more information, please refer to ZK Configuration Reference.

```
<listener>
    <listener-class>foo.MyInitiator</listener-class>
</listener>
```

Once specified, an instance of the given class will be instantiated for each page (Page [8]), and then its method will be called as if they are specified in the page (the init directive).

Version History

Version	Date	Content
5.0.7	April 2011	The system-level initiator was introduced.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Initiator.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Initiator.html#doCatch(java.lang.Throwable)
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ExecutionCleanup.html#

Forward and Redirect

A Web application jumps from one URL to another is usually caused by the user's click on a hyperlink, such as clicking on a button, toolbarbutton, menuitem and a that is associated with the `href` attribute.

```
<button label="Next" href="next.zul"/>
```

It is done at the client without Java code, so it is efficient. However, you could control it on the server (in Java) too, such that you could redirect it based on some information that is available only at the server.

Redirect to Another URL

Redirecting to another URL is straightforward: pass the URL to Executions.sendRedirect(java.lang.String) [1]. A typical use case is to redirect after authenticating the user's login.

```
if (someCondition())
   Executions.sendRedirect("/ready.zul");
```

You could also ask the browser to open another browser window from a given URL by the use of java.lang.String) Execution.sendRedirect(java.lang.String, java.lang.String) [2].

Redirect When Loading

Executions.sendRedirect(java.lang.String) [1] is designed to be used when serving an AU request (aka., Ajax). If you want to redirect to another page when loading a ZUML document, it is more efficient to call HttpServletResponse.sendRedirect [3][4] , such that the browser will handle the redirect for you without running any JavaScript code.

For example,

```
if (!isLogin()) {
    Execution exec = Executions.getCurrent();
    HttpServletResponse response =
(HttpServletResponse)exec.getNativeResponse();
    response.sendRedirect(response.encodeRedirectURL("/login"));
//assume there is /login
    exec.setVoided(true); //no need to create UI since redirect will
take place
}
```

Notice that we invoke Executionj.setVoided(boolean) [5] to *void* an execution, such that ZK Loader will abort the evaluation of a ZUML document (if you prefer not to generate any UI when redirecting).

Also notice that the casting to `javax.servlet.http.HttpServletResponse` in the above example does *not* work in a portlet, since the native response is an instance of `javax.portlet.RenderResponse`.

To check whether to redirect can be packed as Initiator [1], see below for an example:

```
public class AuthenticateInit extends
org.zkoss.zk.ui.util.GenericInitiator {
    public void doInit(Page page, Map args) throws Exception {
        if (!isLogin()) {
            Execution exec = Executions.getCurrent();
```

```
            HttpServletResponse response =
(HttpServletResponse)exec.getNativeResponse();

response.sendRedirect(response.encodeRedirectURL("/login")); //assume
there is /login
            exec.setVoided(true); //no need to create UI since redirect
 will take place
        }
    }
}
```

Then, you could specify it in your ZUML document:

```
<?init class="foo.AuthenticateInit"?>
```

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#sendRedirect(java.lang.String)
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Execution.html#sendRedirect(java.lang.String,
[3] http://download.oracle.com/javaee/1.4/api/javax/servlet/http/HttpServletResponse.html#sendRedirect%28java.lang.String%29
[4] It actually sets the refresh header (http://www.metatags.org/meta_http_equiv_refresh).
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executionj.html#setVoided(boolean)

Forward to Another Page

Sometimes we have to forward to another page. For example, when a user visits a page that requires authorization, we could forward it to a login page[1] .

The simplest way is to use the forward directive:

```
<?forward uri="/login.zul" if="${!foo:isLogin()}"?>
```

where we assume `isLogin` is an EL function that returns whether the user has logged in. For more information, please refer to the Conditional Evaluation section.

You could forward to another page by the use of Executions.forward(java.lang.String) (http:// www. zkoss. org/ javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#forward(java.lang.String)) too.

Notice that forwarding can be called only when loading a page. You cannot call it when handling an Ajax request (such as when a user clicks a button). For handling an Ajax request, you have to use *redirect* as described in the previous section.

Unlike redirect, forward does *not* change the URL that the browser knows. Rather, it is purely server-side activity: using another page's content instead of the original one to render the output of the given (and the same) URL.

[1] In additions to forwarding, we could popup a window to ask him to login, i.e., without leaving the current desktop

Version History

Version	Date	Content

File Upload and Download

File Upload

You could associate the upload feature to a component, such as Button [2], Toolbarbutton [1] or Menuitem [2]. In other words, you could make a button or a menu item to upload a file when the user clicks on it.

The implementation is straightaway:

1. Assign the `upload` attribute to true for the component used to upload a file
2. Assign an event listener to the component for the `onUpload` event[3]

For example,

```
<zk>
      <zscript>
      void upload(UploadEvent event) {
            org.zkoss.util.media.Media media = event.getMedia();
            if (media instanceof org.zkoss.image.Image) {
                  org.zkoss.zul.Image image = new
org.zkoss.zul.Image();
                  image.setContent( (org.zkoss.image.Image) media);
                  image.setParent(pics);
            } else {
                  Messagebox.show("Not an image: "+media, "Error",
Messagebox.OK, Messagebox.ERROR);
            }
      }
      </zscript>
      <button label="Upload" upload="true" onUpload="upload(event)"/>
      <vlayout id="pics" />
</zk>
```

You could control the maximal allowed number of files, the maximal allowed size and other information by use of Button.setUpload(java.lang.String) [4].

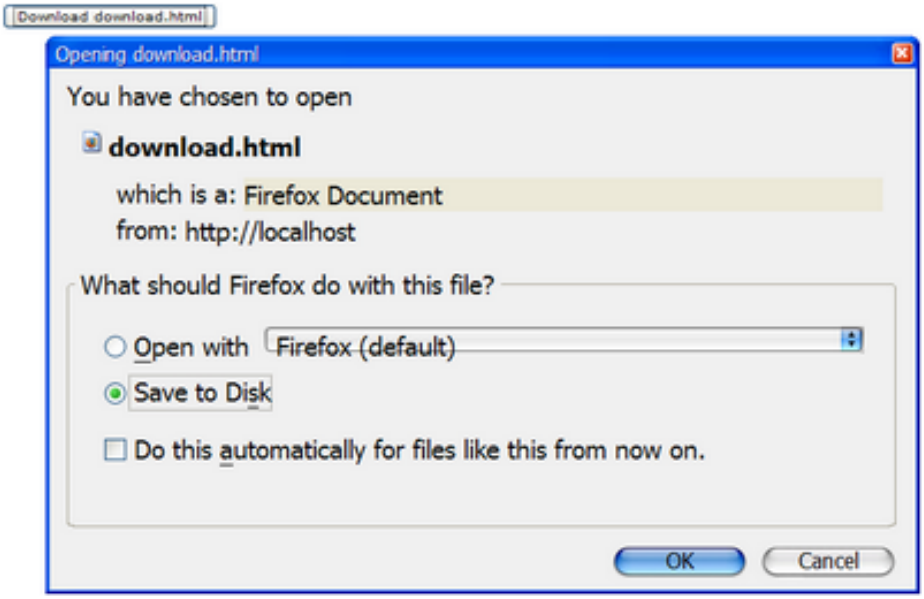
```
<menupopup>
    <menuitem label="Upload" upload="true,maxsize=-1,native"/>
</menupopup>
```

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Toolbarbutton.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Menuitem.html#
[3] If you enabled the use of event threads, you could use Fileupload.get() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Fileupload.html#get()) and related. For more information, please refer to the Event Thread section.
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Button.html#setUpload(java.lang.String)

File Download

Filedownload provides a set of utilities to prompt a user for downloading a file from the server to a local folder. For example,

```
<button label="Download calendar.pdf" onClick='Filedownload.save("/resources/calendar.pdf
```



The file could be a static resource, an input stream, a file, a URL and others. Please refer to ZK Component Reference and Filedownload (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Filedownload.html#) for more information.

Version History

Version	Date	Content

Browser Information and Control

To retrieve the information about the client, you can register an event listener for the `onClientInfo` event to a root component. To control the behavior of the client, you can use the utilities in the Clients [1] class.

Browser Information

Sometimes an application needs to know the client's information, such as time zone. Then, you can add an event listener for the `onClientInfo` event. Once the event is added, the client will send back an instance of the ClientInfoEvent [2] class, from which you can retrieve the information of the client.

For example,

```
<grid onClientInfo="onClientInfo(event)">
    <rows>
        <row>Time Zone <label id="tm"/></row>
        <row>Screen <label id="scrn"/></row>
    </rows>

    <zscript>
     void onClientInfo(ClientInfoEvent evt) {
         tm.setValue(evt.getTimeZone().toString());
         scrn.setValue(
```

```
evt.getScreenWidth()+"x"+evt.getScreenHeight()+"x"+evt.getColorDepth());
     }
    </zscript>
</grid>
```

Notice that the `onClientInfo` event is meaningful only to the root component (aka., a component without any parent).

The client information is not stored by ZK, so you have to store it manually if necessary. Since a session is associated with the same client, you can store the client info in the session's attribute.

For example, we coud use it to control the default time zone[3] .

```
 session.setAttribute("org.zkoss.web.preferred.timeZone",
event.getTimeZone());
```

Notice that the `onClientInfo` event is sent from the client after the UI is rendered at the client. Thus, if some of your component's data depends on the client's info, say, screen resolution, it is better to handle it (say, adjust UI's size) when the `onClientInfo` event is received.

If it is, though rarely, too late (i.e., it has to be done at beginning), you could ask the client to re-send the request again with Executions.sendRedirect(java.lang.String) [1]. For example,

```
 import org.zkoss.util.TimeZones;
 ...
  if (!TimeZones.getCurrent().equals(event.getTimeZone()) {
     session.setAttribute("org.zkoss.web.preferred.timeZone",
event.getTimeZone()); //update to the session
     Executions.sendRedirect(null); //ask to re-send (i.e., redo)
  }
```

---

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

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

[3] For more information about time zone, please refer to the Time Zone section.

Browser Control

Clients (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/ui/util/Clients.html#) has utilities to control the client's visual presentation (more precisely, the browser window), such as printing, submitting, resizing and so on. For example, you can scroll the browser window (aka., the desktop) as follows.

```
 Clients.scrollBy(100, 0);
```

Here we describe some special controls that are worth to notice. For complete functionality, please refer to Clients (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/ui/util/Clients.html#).

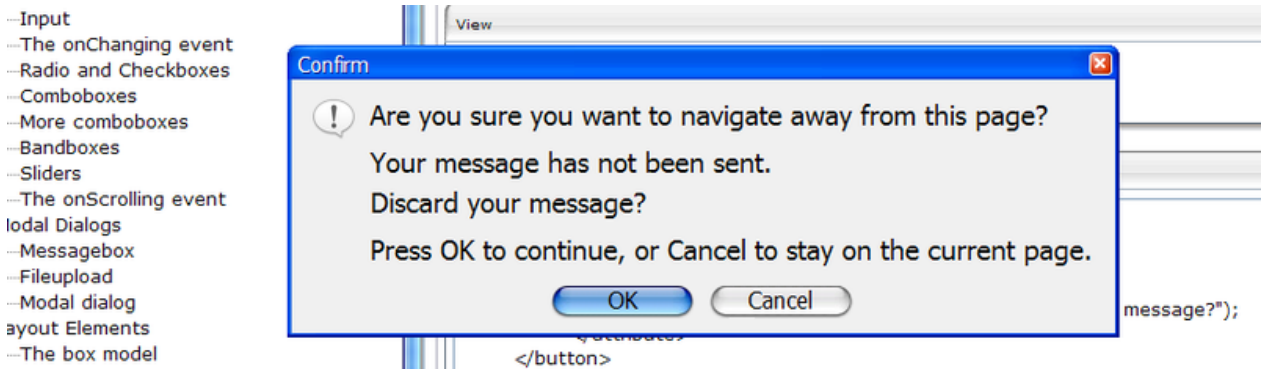
Prevent User from Closing a Browser Window

In some situation, you might want to confirm a user when he tries to close the window or browse to another URL. It can be done by the use of Clients.confirmClose(java.lang.String) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Clients.html#confirmClose(java.lang.String)).

For example, when a user is composing a mail that is not sent or saved yet.

```
 if (mail.isDirty()) {
     Clients.confirmClose("Your message has not been sent.\nDiscard
your message?");
 } else {
     Clients.confirmClose(null); //reset. no more confirmation.
 }
```

As shown, it is done by passing the warning message to the client with Clients.confirmClose(java.lang.String) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Clients.html#confirmClose(java.lang.String)). Then, a confirmation dialog is shown up when the user tries to close the browser window, reload, or browse to another URL:



To disable the confirmation, just invoke Clients.confirmClose(java.lang.String) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Clients.html#confirmClose(java.lang.String)) with null.

Notify User Component under Processing

Sometimes a request may take long time to process, and it is better to notice to the user that it is under processing. It can be done by the use of java.lang.String) Clients.showBusy(org.zkoss.zk.ui.Component, java.lang.String) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Clients.html#showBusy(org.zkoss.zk.ui.Component,), if only a component is not accessible, or Clients.showBusy(java.lang.String) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Clients.html#showBusy(java.lang.String)) if the whole browser is not accessible. For example,

```
showBusy(grid, "Loading data...");
```

After the loading is completed, you could invoke Clients.clearBusy(org.zkoss.zk.ui.Component) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Clients.html#clearBusy(org.zkoss.zk.ui.Component)) (or Clients.clearBusy() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Clients.html#clearBusy()))to clean it up. For more information, please refer to the Use Echo Events section.

Log the Message to Browser

```
[since 5.0.8]
```

In addition to logging messages to the console, you could log the messages to the browser for debugging by the use of Clients.log(java.lang.String) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zk/ ui/ util/ Clients. html#log(java.lang.String)). For example,

```
//in Java
void doSomething() {
   Clients.log("doSomething called");
}
```

Control in JavaScript

If you are familiar with JavaScript, you could have more control by sending any JavaScript code to the client for evaluation. It can be done by preparing the JavaScript code in AuInvoke (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/au/out/AuInvoke.html#) or AuScript (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/au/out/AuScript.html#), and then send back to client by calling Clients.response(org.zkoss.zk.au.AuResponse) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zk/ ui/ util/ Clients. html#response(org. zkoss. zk. au. AuResponse)).

For example, we could use AuScript (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/au/out/AuScript.html#) to inject any code, while AuInvoke (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zk/ au/ out/ AuInvoke.html#) is better if you want to invoke a function of a client-side widget.

```
Clients.response(new AuScript(null, "alert('Hello, Client')"));
```

For client-side API, please refer to JavaScript API (http://www.zkoss.org/javadoc/latest/jsdoc/).

Version History

Version	Date	Content
5.0.8	June, 2010	Clients.log(java.lang.String) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Clients.html#log(java.lang.String)) was introduced.

Browser History Management

History Management with Bookmark

In traditional multi-page Web applications, users usually use the BACK and FORWARD button to surf around multiple pages, and bookmark them for later use. With ZK, you still can use multiple pages to represent different set of features and information, as you did in traditional Web applications.

However, it is more common for ZK applications to represent a lot of features in one desktop, which usually requires multiple Web pages in a traditional Web application. To make user's surfing easier, ZK supports the browser's history management that enables ZK applications to manage browser's history simply on the server.

The concept is simple. For each state of a desktop, you could add a so-called bookmark[1] to the browser's history. Then, the user can use the BACK and FORWARD button of the browser to switch around different bookmarks. The change of books will be sent back to the server called `onBookmarkChange`, and you application can switch to the corresponding accordingly.

From application's viewpoint, it takes two steps to manage the browser's history:

1. Add a bookmark to the browser's history for each of the visited states of your desktop.
2. Listen to the `onBookmarkChange` event for bookmark change, and switch the state accordingly.

---

[1] Each bookmark is an arbitrary string added to the browser's history.

Add Bookmarks to Browser's History

Your application has to decide what the appropriate states are to add to the browser's history. For example, in a multi-step operation, each step is a good candidate of states for adding to a browser's history, such that the user can switch around these steps by pressing BACK or FORWARD buttons.

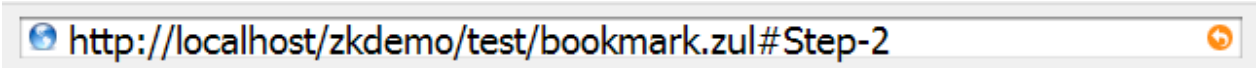
Once you decide when to add a state to the browser's history, you can simply invoke Desktop.setBookmark(java.lang.String) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Desktop.html#setBookmark(java.lang.String)). Notice that it is *not* the bookmarks that users add to the browser (aka., My Favorites in Internet Explorer).

For example, assume you want to bookmark the state when the Next button is clicked, then you do as follows.

```
<button label="Next" onClick='desktop.setBookmark("Step-2")'/>
```

If you look carefully at the URL, you will find ZK appends `#Step-2` to the URL.

If you press the BACK button, you will see as follows.



Listen to onBookmarkChange and Change the State Accordingly

After adding a bookmark to the browser's history, users can then surf among these bookmarks such as pressing the BACK button to return to the previous bookmark. When the bookmark is changed, ZK will notify the application by broadcasting the `onBookmarkChange` event (an instance of the BookmarkEvent (http:// www. zkoss. org/javadoc/latest/zk/org/zkoss/zk/ui/event/BookmarkEvent.html#) class) to all root components in the desktop.

Unlike traditional multi-page Web applications, you have to change the desktop's state on the server manually, when `onBookmarkChange` is received. ZK does nothing to allow an application to set a bookmark and notify for the bookmark change. It is the application developer's job to manipulate the desktop to reflect the state that a bookmark has represented.

To listen the `onBookmarkChange` event, you can add an event listener to any pages of the desktop, or to any of its root components.

```
<window onBookmarkChange="goto(event.bookmark)">
    <zscript>
     void goto(String bookmark) {
         if ("Step-2".equals(bookmark)) {
             ...//create components for Step 2
         } else { //empty bookmark
             ...//create components for Step 1
         }
    </zscript>
</window>
```

Like handling any other events, you can manipulate the UI any way you want, when the `onBookmarkChange` event is received. It is totally up to you.

A typical approach is to use one of the `createComponents` methods of the Executions (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#) class. In other words, you could represent each state with one ZUML page, and then use `createComponents` to create all components in it when `onBookmarkChange` is received.

```
if ("Step-2".equals(bookmark)) {
    //1. Remove components, if any, representing the previous state
    try {
        self.getFellow("replacable").detach();
    } catch (ComponentNotFoundException ex) {
        //not created yet
    }

    //2. Creates components belonging to Step 2
    Executions.createComponents("/bk/step2.zul", self, null);
}
```

Example

In this example, we bookmarks each tab selection.

```
<window id="wnd" title="Bookmark Demo" width="400px" border="normal">
    <zscript>
     page.addEventListener(Events.ON_BOOKMARK_CHANGE,
         new EventListener() {
             public void onEvent(Event event) throws UiException {
                 try {

wnd.getFellow(wnd.desktop.bookmark).setSelected(true);
                 } catch (ComponentNotFoundException ex) {
                     tab1.setSelected(true);
                 }
             }
         });
    </zscript>

    <tabbox id="tbox" width="100%" onSelect="desktop.bookmark = self.selectedTab.id">
        <tabs>
            <tab id="tab1" label="Tab 1"/>
            <tab id="tab2" label="Tab 2"/>
            <tab id="tab3" label="Tab 3"/>
        </tabs>
        <tabpanels>
            <tabpanel>This is panel 1</tabpanel>
            <tabpanel>This is panel 2</tabpanel>
            <tabpanel>This is panel 3</tabpanel>
        </tabpanels>
    </tabbox>
</window>
```

Bookmarking with iframe

If a page contains one or more `iframe` components, it is sometimes better to bookmark the status of the `iframe` components too. For example, when the contained `iframe` was navigated to another URL, you might want to change the bookmark of the page (the container), such that you can restore to the `iframe` to the right content. To do this, you have to listen to the `onURIChange` event as follows.

```
<window onURIChange="desktop.bookmark = storeURI(event.getTarget(), event.getURI())">
    <iframe src="${uri_depends_on_bookmark}" forward="onURIChange"/>
</window>
```

The `onURIChange` event is sent as an instance of URIEvent (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/URIEvent.html#).

Notice that the `onURIChange` event is sent only if the iframe contains another ZK page. If it contains non-ZK page, you have to handle it manually. Please refer to ZK Component Reference: iframe for more information.

Version History

Version	Date	Content

Session Timeout Management

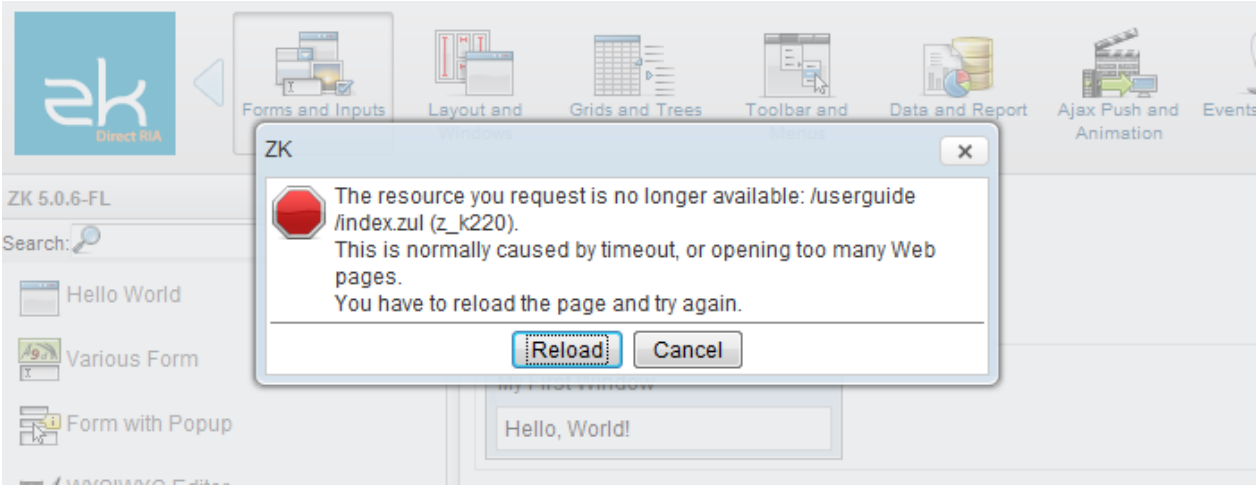
After a session is timeout, all desktops and UI objects it belongs are removed. If a user keeps accessing the desktop that no longer exists, ZK will prompt the user for the session-timeout situation. ZK supports several ways to prompt the user for session timeout:

- Show a message
- Redirect to another page
- Totally Control by running JavaScript code

You could pick one depending on your application requirement. In addition, you could configure your application to enforce the user prompting to take place, without waiting the user's activity. It is called automatic timeout.

Show a Message

By default, a message is shown up to prompt the user and prevent from further accessing as depicted below.



Custom Message

You could show a custom message by specifying `timeout-message` in `WEB-INF/zk.xml`. For example,

```
<session-config>
      <device-type>ajax</device-type>
      <timeout-message>Session timeout. Please reload.</timeout-message>
</session-config>
```

Internationalization

If you want to specify a Locale-dependent message, you could specify the key and prefix it with label: as follows.

```
<session-config>
      <device-type>ajax</device-type>
      <timeout-message>label:timeout</timeout-timeout>
</session-config>
```

Then, you have to prepare the zk-label properties files as described in the Labels section.

```
#zk-label.properties
timeout={
Session timeout.
(multi-line is allowed)
}
```

Redirect to Another Page

Sometimes it is better to redirect to another page that gives users more complete description and guides they to the other resources, or asks them to login again. You can specify the target URI, that you want to redirect users to when timeout, with the timeout-uri element in WEB-INF/zk.xml. For example, the target URI is /timeout.zul and then you can add the following lines to zk.xml.

```
<session-config>
    <device-type>ajax</device-type>
    <timeout-uri>/timeout.zul</timeout-uri>
</session-config>
```

In addition to WEB-INF/zk.xml, you could change the redirected URI manually as follows.

```
Devices.setTimeoutURI("ajax", "/timeout.zul");
```

About Device: A device represents the client device, such as Ajax browsers and Android devices. Each desktop is associated with one device, and vice versa.

If you prefer to reload the page instead of redirecting to other URI, you can specify an empty URI as follows.

```
<session-config>
    <device-type>ajax</device-type>
    <timeout-uri></timeout-uri>
</session-config>
```

Total Control in JavaScript

If you want more amazing effect, you could provide some JavaScript code and configure ZK to run it if timeout. For example, our demo [1] shows up a message on the top of window with some animation, and then automatically reloads if it detects any mouse move (it means the user is back).

For example, you have a function called `foo.timeout` to handle the timeout effect, then you could configure `WEB-INF/zk.xml` as follows.

```
<session-config>
      <device-type>ajax</device-type>
      <timeout-message>script:foo.timeout('Session Timeout')</timeout-timeout>
</session-config>
```

The code depends on the client. For Ajax devices, it has to be JavaScript.

Automatic Timeout

By default, the session-timeout mechanism is triggered only if the client sends back a request (such as clicking on a button). If you prefer to prompt the user even if it doesn't do anything, you could specify the automatic-timeout element in `WEB-INF/zk.xml` as follows.

```
<session-config>
      <device-type>ajax</device-type>
      <automatic-timeout/>
</session-config>
```

Then, ZK Client will trigger the session-time mechanism (showing a message, redirecting to another page, or running some JavaScript code).

Page-level Automatic Timeout

If you want to specify whether to automatically timeout for particular pages, you can use the page directive.

Moreover, it is better to turn off the automate timeout for the timeout page you want to redirect to (if the page is a ZUML page). For example,

```
<!-- my timeout page -->
<?page automaticTimeout="false"?>
...
```

Never Timeout

Though not recommended, you could prevent the session from timeout by making a "keep-alive" timer, such that the desktop keeps alive until the user surfs away.

To do that, you first configure WEB/zk.xml as follows.

```
<session-config>
  <timer-keep-alive>true</timer-keep-alive>
</session-config>
```

and create a timer in your ZUL page:

```
<timer id="timerKeepAliveSession" repeats="true" delay="600000"/>
```

This will prevent the session to time out when the ZUL page is opened in the browser. The session still timeouts when the user has navigated the browser away. The delay (600000 is 10 minutes) shall be as long as possible but smaller than your session timeout.

The timer-keep-alive element is used to specify whether the session shall consider timer as a normal request. If it is considered as a normal request, the session timeout mechanism will be restarted when it is received. Otherwise, the timer, by default, won't restart the timeout mechanism.

Version History

Version	Date	Content
5.0.5	October 2010	The support of Custom Message and JavaScript was introduced.

References

[1] http://www.zkoss.org/zkdemo

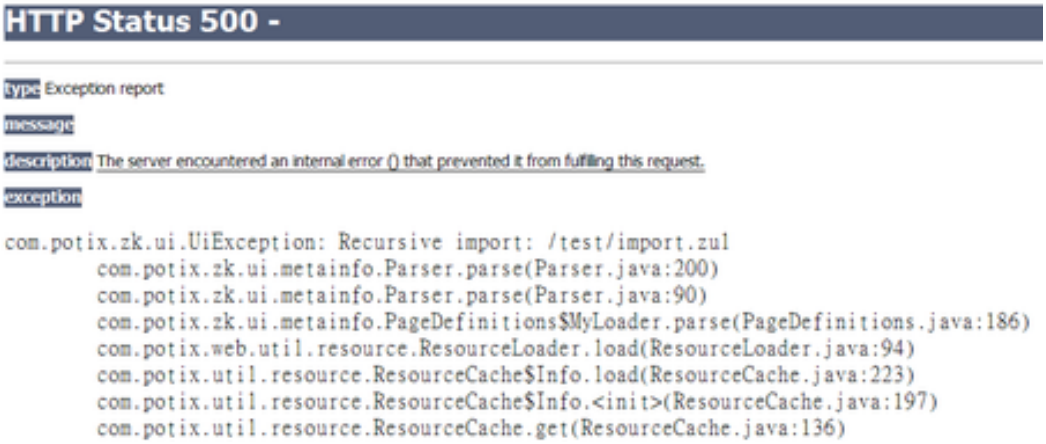
Error Handling

Here we describe how to handle errors. An error is caused by an exception that is not caught by the application. An exception might be thrown in two situations: when loading a ZUML document or when serving an AU request (aka, an Ajax request).

Error Handling When Loading ZUML Documents

If an un-caught exception is thrown when loading a ZUML document, it is handled directly by the Web server. In other words, the handling is no different from other servlets.

By default, the Web server displays an error page showing the error message and stack trace. For example,



You can customize the error handling by specifying the error page in `WEB-INF/web.xml` as follows[1].

```
<!-- WEB-INF/web.xml -->
<error-page>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/WEB-INF/sys/error.zul</location>
</error-page>
```

Then, when an error occurs on loading a page, the Web server forwards the error page you specified, /error/error.zul. Upon forwarding, the Web server passes a set of request attributes to the error page to describe what happens. These attributes are as follows.

Request Attribute	Type
javax.servlet.error.status_code	java.lang.Integer
javax.servlet.error.exception_type	java.lang.Class
javax.servlet.error.message	java.lang.String
javax.servlet.error.exception	java.lang.Throwable
javax.servlet.error.request_uri	java.lang.String
javax.servlet.error.servlet_name	java.lang.String

Then, in the error page, you can display your custom information by the use of these attributes. For example,

```
<window title="Error ${requestScope['javax.servlet.error.status_code']}">
    Cause: ${requestScope['javax.servlet.error.message']}
</window>
```

Tips:

- The error page can be any kind of servlets. In addition to ZUML, you can use JSP or whatever servlet you preferred.

---

[1] Please refer to Java Servlet Specification (http://www.oracle.com/technology/sample_code/tech/java/codesnippet/servlets/HandlingServletExceptions/HandlingServletExceptions.html) for more details.

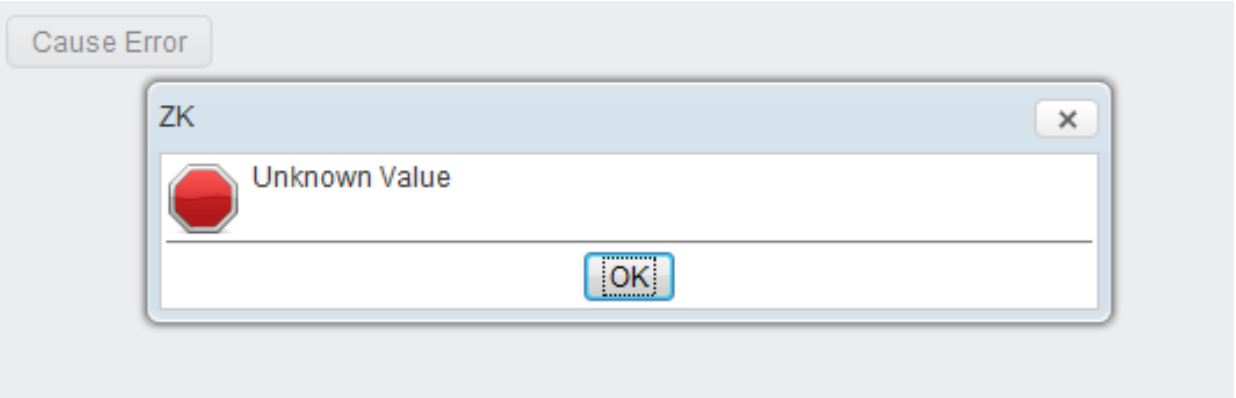
Error Handling When Serving AU Requests

If an uncaught exception is thrown when serving an AU request (aka., an Ajax request; such as caused by an event listener), it is handled by the ZK Update Engine. By default, it simply shows up an error message to indicate the error.

For example, suppose we have the following code:

```
<button label="Cause Error" onClick='throw new NullPointerException("Unknown Value")'/>
```

Then, if you click the button, the following error message will be shown.



You can customize the error handling by specifying the error page in `WEB-INF/zk.xml` as described in ZK Configuration Reference. For example,

```
<!-- zk.xml -->
<error-page>
    <exception-type>java.lang.Throwable</exception-type>
    <location>/WEB-INF/sys/error.zul</location>
</error-page>
```

Then, when an error occurs in an event listener, the ZK Update Engine will create a dialog by the use of the error page you specified, /error/error.zul.

Like error handling in loading a ZUML page, you can specify multiple <error-page> elements. Each of them is associated with a different exception type (the value of <exception-type> element). When an error occurs, ZK will search the error pages one-by-one until the exception type matches.

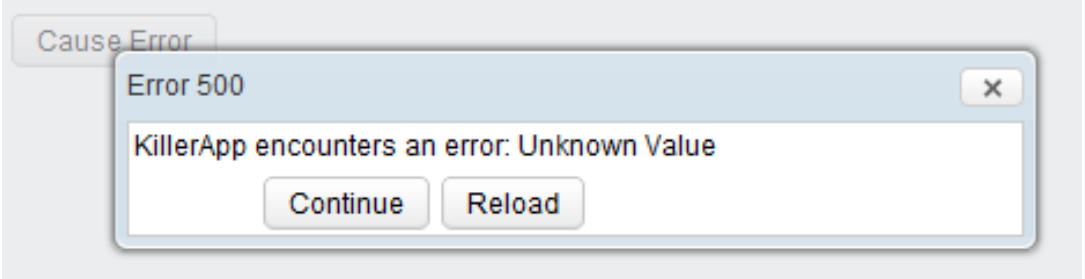
In addition, ZK passes a set of request attributes to the error page to describe what happens. These attribute are as follows.

Request Attribute	Type
javax.servlet.error.exception_type	java.lang.Class
javax.servlet.error.message	java.lang.String
javax.servlet.error.exception	java.lang.Throwable

For example, you can specify the following content as the error page.

```
<window title="Error ${requestScope['javax.servlet.error.status_code']}"
width="400px" border="normal" mode="modal" closable="true">
      <vbox>
KillerApp encounters an error:
${requestScope['javax.servlet.error.message']}
            <hbox style="margin-left:auto; margin-right:auto">
                  <button label="Continue" onClick="spaceOwner.detach()"/>
                  <button label="Reload" onClick="Executions.sendRedirect(null)"/>
            </hbox>
      </vbox>

      <!-- optional: record the error for improving the app -->
      <zscript>
      org.zkoss.util.logging.Log.lookup("Fatal").error(
            requestScope.get("javax.servlet.error.exception"));
      </zscript>
</window>
```

Then, when the button is clicked, the following will be shown.



Tips:

- The error page is created at the same desktop that causes the error, so you can retrieve the relevant information from the desktop.

Version History

Version	Date	Content

Actions and Effects

```
[since 5.0.6]
```

The client-side action (CSA) is used to control how to perform an action at the client. Typical use is to control the effect of showing or hiding a widget. For example, with CSA, you could use the so-called *slide-down* effect to display a widget.

It is a generic feature available to HtmlBasedComponent [3], so you could apply to almost all widgets.

CSA allows the developer to control some actions without JavaScript. If you want to have the full control (and are OK to write some JavaScript code), please refer to ZK Client-side Reference for the complete control of the client-side behavior.

How to Apply Client-side Actions

To apply the client-side action to a widget, you have to assign a value to the action property (HtmlBasedComponent.setAction(java.lang.String) [1]). The syntax is as follows.

```
action="action-name1: effect1; action-name2: effect2"
```

The action name (e.g., action1) has to be one of the predefined names, such as show and hide. The action effect (e.g., effect1) has to be one of the predefined effects, such as slideDown and slideUp.

For example, we could use the *slide-down* effect to display a window as follows[2] .

```
<zk>
      <button label="Show a modal window" onClick="wnd.doModal()"/>
      <window id="wnd" title="Modal" border="normal" width="300px"
       action="show: slideDown" visible="false">
            This is a modal window.
      </window>
</zk>
```

In addition, you could specify additions options by enclosing it with the parentheses as follows.

```
<div action="show: slideDown({duration: 1000}); hide: slideUp({duration: 300})">
....
</div>
```

which specifies the duration of sliding down is 100 milliseconds, and the duration of sliding-up is 300 milliseconds.

Security Note: the optios is actually a JavaScript object (i.e., a map, Map [3]), and ZK passes whatever being specified to the client for evaluation. Thus, if you allow the user to specify the effect, you shall encode it first to avoid cross-site scripting.

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlBasedComponent.html#setAction(java.lang.String)
[2] If you are using the effects with a modal window, it is important to specify the width. Otherwise, the calculation of the position might be wrong.
[3] http://www.zkoss.org/javadoc/latest/jsdoc/_global_/Map.html#

Predefined Actions

Here is a list of predefined actions.

Name	Description
show	The show action is used to display a widget (making a widget visible). When a visible widget is attached to a page, the show action will take place too.
hide	The hide action is used to hide a widget (making a widget invisible). When a visible widget is detached from a page, the hide action will take place too.
invalidate	The invalidate action is invoked when a visible widget is invalidated, i.e., when Component.invalidate() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#invalidate()) is called. Example, action="invalidate: slideDown".

Predefined Effects

Here is a list of predefined actions.

Name	Description
slideDown	Slides down to display this widget (making a wdiget visible). Options: • **duration** - the number of milliseconds to slide down the widget
slideUp	Slides up to hide this widget. (making a widget invisible) Options: • **duration** - the number of milliseconds to slide up the widget
slideIn	Slides in to display this widget (making a wdiget visible). Options: • **duration** - the number of milliseconds to slide in the widget
slideOut	Slides out to hide this widget (making a widget invisible). Options: • **duration** - the number of milliseconds to slide out the widget

Custom Actions

If you want to take some actions other than the predefined actions listed above, you have to override the correspond method at client. For example, suppose you'd like to change the color when a label's value (Label.setValue(java.lang.String) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zul/ Label. html#setValue(java.lang.String))) is changed. Then, you could do as follows:

```
<label id="inf2">
      <attribute w:name="setValue">
      function (value, fromServer) {
            this.$setValue(value, fromServer);
            if (this.desktop) {
                  this._red = !this._red;
                  this.setStyle('background:'+(this._red ?
'red':'green'));
            }
      }
```

```
      </attribute>
</label>
```

For more information, please refer to ZK Client-side Reference: Widget Customization.

Custom Effects

For adding your custom effects, please refer to ZK Client-side Reference: Customization: Actions and Effects for details.

Notes for Upgrading from ZK 3

They are both called Client-side Actions, but they are different and you have to rewrite them to make it work under ZK 5:

1. The action names was changed and the support is limited to show and hide (while ZK 3 supports any onxxx).
2. The action operation must be the name of one of the methods defined in Actions (http://www.zkoss.org/javadoc/latest/jsdoc/zk/eff/Actions.html#) (while ZK 3 is the JavaScript code).
3. It is part of HtmlBasedComponent (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlBasedComponent.html#) (while ZK 3 is XulElement (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/impl/XulElement.html#)).

Version History

Version	Date	Content
5.0.6	December 2010	Client-side actions were introduced since 5.0.6

HTML Tags

Here we discuss how to use HTML tags directly in a ZUML document. There are several ways as described in the following sections, and you could choose one based on your requirement.

What to consider	html component	native namespace	XHTML components	JSP
Update Content Dynamically	Yes	No[1]	Yes	No[2]
Mix with ZUL components	No	Yes	Yes	Yes/No[3]
Memory Footprint	Small	Small	Large	Small

In addition, you could use iframe to embed a complete HTML document which might be from a different website with different technology. Or, use include to include a HTML fragment.

[1] We cannot update content dynamically at the server. However, we could modify the DOM tree directly at client. Please refer to the Client-side UI Composing section.

[2] Technically you could modify the browser's DOM tree dynamically at the client.

[3] You could mix HTML tags with ZK components, if ZK JSP Tags (http://www.zkoss.org/product/zkjsp.dsp) is used. Otherwise, you could only have a JSP page to include other ZUL pages, or vice versa.

The html Component

Overview

One of the simplest ways is to use a XUL component called html to embed whatever HTML tags you want. The html component works like an HTML SPAN tag enclosing the HTML fragment. For example,

```
<window border="normal" title="Html Demo">
    <html><![CDATA[
        <h4>Hi, ${parent.title}</h4>
        <p>It is the content of the html component.</p>
    ]]></html>
</window>
```

As shown above, we enclose them with `<![CDATA[` and `]]>` to prevent ZK Loader from interpreting the HTML tags embedded in the html element. In other words, they are not the child component. Rather, they are stored in the `content` property (by use of Html.setContent(java.lang.String) [1][2] . In other words, `<h4>...</p>` will become the content of the `html` element.

Also notice that EL expressions are allowed.

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Html.html#setContent(java.lang.String)
[2] Fore more information please refer to ZUML Reference.

What Are Generated

The `html` component will eventually generate the HTML SPAN tag to enclose the content when attached to the browser's DOM tree. In other words, it generates the following HTML tags when attached to the browser's DOM tree.

```
<span id="z_4a_3">
    <h4>Hi, Html Demo</h4>
    <p>It is the content of the html component.</p>
</span>
```

It's a Component

The `html` component is no different to other XUL components. For example, you specify the CSS style and change its content dynamically.

```
<zk>
      <html id="h" style="border: 1px solid blue;background: yellow">
      <![CDATA[
           <ul>
               <li>Native browser content</li>
           </ul>
      ]]>
      </html>
      <button label="change" onClick="h.setContent(&quot;Hi, Update&quot;)" />
</zk>
```

You can change its content dynamically.

```
htm.setContent("<ul><li>New content</li></ul>");
```

Limitation

Since `SPAN` is used to enclose the embedded HTML tags, the following code snippet is incorrect.

```
<zk>
      <html><![CDATA[
                <ul>
         <li> <!-- incorrect since <ul><li> is inside <span> -->
             ]]>
            </html>

            <textbox />

            <html><![CDATA[
                     </li>
                </ul>
```

```
            ]]>
      </html>
</zk>
```

If you need to generate the embedded HTML tags directly without the enclosing `SPAN` tag, you can use the xhtml component set or the native namespace as described in the following section.

Version History

Version	Date	Content

The native Namespace

Overview

With the native namespace, an XML element in a ZUML document will be interpreted as a native tag that will be sent to the browser directly, rather than becoming a ZK component.

For example,

```
<n:ul xmlns:n="native">
    <n:li>
    <textbox/>
    </n:li>
    <n:li>
    <textbox/>
    </n:li>
</n:ul>
```

will attach the following HTML tags to the browser's DOM tree:

```
<ul>
    <li>
    <input id="z_a3_2"/>
    </li>
    <li>
    <input id="z_a3_5"/>
    </li>
</ul
```

where `<input>` is the HTML tag(s) generated by the `textbox` component. Unlike `textbox` in the example above, ZK Loader doesn't really create a component for each of `ul` and `li`.[1] Rather, they are sent to the client directly. Of course, they must be recognizable by the client. For a HTML browser, they must be the valid HTML tags.

[1] ZK ZK actually creates a special component to represent as many XML elements with the native namespace as possible.

Dynamic Update

The XML elements associated with the native namespace will be considered as tags that the client accepts, and they are sent directly to the client to display. They are not ZK components, and they don't have the counterpart (widget) at the client either. The advantage is the better performance in term of both memory and processing time.

However, the disadvantage that is you cannot access or change them (neither component nor widget) dynamically. For example, the following code snippet is incorrect, since there is no component called `x`.

```
<n:ul id="x" xmlns:n="native"/>
<button label="add" onClick="new Li().setParent(x)"/> <!-- Failed since x is not availab
```

If you want to change them dynamically, you could:

1. Use client-side code to modify the browser's DOM tree at the client. Notice that, since ZK doesn't create the widget at the client too, you have to manipulate the DOM tree directly.
2. Use the html component if you won't mix ZUL with HTML tags.
3. Use the components from the XHTML component set as described in the following section.

For example, we could use jQuery to modify the DOM tree as follows:

```
<zk xmlns:n="native" xmlns:w="client">
      <n:input id="inp"/>
      <button label="change" w:onClick="jq('#inp')[0].value = 'clicked'"/>
</zk>
```

The rule of thumb is to use the native namespace if possible. If you need to change the content dynamically, you might consider the html component first. If still not applicable, use the XHTML component set.

Relation with Other Components

Though no component is associated with the element specified with the native namespace, you still could manipulate its parent, such as invalidate and move. For example, the following works correctly.

```
<window border="normal" title="Redraw">
  <n:ul xmlns:n="native">
     <n:li>ZK is simply best</n:li>
  </n:ul xmlns:n="native">
  <button label="Redraw" onClick="self.getParent().invalidate()"/><!-- OK to invalidate a
</window>
```

As shown, it is OK to invalidate a component even if it has some native tags.

Also notice that, though the native HTML tags will be generated for the native namespace, ZK Loader actually creates a component to represent as many as these native HTML tags. Thus, if you invoke Component.getPreviousSibling() (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zk/ ui/ Component. html#getPreviousSibling()) of the button above, it will return this component. However, don't access it since the real class/implementation of the component depends on the version you use and might be changed in the future.

In Pure Java

You could use the native namespace in Java too. For example, you could create a native table directly as follows.

```
import org.zkoss.zk.ui.Component;
import org.zkoss.zk.ui.HtmlNativeComponent;
import org.zkoss.zul.Datebox;

public class TableCreator {
    public void createTable(Component parent) {
        HtmlNativeComponent n =
            new HtmlNativeComponent("table", "<tr><td>When:</td><td>", "</td></tr>");
        n.setDynamicProperty("border", "1");
        n.setDynamicProperty("width", "100%");
        n.appendChild(new Datebox());
        parent.appendChild(n);
    }
}
```

As shown, the first argument of java.lang.String, java.lang.String) HtmlNativeComponent.HtmlNativeComponent(java.lang.String, java.lang.String, java.lang.String) (http:/ / www.zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zk/ ui/ HtmlNativeComponent. html#HtmlNativeComponent(java. lang. String,) is the name of tag. The second argument specifies the content that will be generated right before the children, if any. The third specifies the content after the children, if any. In addition, the setDynamicProperty method could be used to set the attributes of the tag.

In summary, the above example is equivalent to the following ZUML document:

```
<table border="1" width="100%" xmlns="native" xmlns:u="zul">
    <tr>
        <td>When:</td>
        <td><u:datebox/></td>
    </tr>
</table>
```

Output Tags with Another Namespace

If the HTML tag you want to output requires a XML namespace (such as XAML), you can use the following format to specify the URI of the XML namespace you want to output:

```
<element xmlns="native:URI-of-another-namespace">
```

For example, if you want to output the XAML tags directly to the client, you can specify XAML's XML namespace as follows.

```
<div>
      <Canvas xmlns="native:http://schemas.microsoft.com/client/2007">
        <TextBlock>Hello World!</TextBlock>
      </Canvas>
</div>
```

Then, the result DOM structure will be similar to the following[1] :

```
<div id="zk_uuid">
     <canvas xmlns="http://schemas.microsoft.com/client/2007">
          <textblock>Hello World!</textblock>
     </canvas>
</div>
```

[1] The real DOM structure of a component (div in this example) depends on its implementation. Here is only a simplified version.

Version History

Version	Date	Content

The XHTML Component Set

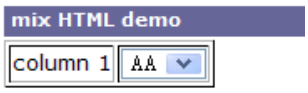
Overview

Like ZUL, the XHTML component set is a collection of components. Unlike ZUL, which is designed to have rich features, each XHTML component represents a HTML tag. For example, the following XML element will cause ZK Loader to create a component called Ul [1]

```
<h:ul xmlns:h="xhtml">
```

Dynamic Update

Because Components are instantiated for XML elements specified with the XHTML namespace, you could update its content dynamically on the server. For example, we could allow users to click a button to add a column as shown below.




```
<window title="mix HTML demo" xmlns:h="xhtml">
    <h:table border="1">
        <h:tr id="row1">
            <h:td>column 1</h:td>
            <h:td>
                <listbox id="list" mold="select">
                    <listitem label="AA"/>
                    <listitem label="BB"/>
                </listbox>
            </h:td>
        </h:tr>
    </h:table>
    <button label="add" onClick="row1.appendChild(new org.zkoss.zhtml.Td())"/>
</window>
```

On the other hand, the native namespace will cause *native* HTML tags being generated. It means you can not modify the content dynamically on the server. Notice that you still can handle them dynamically at the client.

However, when a XHTML component are used, a component running on the server has to be maintained. Thus, you should use the XHTML component set only if there is no better way for doing it.

For example, we could rewrite the previous sample with the native namespace and some client-side code as follows.

```
<window title="mix HTML demo" xmlns:n="native">
     <n:table border="1">
         <n:tr id="row1">
             <n:td>column 1</n:td>
             <n:td>
                 <listbox id="list" mold="select">
                     <listitem label="AA"/>
                     <listitem label="BB"/>
                 </listbox>
             </n:td>
         </n:tr>
     </n:table>
     <button label="add" w:onClick="jq('#row1').append('&lt;td>&lt;/td>')" xmlns:w="clien
 </window>
```

ID and UUID

Unlike other components, if you assign ID to a XHTML component, its UUID (Component.getUuid() [2]) is changed accordingly. It means you cannot have two XHTML components with the same ID, no matter if they are in different ID spaces.

Filename Extension

As described in ZUML, the XHTML component set is associated with zhtml, xhtml, html and htm. It means you could name a ZUML page as foo.zhtml if you map *.zhtml to ZK Loader. However, when this kind of file is interpreted, ZK Loader assumes it will have its own HTML, HEAD, BODY tags. On the other hand, these tags are generated automatically if the filename extension is zul.

For example, suppose we have a file called foo.zhtml, then the content might look as follows.

```
<?link rel="shortcut icon" href="/favicon.ico" type="image/x-icon"?>
<html xmlns:zk="zk" xmlns:z="zul">
  <head>
    <title>ZHTML Demo</title>
    <zkhead/><!-- a special tag to indicate where to generate ZK CSS and JS files -->
  </head>
  <body style="height:auto">
      <h1>ZHTML Demo</h1>
      <ul id="ul">
            <li>The first item.</li>
            <li id="li2" zk:onClick='self.setId("li2".equals(self.getId()) ? "":"li2")'>C
      </ul>
  </body>
```

```
</html>
```

where

1. Since the extension is zhtml, the default namespace is XHTML. Thus, we have to specify the zk and zul namespace explicitly.
   - Notice that we have to specify the zk namespace too, because XHTML will cause ZK Loader to consider any unrecognized element as native HTML tag.
2. We have to specify HTML, HEAD and BODY to make it a valid HTML document.
3. We could specify zkhead (line 5) to indicate where to generate ZK CSS and JavaScript files. It is optional. If not specified, ZK will try to identify the proper location for ZK CSS and JavaScript files. Specify it if you want some CSS or JavaScript file to be evaluated *before* or *after* ZK's default ones.
4. By default, BODY's CSS is width:100%;height:100%. However, it is appropriate for Web-look page[3] For Web-look, we could specify height:auto to reset it back to the browser's default.

---

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zhtml/Ul.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#getUuid()
[3] height:100% is more for desktop-application-look, such as using with <javadoc>org.zkoss.zul.Borderlayout.

Version History

Version	Date	Content

Long Operations

Events for the same desktop are processed sequentially. It simplifies the GUI programming and component development. However, it means an event handler that spends a lot of time to execute will block any following handlers. Worse of all, the user, due to the limitation of HTTP, got no hint but the small busy dialog shown at the left-top corner on the browser.

There are basically two approaches:

1. Handle everything in an event thread and have the user to wait but show more visible message to notice him
2. Handle the long operation in an independent thread, such that the user can access other functions

The first approach could be done with a technique called *echo events* as describe in the Use Echo Events section.

The second approach can be done in several ways, such as starting a working thread to do the long operation and then using a timer to check if the data ready and show to the client. However, there is a simple approach: use an event queue to run an asynchronous listener as described in the Use Event Queues section.

In additions to above approaches, there is a special mechanism called piggyback, which could be used to piggy back UI updates without extra network traffic.

Use Echo Events

Event echoing is useful for implementing a long operation.

As described in the previous section, HTTP is a request-and-response protocol, so the user won't see any feedback until the request has been served and responsed. Thus, if the processing of a request takes too long to execute, the user has no idea if the request is being process, or he doesn't, say, click the button successfully. The user usually tends to click again to ensure it is really clicked, but it only causes the server much slower to response.

The better approach is to send back some busy message to let the user know what happens during processing the long operation. It can be done easily with event echoing. If you prefer to allow the user to keep accessing other functions, please refer to the Use Event Queues section, which is powerful but more sophisticated to implement.

Event echoing for a long operation basically takes three steps

1. Invoke boolean) Clients.showBusy(java.lang.String, boolean) [1] to show a busy message and blocking the user from accessing any function
   - Of course, you could have any effect you like, such as showing a modal window. boolean) Clients.showBusy(java.lang.String, boolean) [1] is yet a built-in approach for showing the busy message.
2. Invoke org.zkoss.zk.ui.Component, java.lang.Object) Events.echoEvent(java.lang.String, org.zkoss.zk.ui.Component, java.lang.Object) [2] to echo an event
3. Listen to the event being echoed and do the long operation in the listener
   - At the end of the event listener, remember to remove the busy message, and update the UI if necessary

For example, assume the long operation is called `doLongOperation`, then:

```
<window id="w" width="200px" title="Test echoEvent" border="normal">
  <attribute name="onLater">
  doLongOperation(); //take long to execute
  Clients.showBusy(null, false); //remove the busy message
  </attribute>

  <button label="Echo Event">
  <attribute name="onClick">
  Clients.showBusy("Execute...", true); //show a busy message to user
  Events.echoEvent("onLater", w, null); //echo an event back
  </attribute>
  </button>
</window>
```

Better Feedback with Button's autodisable

With event echoing, it might still take hundreds of milliseconds to have the busy message, especially with the slow connection. The feedback to user can be further improved by the use of Button.setAutodisable(java.lang.String) [3]. For example,

```
  <button label="Echo Event" autodisable="self">
...
```

Then, the button will be disabled automatically when it is pressed, and enabled automatically when the request has been served.

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Clients.html#showBusy(java.lang.String,
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Events.html#echoEvent(java.lang.String,
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Button.html#setAutodisable(java.lang.String)

Use Event Queues

The event queue provides a simple way to execute a so-called asynchronous event listener in parallel to other event listeners. Thus, it won't block the user from accessing other functions even if the asynchronous event listener spends a lot of time to execute.

The event queue eventually starts a working thread to invoke the asynchronous event listener, though it is transparent to the caller. Thus, it cannot be used in the environment that does not allow the use of working threads, such as Google App Engine [1].

In addition, it will start a server push automatically to send the UI updates back when it is ready. If you prefer to use the client polling or particular implementation, you could start it manually by by use of DesktopCtrl.enableServerPush(org.zkoss.zk.ui.sys.ServerPush) [2], such as:

```
((DesktopCtrl)desktop).enableServerPush(
    new org.zkoss.zk.ui.impl.PollingServerPush(2000,5000,-1));
```

Example

We provide two implementations to illustrate how to use the event queue's asynchronous listener for executing a long operation. The first approach is more generic that you can modify it to use more diverse situations. On the other hand, the second approach is much simpler. If you don't have time, you could skip the first approach and study the second approach.

A Generic Approach

A typical use case is to subscribe an asynchronous event listener for doing the long operation, and to subscribe a synchronous event listener to update the user interface. Then, when starting a long operation, an event is posted to the asynchronous event listener for processing. Since the invocation is asynchronous, the user can still interact with ZK smoothly. At the end of the invocation of the asynchronous event listener, it published an event to the synchronous event listener to update the result of the long operation back to the browser.

For example,

```
<window title="test of long operation" border="normal">
      <html><![CDATA[
      <ul>
      <li>Click the button it will start a long operation.</li>
      <li>With this implementation, you can press the button again even if
      the long operation is still being processed</li>
      </ul>
```

```
      ]]></html>
      <zscript>
      void print(String msg) {
            new Label(msg).setParent(inf);
      }
      </zscript>
      <button label="async long op">
            <attribute name="onClick"><![CDATA[
   if (EventQueues.exists("longop")) {
     print("It is busy. Please wait");
     return; //busy
   }

   EventQueue eq = EventQueues.lookup("longop"); //create a queue
   String result;

   //subscribe async listener to handle long operation
   eq.subscribe(new EventListener() {
     public void onEvent(Event evt) {
       if ("doLongOp".equals(evt.getName())) {
         org.zkoss.lang.Threads.sleep(3000); //simulate a long
operation
         result = "success"; //store the result
         eq.publish(new Event("endLongOp")); //notify it is done
       }
     }
   }, true); //asynchronous

   //subscribe a normal listener to show the resul to the browser
   eq.subscribe(new EventListener() {
     public void onEvent(Event evt) {
       if ("endLongOp".equals(evt.getName())) {
             print(result); //show the result to the browser
             EventQueues.remove("longop");
           }
         }
   }); //synchronous

   print("Wait for 3 seconds");
   eq.publish(new Event("doLongOp")); //kick off the long operation
            ]]></attribute>
      </button>
      <vbox id="inf"/>
</window>
```

An asynchronous event listener is *not* allowed to access the desktop, but it is allowed to invoke EventQueue.publish(org.zkoss.zk.ui.event.Event) [3] to publish an event.

A Simpler Approach

While subscribing the asynchronous and synchronous event listeners separately is geneic, as illustrated above, the event queue provides a simple method to allow you register them in one invocation: org.zkoss.zk.ui.event.EventListener) EventQueue.subscribe(org.zkoss.zk.ui.event.EventListener, org.zkoss.zk.ui.event.EventListener) [4]. In additions, you don't need to publish an event at the end of the asynchronous event listener -- the synchronous event listener is invoked automatically.

```
<window title="test of long operation" border="normal">
      <zscript>
      void print(String msg) {
            new Label(msg).setParent(inf);
      }
      </zscript>
      <button label="async long op">
            <attribute name="onClick"><![CDATA[
   if (EventQueues.exists("longop")) {
     print("It is busy. Please wait");
     return; //busy
   }

   EventQueue eq = EventQueues.lookup("longop"); //create a queue
   String result;

   //subscribe async listener to handle long operation
   eq.subscribe(new EventListener() {
     public void onEvent(Event evt) { //asynchronous
       org.zkoss.lang.Threads.sleep(3000); //simulate a long operation
       result = "success"; //store the result
     }
   }, new EventListener() { //callback
     public void onEvent(Event evt) {
       print(result); //show the result to the browser
            EventQueues.remove("longop");
          }
   });

   print("Wait for 3 seconds");
   eq.publish(new Event("whatever")); //kick off the long operation
            ]]></attribute>
      </button>
      <vbox id="inf"/>
</window>
```

Better Feedback with Button's autodisable

In the above example, we displayed a message if the button was pressed twice. If you prefer to simply disable the button, you could use Button.setAutodisable(java.lang.String) [3]. For example,

```
  <button label="async long op" autodisable="+self">
...
```

Then, the button will be disabled automatically when it is pressed. Notice that we prefix `self` with `+`, and it means you have to enable it manually (once it is OK to run again).

```
if (ready)
    button.setDisabled(false); //enable it when ready
```

Version History

Version	Date	Content

References

[1] http://code.google.com/appengine/
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/sys/DesktopCtrl.html#enableServerPush(org.zkoss.zk.ui.sys.ServerPush)
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/EventQueue.html#publish(org.zkoss.zk.ui.event.Event)
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/EventQueue.html#subscribe(org.zkoss.zk.ui.event.EventListener,

Use Piggyback

Sometimes it is not hurry to update the result to a client. Rather, the UI update could be sent back when the user, say, clicks a button or trigger some request to the server. This technique is called *piggyback*.

In piggyback, all you need to do is to register an event listener for the `onPiggyback` event to one of the root components. Then, the listener will be invoked each time ZK Update Engine has processed an AU request.

For example, suppose we have a long operation which is processed in a working thread, then:

```
<window id="main" title="Working Thread" onPiggyback="checkResult()">
    <zscript>
     List result = Collections.synchronizedList(new LinkedList());

     void checkResult() {
         while (!result.isEmpty())
             main.appendChild(result.remove(0));
     }
    </zscript>
    <timer id="timer" />
    <button label="Start Working Thread">
        <attribute name="onClick">
                timer.start();
                new test.WorkingThread(desktop, result).start();
        </attribute>
    </button>
```

```
</window>
```

The advantage of the piggyback is no extra traffic between the client and the server. However, the user sees no updates if he doesn't have any activity, such as clicking. Whether it is proper is really up to the application requirements.

> **Note**: A deferrable event won't be sent to the client immediately, so the `onPiggyback` event is triggered only if a non-deferrable event is fired. For more information, please refer to the Deferrable Event Listeners section.

Version History

Version	Date	Content

Communication

Here we dicuss how to communicate among pages, desktops and Web applications.

Inter-Page Communication

Communicating among pages in the same desktop is straightforward. First, you can use attributes to share data. Second, you can use event to notify each other.

Identify a Page

To communicate among pages, we have to assign an identifier to the target page. In ZUML, it is done by the use of the page directive:

```
<?page id=" foo"?>
<window id="main"/>
```

Then we could retrieve it by use of Desktop.getPage(java.lang.String) [1] or by use of a utility class called Path [8]. For example, the following statements could access the *main* window above:

```
comp.getDesktop().getPage("foo").getFellow("main");
Path.getComponent("//foo/main");
```

As shown, Path.getComponent(java.lang.String) [2] considers an ID starting with double slashes as a page's ID.

Use Attributes

Each component, page, desktop, session and Web application has an independent map of attributes. It is a good place to share data among components, pages, desktops and even sessions.

In Java , you could use "setAttribute()","removeAttribute()" and "getAttribute()" of Component [1], Page [8] and so on to share data. Another way is using the scope argument to identify which scope you want to access. (In the following example, assuming comp is a component.)

```
 comp.setAttribute("some","anyObject");
 comp.getAttribute("some", comp.DESKTOP_SCOPE);
 comp.getDesktop().getAttribute("some"); //is equivalent to previous
line
```

In `zscript` and EL expressions, you could use the implicit objects: `componentScope`, `pageScope`, `desktopScope`, `sessionScope`, `requestScope` and `applicationScope`.

```
<window>
    <zscript><![CDATA[
        desktop.setAttribute("some","anyObject");
        desktopScope.get("some");
    ]]></zscript>
    1:${desktopScope["some"]}
</window>
```

Post and Send Events

You could communicate among pages in the same desktop. The way to communicate is to use the Events.postEvent(org.zkoss.zk.ui.event.Event) [3] or Events.sendEvent(org.zkoss.zk.ui.event.Event) [4] to notify a component in the target page.

For example,

```
 Events.postEvent(new Event("SomethingHappens",
     comp.getDesktop().getPage("foo").getFellow("main"));
```

You can also pass the data with the event object. The third parameter in Events.postEvent(org.zkoss.zk.ui.event.Event) [3] will be put into Event.getData() [5]. You could the data you want with it.

```
Events.postEvent("onTest", target, "this will be send");
```

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Desktop.html#getPage(java.lang.String)
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Path.html#getComponent(java.lang.String)
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Events.html#postEvent(org.zkoss.zk.ui.event.Event)
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Events.html#sendEvent(org.zkoss.zk.ui.event.Event)
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Event.html#getData()

Inter-Desktop Communication

Unlike pages, you cannot access two desktops at the same time. You cannot send or post an event from one desktop to another directly either. Rather, we have to use an event queue with a proper scope, such as group, session or application -- depending on where the other desktop is located.

Desktops in the Same Browser Window

In most cases, each browser window has at most one desktop. However, it is still possible to have multiple desktops in one browser window:

- Use HTML IFRAME or FRAMESET to integrate multiple ZUML pages
- Use a portal server to integrate multiple ZK portlets
- Assembly multiple ZUML pages at the client, such as the templating technology described in this section

In this case, you could communicate among desktops by the use of an event queue with the group scope (EventQueues.DESKTOP [1]).

```
EventQueue que = EventQueues.lookup("groupTest", EventQueues.GROUP,
true);
que.subscribe(new EventListener() {
      public void onEvent(Event evt) {
            //receive event from this event queue (within the same
group of desktops)
      }
});
```

Notice that the desktop-scoped event queue does not require Server Push, so there is no performance impact at all.

Here is a dumb example: chat among iframes.

```
<!-- main -->
<window title="main" border="normal" onOK="publish()">
      <zscript>
      EventQueue que = EventQueues.lookup("groupTest", "group", true);
      que.subscribe(new EventListener() {
            public void onEvent(Event evt) {
                  o.setValue(o.getValue() + evt.getData() + "\n");
            }
```

```
	});
	void publish() {
		String text = i.getValue();
		if (text.length() > 0) {
			i.setValue("");
			que.publish(new Event("onGroupTest", null, text));
		}
	}
	</zscript>
	Please enter:
	<textbox id="i" onChange="publish()"/>
	<textbox id="o" rows="6"/>
	<separator/>
	<iframe src="includee.zul" height="500px" width="30%"/>
	<iframe src="includee.zul" height="500px" width="30%"/>
	<iframe src="includee.zul" height="500px" width="30%"/>
</window>
```

And, this is the ZUML page being referenced (by iframe).

```
<!-- includee.zul -->
<window title="frame2" border="normal" onOK="publish()">
	<zscript>
		EventQueue que = EventQueues.lookup("groupTest", "group",
true);
		que.subscribe(new EventListener() {
			public void onEvent(Event evt) {
				o.setValue(o.getValue() + evt.getData() +
"\n");
			}
		});
		void publish() {
			String text = i.getValue();
			if (text.length() > 0) {
				i.setValue("");
				que.publish(new Event("onGroupTest", null,
text));
			}
		}
	</zscript>
	<textbox id="i" onChange="publish()"/>
	<textbox id="o" rows="6"/>
</window>
```

Desktop in Different Sessions

Similarly, we could use an event queue to communicate among desktops belonging to different sessions. The only difference is to specify EventQueues.APPLICATION [2] as the scope.

```
EventQueue que = EventQueues.lookup("groupTest",
EventQueues.APPLICATION, true);
```

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/EventQueues.html#DESKTOP
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/EventQueues.html#APPLICATION

Inter-Application Communication

An EAR file or the installation of Web server could have multiple WAR files. Each of them is a Web application. There are no standard way to communicate between two Web applications. However, there are a few ways to work around it.

Use ZK Specific URI: ~app/

ZK supports a way to reference the resource from another Web applications. For example, assume you want to include a resource, say `/foreign.zul`, from another Web application, say `app2`. Then, you could do as follows.

```
<include src="~app2/foreign.zul"/>
```

Similarly, you could reference resources from another Web application.

```
<style src="~app2/foo.css"/> <!-- assume foo.css is in the context called app2 -->
<image src="~/foo.png"/> <!-- assume foo.png is in the root context -->
```

Note: Whether you can access a resource located in another Web application depends on the configuration of the Web server. For example, you have to specify `crossContext="true"` in conf/context.xml, if you are using Tomcat.

Use Cookie

Cookie [1] is another way to communicate among Web applications. It can be done by setting the path to "/", such that every Web application in the same host will see it.

```
HttpServletResponse response =
(HttpServletResponse)Executions.getCurrent().getNativeResponse();
Cookie userCookie = new Cookie("user", "foo");
userCookie.setPath("/");
response.addCookie(userCookie);
```

Web Resources from Classpath

Though it is not necessary for inter-application communication, you could, with ZK, reference a resource that is locatable by the classpath. The advantage is that you could embed Web resources in a JAR file, which simplifies the deployment.

```
<image src="~./my/jar.gif"/>
```

Then, it tries to locate the resource, `/my/jar.gif`, at the `/web` directory by searching resources from the classpath. Notice that `WEB-INF/classes` is also part of the classpath, so you could put it under `WEB-INF/classes/web/my/jar.gif` too.

Version History

Version	Date	Content

References

[1] http://en.wikipedia.org/wiki/HTTP_cookie

Templating

Templating is a technique that allows developers to define UI fragments, and how to assemble them into a complete UI at runtime. With ZK, it can be done by the use of annotations and composers (or initators, utilInitiator [1]).

In general, templating can be done by specifying the name of a fragment as annotations in a ZUML document that shall represent a complete UI, and a composer that is capable to parse annotations and replace them with the fragment. For example,

```
<div apply="foo.MyTemplateManager"><!-- your template manager -->
    <include src="@{header}"/><!-- you could use any component as long as your manager kn
    <include src="@{content}"/>
    <include src="@{footer}"/>
</div>
```

Here is a list of the implementations that ZK supports by default. You could implement your own, if it does not fulfill your requirement. If the templating is stateful and dynamical, you might consider ZK Spring [13] for using Spring Web Flow instead.

References

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

Composition

Composition [1] is one of the built-in templating implementations. The concept is simple:

1. Define a template (a ZUML document representing a complete UI)
2. Define a ZUML document that contains a collections of fragments that a template might reference

Notice that the user shall visit the ZUML document with a collection of fragments rather than the template document.

The advantage of Composition [1] is that you don't need additional configuration file.

Note: the composition doesn't support to mix up with ZUML and ZHTML language, that is, if you define a ZHTML template as the HTML content that contains *Html* and *Body* tags, you cannot use that template in a ZUML page.

Defines a Template

A template document is a ZUML document that defines how to assemble the fragments. For example,

```
<!-- /WEB-INF/layout/template.zul -->
<vbox>
  <hbox self="@{insert(content)}"/>
  <hbox self="@{insert(detail)}"/>
</vbox>
```

As shown, the anchor (i.e., the component that a fragment will insert as children) is defined by specify an annotation as @{insert(*name*)}. Then, when Composition [1] is applied to a ZUML document with a collections of fragments, the matched fragment will become the child of the annotated component (such as hbox in the above example).

Define Fragments

To apply a template to a ZUML document that an user visits, you have to defined a collection of fragments that a template might use, and then specify Composition [1] as one of the initiators of the document:

```
<!-- foo/index.zul -->
<?init class="org.zkoss.zk.ui.util.Composition"
arg0="/WEB-INF/layout/template.zul"?>
<zk>
  <window self="@{define(content)}" title="window1" width="100px"/>
  <window self="@{define(content)}" title="window2" width="200px"/>
  <grid self="@{define(detail)}" width="300px" height="100px"/>
</zk>
```

As shown, a fragment is defined by specifying an annotation as self="@{define(*name*)}. Furthermore, the template is specified in the init directive.

Then, when the user visits this page (foo/index.zul in the above example), Composition [1] will do:

1. Load the template, and render it as the root components of this page(foo/index.zul)
2. Move the fragments specified in this page to become the children of the anchor component with the same annotation name

Thus, here is the result

```
<vbox>
  <hbox>
    <window title="window1" width="100px"/>
    <window title="window2" width="200px"/>
  </hbox>
  <hbox>
    <grid width="300px" height="100px"/>
  </hbox>
</vbox>
```

Multiple Templates

You could apply multiple templates to a single page too:

```
<?init class="org.zkoss.zk.ui.util.Composition"
arg0="/WEB-INF/layout/template0.zul"
arg1="/WEB-INF/layout/template1.zul"?>
```

The templates specified in `arg0` and `arg1` (etc.) will be loaded and rendered one-by-one.

Grouping Fragments into Separated Files

In a complex templating environment, it might not be appropriate to put fragments in the target page (e.g., foo/index.zul in the above example), since you might want to use the same collection of fragments in several target pages. It can be easily by use of the include component as follows.

```
<!-- foo/index.zul -->
<?init class="org.zkoss.zk.ui.util.Composition"
arg0="/WEB-INF/layout/template.zul"?>
<include src="/WEB-INF/layout/fragments.zul"/>
```

Then, you could group fragments into one or multiple individual ZUL documents, such as

```
<!-- /WEB-INF/layout/fragments.zul -->
<zk>
  <window self="@{define(content)}" title="window1" width="100px"/>
  <window self="@{define(content)}" title="window2" width="200px"/>
  <grid self="@{define(detail)}" width="300px" height="100px"/>
</zk>
```

Version History

Version	Date	Content

References

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

Templates

As described in the MVC: Template section, a template is a ZUML fragment that defines how to create components. A template is enclosed with the template element as shown below.

```
<window>
    <template name="foo">
      <textbox/>
      <grid model=${data}>
         <columns/>
         <template name="model"> <!-- nested template -->
            <row>Name: <textbox value=${each.name}"/></row>
         </template>
      </grid>
   </template>
...
```

Using Template in Application

"Template" is a generic feature and its use is not limited to custom model rendering. Users are able to use "template" in ZK applications too.

Each template is stored as part of a component and can be retrieved it by invoking the Component.getTemplate(java.lang.String) [1]. To create the components defined in the template, just invoke the org.zkoss.zk.ui.Component, org.zkoss.xel.VariableResolver, org.zkoss.zk.ui.util.Composer) Template.create(org.zkoss.zk.ui.Component, org.zkoss.zk.ui.Component, org.zkoss.xel.VariableResolver, org.zkoss.zk.ui.util.Composer) [2]. For example,

```
comp.getTemplate("foo").create(comp, null, null, null);
```

The third argument of the create method is a variable resolver (VariableResolver [2]). Depending on the requirement, you could pass any implementation you like. For example, the implementation of a listbox actually utilizes it to return the data being rendered; the code is similar to the following (for easy understanding, the code has been simplified).

For more detailed information about the variable resolver, please refer to ZUML Reference.

```
public class TemplateBasedRenderer implements ListitemRenderer {
      public void render(Listitem item, final Object data, int index) {
            final Listbox listbox = (Listbox)item.getParent();
            final Component[] items =
listbox.getTemplate("model").create(listbox, item,
                  new VariableResolver() {
                        public Object resolveVariable(String name) {
                              return "each".equals(name) ? data: null;
                        }
                  }, null);

            final Listitem nli = (Listitem)items[0];
            if (nli.getValue() == null) //template might set it
                  nli.setValue(data);
```

```
            item.detach();
        }
}
```

In addition, the template allow users to specify any number of parameters with any name, and these parameters can be retrieved back by the `getParameters` method of the `Template` interface:

```
<template name="foo" var1="value1" var2="${el2}">
...
</template>
```

If the content of a template is located elsewhere as a separated file, to reference it, specify it in the `src` attribute as follows.

```
<template name="foo" src="foo.zul">
...
</template>
```

Children Binding

ZK Data Binding provides a powerful way called Children Binding to render a template based on the data (such as a list of elements). Moreover, the UI will be updated automatically if the data has been changed. For more information, please refer to the Children Binding section.

Version History

Version	Date	Content
6.0.0	July 2011	The template feature was introduced.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#getTemplate(java.lang.String)

XML Output

In addition to generating HTML output to a browser, ZK could be used to generate (static) XML output to any client that recognizes it, such as RSS [1] and Web Services [2].

Using ZK to generate XML output is straightforward:

1. Uses the XML component set (http://www.zkoss.org/2007/xml and shortcut is `xml`).
2. Maps the file extension to ZK Loader
3. Maps the file extension to the XML component set

The XML component set also provides some special components, such as transformer that supports XSTL. Fore more information please refer to XML Components.

Use the XML Component Set

The XML component set (aka., the XML language, in ZK terminology) is used to generate XML output. Unlike the ZUL or XHTML component sets, all unknown[3] tags in a ZUML document are assumed to belong the native namespace. It means ZK generates them directly to the output without instantiating a ZK component for each of them.

The following is an example that generates the SVG output. It looks very similar to the XML output you want to generate, except you can use zscript, EL expressions, macro components and other ZK features.

```
<?page contentType="image/svg+xml;charset=UTF-8"?>
<svg width="100%" height="100%" version="1.1" xmlns="http://www.w3.org/2000/svg"
xmlns:z="zk">
    <z:zscript><![CDATA[
    String[] bgnds = {"purple", "blue", "yellow"};
    int[] rads = {30, 25, 20};
    ]]></z:zscript>
    <circle style="fill:${each}" z:forEach="${bgnds}"
        cx="${50+rads[forEachStatus.index]}"
        cy="${20+rads[forEachStatus.index]}"
        r="${rads[forEachStatus.index]}"/>
</svg>
```

The generated output will be

```
<svg xmlns="http://www.w3.org/2000/svg" width="100%" height="100%"
     version="1.1">
     <circle style="fill:purple" cx="80" cy="50" r="30">
     </circle>
     <circle style="fill:blue" cx="75" cy="45" r="25">
     </circle>
     <circle style="fill:yellow" cx="70" cy="40" r="20">
     </circle>
</svg>
```

where

- The content type is specified with the page directive. For SVG, it is `image/svg+xml`. The `xml` processing instruction (`<?xml?>`) and `DOCTYPE` of the output are also specified in the `page` directive.
- All tags in this example, such as `svg` and `circle`, are associated with a namespace (http://www.w3.org/2000/svg) that is unknown to ZK Loader. Thus, they are assumed to belong the native namespace. They are output directly rather than instantiating a ZK component for each of them.
- To use `zscript`, `forEach` and other ZK specific features, you have to specify the ZK namespace (zk).

[1] http://www.whatisrss.com/
[2] http://en.wikipedia.org/wiki/Web_service
[3] By the unknown tag we mean a XML element that is not associated with a XML namespace, or the namespace is unknown.

Maps the File Extension to ZK Loader

To let ZK Loader process the file, you have to associate it with the ZK Loader in `WEB-INF/web.xml`. In this example, we map all files with the `.svg` extension to ZK Loader[1]:

```
<servlet-mapping>
    <servlet-name>zkLoader</servlet-name>
    <url-pattern>*.svg</url-pattern>
</servlet-mapping>
```

[1] We assume ZK Loader (`zkLoader`) is mapped to `org.zkoss.zk.ui.http.DHtmlLayoutServlet`.

Maps the File Extension to the XML Component Set

Unless the file extension is `.xml`, you have to associate it with the XML component set (aka., the XML language) explicitly in `WEB-INF/zk.xml`. In this example, we map `.svg` to the XML component set:

```
<language-mapping>
    <language-name>xml</language-name>
    <extension>svg</extension>
</language-mapping>
```

where `xml` is the language name of the XML component set. Thus, when ZK Loader parses a file with the `.svg` extension, it knows the default language is the XML component set[1].

[1] For more information about language identification, please refer to ZUML Reference.

Version History

Version	Date	Content

Event Threads

By default, ZK processes an event in the same Servlet thread that receives the HTTP request. It is the suggested approach because the performance is better and it is easy to integrate with other frameworks[1] .

However, it also implies the developer cannot suspend the execution. Otherwise, the end user won't see any updates from the server. To solve it, ZK provides an alternative approach: processes the event in an independent thread called the event processing thread. Therefore, the developer can suspend and resume the execution at any time, without blocking the Servlet thread from sending back the responses to the browser. To turn it on[2] , you have to specify the following in `WEB-INF/zk.xml` (ZK Configuration Guide: disable-event-thread).

```
<system-config>
    <disable-event-thread>false</disable-event-thread>
</system-config>
```

In short, it is recommended to disable the event thread. Enable the event thread only if the project does not need to integrate other frameworks (such as Spring), depends on Messagebox [1] and modal windows a lot, and does not have a lot of concurrent users.

Here is the advantages and limitations about using the Servlet thread to process events. In the following sections we will talk more about the limitations and workarounds when using the Servlet thread.

	Using Servlet Thread	Using Event Processing Thread
Integration	Less integration issues. Many containers assume the HTTP request is handled in the Servlet thread, and many frameworks store per-request information in the thread local storage.	You may have to implement `EventThreadInit` and/or `EventThreadCleanup` to solve the integration issue, such as copying the per-request information from the Servlet thread to the event processing thread. Threre are several implementations to solve the integration issue, such as HibernateSessionContextListener [3] (they can be found under the org.zkoss.zkplus package [4]).
SuspendResume	No way to suspend the execution of the event listener. For example, you cannot create a modal window.	No limitation at all.
Performance	No extra cost	It executes a bit slower to switch from one thread to another, and it might consume a lot more memory if there are a lot of suspended event processing threads.

[1] Many frameworks store per-request information in the thread-local storage, so we have to copy them from Servlet thread to the Event Processing Thread.

[2] For ZK 1.x, 2.x and 3.x, the event processing thread is enabled by default.

[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/hibernate/HibernateSessionContextListener.html#

[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/package-summary.html

Modal Windows

Modal Windows with Servlet Thread

When the event is processed in the Servlet thread (default), the execution cannot be suspended. Thus, the modal window behaves the same as the highlited window (Window.doHighlighted() [1]). At the client side, the visual effect is the same: a semi-transparent mask blocks the end user from access components other than the modal window. However, at the server side, it works just like the overlapped mode – it returns immediately without waiting for user's closing the window.

```
win.doModal(); //returns once the mode is changed; not suspended
System.out.println("next");
```

The "next" message will be printed to the console before the end user closes the modal window.

Modal Windows with Event Thread

If the event thread is enabled, Window.doModal() [2] will suspend the current thread. Thus, the "next" message won't be shown, until the modal window is closed.

When the event thread is suspended, the Servlet thread will be resumed and continue to loork another event thread to process other events, if any. Thus, the end user still have the control (such that he can close the modal window if he want).

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Window.html#doHighlighted()
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Window.html#doModal()

Message Box

Message Boxes with Servlet Thread

When Messagebox.show(java.lang.String) [1] is called, it returns immediately after showing the message dialog. Furthermore, it always returns `Messagebox.OK`. Thus, it is meaningless to show buttons other than the OK button. For example, the `if` clause in the following example is never true.

```
if (Messagebox.show("Delete?", "Prompt", Messagebox.YES|Messagebox.NO,
    Messagebox.QUESTION) == Messagebox.YES) {
    this_never_executes();
}
```

Rather, you have to provide an event listener as follows.

```
Messagebox.show("Delete?", "Prompt", Messagebox.YES|Messagebox.NO,
    Messagebox.QUESTION,
    new EventListener() {
        public void onEvent(Event evt) {
            switch (((Integer)evt.getData()).intValue()) {
            case Messagebox.YES: doYes(); break; //the Yes button is
pressed
            case Messagebox.NO: doNo(); break; //the No button is
pressed
            }
        }
    }
);
```

The event listener you provided is invoked when the user clicks one of the buttons. Then, you can identify which button is clicked by examining the data (Event's `getData`). The data is an integer whose value is the button's identifier, such as `Messagebox.YES`.

Alternatively, you can examine the event name:

```
public void onEvent(Event evt) {
    if ("onYes".equals(evt.getName())) {
        doYes(); //the Yes button is pressed
    } else if ("onNo".equals(evt.getName())) {
        doNo(); //the No button is pressed
    }
}
```

Note: The event name for the OK button is `onOK`, not `onOk`. **Notice**: If you want to make it run under clustering environment, you shall implement SerializableEventListener [2]. For more information, please refer to ZK Developer's Reference: Clustering.

Message Boxes with Event Thread

If the event thread is enabled, Messagebox.show(java.lang.String) [1] will suspend the thread until the end user makes the choice. Thus, the following code works correctly.

```
if (Messagebox.show("Delete?", "Prompt", Messagebox.YES|Messagebox.NO,
    Messagebox.QUESTION) == Messagebox.YES) {
    //execute only if the YES button is clicked
}
```

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Messagebox.html#show(java.lang.String)
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/SerializableEventListener.html#

File Upload

File Upload with Servlet Thread

When the event thread is disable (default), it is recommended to use Button [2], Toolbarbutton [1] or Menuitem [2] with upload="true" instead. For example,

```
<zk>
      <zscript>
      void upload(UploadEvent event) {
            org.zkoss.util.media.Media media = event.getMedia();
            if (media instanceof org.zkoss.image.Image) {
                  org.zkoss.zul.Image image = new
org.zkoss.zul.Image();
                  image.setContent( (org.zkoss.image.Image) media);
                  image.setParent(pics);
            } else {
                  Messagebox.show("Not an image: "+media, "Error",
Messagebox.OK, Messagebox.ERROR);
            }
      }
      </zscript>
      <button label="Upload" upload="true" onUpload="upload(event)"/>
      <toolbarbutton label="Upload" upload="true" onUpload="upload(event)"/>
      <vbox id="pics" />
</zk>
```

If you prefer to use a dialog (Fileupload.get() [1]), please take a look at ZK Component Reference: Fileupload for more inormation.

File Upload with Event Thread

If the event thread is disabled, the developer can use Button [2] or Toolbarbutton [1] with upload="true" instead. They behaves the same no matter the event thread is disabled or not.

However, if the event thread is disabled, it is convenient to use Fileupload.get() [1] and other static methods.

```
<zk>
	<button label="Upload">
	<attribute name="onClick">{
		org.zkoss.util.media.Media[] media = Fileupload.get(-1);
		if (media != null) {
			for (int i = 0; i &lt; media.length; i++) {
				if (media[i] instanceof org.zkoss.image.Image)
{
					org.zkoss.zul.Image image = new
org.zkoss.zul.Image();
					image.setContent(media[i]);
					image.setParent(pics);
				} else {
					Messagebox.show("Not an image:
"+media[i], "Error", Messagebox.OK, Messagebox.ERROR);
					break; //not to show too many errors
				}
			}
		}
	}</attribute>
	</button>
	<vbox id="pics" />
</zk>
```

As shown, Fileupload.get(int) [2] won't return until the end user uploads the files (and/or closes the dialog).

Version History

Version	Date	Content

:

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Fileupload.html#get()
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Fileupload.html#get(int)

Theming and Styling

Depending on the requirement, there are different ways to customize the look and feel of components.

- Molds

  A component could have multiple different appearance, such as accordion vs regular tabbox. Each appearance is called a mold. You could choose one fulfilling your need.
- CSS

  To fine-tune the look and feel of a particular component, you could specify CSS styles and classes without changing the DOM structure at the client.
- Theme Customization

  The default themes (breeze, classic blue.. provided by ZK) allows some generic customization, such as font size. In additions, you could customize CSS and also DOM structures.
- Theme Provider

  If you allow users to have different themes they prefer to use, you could implement a theme provider to allow them to switch among the themes you provide.

The CSS styling really depends on the implementation of the component (and the mold). It is suggested to refer to ZK Style Guide. In addition, if you have any doubt, you could use the HTML or CSS inspector shipped with the browser, such as Firebug [1] for Firefox, and Developer Tools [2] for Internet Explorer, to investigate how CSS styles are used.

References

[1] http://getfirebug.com/

[2] http://en.wikipedia.org/wiki/Internet_Explorer_Developer_Toolbar

Molds

A component could have multiple different visual appearances. Each appearance is called a mold. A mold is basically a combination of a DOM structure plus CSS. That is, it is the visual representation of a component. Developers could dynamically change the mold by use of Component.setMold(java.lang.String) [1].

All components support at least a mold called `default`, which is the default value. Some components might have support two or more molds. For example, tabbox supports both `default` and `accordion` molds.

If `tabbox`'s `mold` is not set, it uses the default mold.

```
<tabbox>
	<tabs>
		<tab label="First tab" />
		<tab label="Second tab" />
	</tabs>
	<tabpanels>
		<tabpanel>The first panel.</tabpanel>
		<tabpanel>The second panel.</tabpanel>
	</tabpanels>
</tabbox>
```

And you could set `tabbox`'s mold to "accordion" to change the look.

```
<tabbox mold="accordion">
	<tabs>
		<tab label="First tab" />
		<tab label="Second tab" />
	</tabs>
	<tabpanels>
		<tabpanel>The first panel.</tabpanel>
		<tabpanel>The second panel.</tabpanel>
	</tabpanels>
</tabbox>
```

For more information about component's molds, please refer to ZK Component Reference.

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#setMold(java.lang.String)

CSS Classes and Styles

CSS (Cascading Style Sheets [1]) is a style sheet language used to describe the presentation of a (HTML) document. It is an important part of ZK to customize component's look and feel. If you are not familiar with CSS, please refer to CSS Tutorial [2].

There are a set of methods that could be used to set CSS styles for an individual component.

- HtmlBasedComponent.setStyle(java.lang.String) [3] assigns CSS styles directly to a component
- HtmlBasedComponent.setSclass(java.lang.String) [4] (i.e., sclass) assigns one or multiple CSS style classes to a component.
- HtmlBasedComponent.setZclass(java.lang.String) [5] (i.e., zclass) assigns the main CSS style class to a component. Unlike style and sclass, if zclass is changed, all default CSS styles won't be applied.
- Some components have a so-called content area and they have a separated set of methods to change the CSS style of the content area, such as Window.setContentStyle(java.lang.String) [6] and Window.setContentSclass(java.lang.String) [7].

Notice that the DOM structures of many ZUL components are complicate, and CSS customization might depend on the DOM structure. For more information about how individual component is styled, please refer to ZK Style Guide.

style

Specifying the style is straightforward:

```
<textbox style="color: red; font-style: oblique;"/>
```

or, in Java:

```
Textbox tb = new Textbox();
tb.setStyle("color: red; font-style: oblique;");
```

sclass

In addition, you could specify the style class by use of HtmlBasedComponent.setSclass(java.lang.String) [4], such that you could apply the same CSS style to multiple components.

```
<window>
    <style>
        .red {
            color: blue;
            font-style: oblique;
        }
    </style>
    <textbox sclass="red" /> <!-- first textbox -->
    <textbox sclass="red" /> <!-- another textbox -->
</window>
```

You could apply multiple style classes too. As shown below, just separate them with a space.

```
<textbox sclass="red error"/>
```

zclass

Like sclass, zclass is used to specify the CSS style class. However, zclass is the main CSS that each mold of each component has. If it is changed, all default CSS of the given component won't be applied. In other words, you have to provide a full set of CSS rules that a component's mold has.

Rule of thumb: specify zclass if you want to customize the look completely. Otherwise, use sclass to customize one or a few CSS styles.

For more more information, please refer to ZK Style Guide.

content style and sclass

Some container component such as window, groupbox, detail have a content block, you have to use `contentStyle` to set its style.

For example,

```
<window title="below is content"  contentStyle="background:yellow">
    Hello, World!
</window>
```

Scrollable Window

A typical use of contentStyle is to make a window scrollable as follows.

```
<window title="Scroll Example" width="150px" height="100px" contentStyle="overflow:auto"
This is a long line to spread over several lines, and more content to
display.
Finally, the scrollbar becomes visible.
This is another line.
</window>
```

Version History

Version	Date	Content

References

[1] http://en.wikipedia.org/wiki/Cascading_Style_Sheets
[2] http://www.w3schools.com/css/default.asp
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlBasedComponent.html#setStyle(java.lang.String)
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlBasedComponent.html#setSclass(java.lang.String)
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/HtmlBasedComponent.html#setZclass(java.lang.String)
[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Window.html#setContentStyle(java.lang.String)
[7] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Window.html#setContentSclass(java.lang.String)

Theme Customization

Here we discuss how to customize the standard themes, such as breeze, classic blue and silver gray.

Change Font Size and Family

The default theme of ZK components uses the library properties to control the font size and family. You can change them easily by specifying different values.

Notice that the library properties control the theme for the whole application. If you want to provide *per-user* theme (like zkdemo does), you have to implement a theme provider.

Font Size

The default theme uses the following library properties to control the font sizes.

Name	Default	Description
org.zkoss.zul.theme.fontSizeM	12px	The default font size. It is used in the most components.
org.zkoss.zul.theme.fontSizeS	11px	The smaller font size used in the component that requires small fonts, such as `toolbar`.
org.zkoss.zul.theme.fontSizeXS	10px	The extremely small font size; rarely used.
org.zkoss.zul.theme.fontSizeMS	11px	The font size used in the menu items.

To change the default value, you can specify the library properties in `WEB-INF/zk.xml` as follows.

```
<library-property>
	<name>org.zkoss.zul.theme.fontSizeM</name>
	<value>12px</value>
</library-property>
<library-property>
	<name>org.zkoss.zul.theme.fontSizeS</name>
	<value>10px</value>
</library-property>
<library-property>
	<name>org.zkoss.zul.theme.fontSizeXS</name>
	<value>9px</value>
</library-property>
```

Font Family

The following library properties control the font family.

Name	Description
org.zkoss.zul.theme.fontFamilyT	Default: `Verdana, Tahoma, Arial, Helvetica, sans-serif` The font family used for titles and captions.
org.zkoss.zul.theme.fontFamilyC	Default: `Verdana, Tahoma, Arial, serif` The font family used for conntents.

Add Additional CSS

If you want to customize certain components, you can provide a CSS file to override the default setting. For example, if you want to customize the look and feel of the `a` component, you can provide a CSS file with the following content.

```
.z-a-disd {
      color: #C5CACB !important; cursor: default!important;
      text-decoration: none !important;
}
.z-a-disd:visited, .z-a-disd:hover {
      text-decoration: none !important; cursor: default !important;;
      border-color: #D0DEF0 !important;
}
```

Then, specify it in `WEB-INF/zk.xml` as follows.

```
 <desktop-config>
     <theme-uri>/css/my.css</theme-uri>
 </desktop-config>
```

For more information, please refer to the ZK Style Guide.

Version History

Version	Date	Content

Theme Providers

A theme provider (ThemeProvider [1]) allows you the full control of CSS styling, including but not limited to

- Allow you to switch among multiple themes based on, say, the user's preference, cookie, locale or others
- Replace the CSS styling of particular component(s) with your own customization
- Add additional CSS files

We will illustrate the theme provider with two examples. One is straightforward: set the corresponding attributes based on the cookie. The other **injects** a fragment to the URI such that we can allow the browser to cache the CSS file.

For information of 3.6 and earlier, please refer to ZK 3 Theme Provider.

Examples

A Simple Example

In the following example, we store the preferred font size and the skin (theme) in the cookie and retrieve them when required.

```
package my;
public class MyThemeProvider implements ThemeProvder {
	public Collection getThemeURIs(Execution exec, List uris) {
		if ("silvergray".equals(getSkinCookie(exec))) {
			uris.add("~./silvergray/color.css.dsp");
			uris.add("~./silvergray/img.css.dsp");
		}
		return uris;
	}
	public int getWCSCacheControl(Execution exec, String uri) {
		return -1;
	}
	public String beforeWCS(Execution exec, String uri) {
		final String fsc = getFontSizeCookie(exec);
		if ("lg".equals(fsc)) {
			exec.setAttribute("fontSizeM", "15px");
			exec.setAttribute("fontSizeMS", "13px");
			exec.setAttribute("fontSizeS", "13px");
			exec.setAttribute("fontSizeXS", "12px");
		} else if ("sm".equals(fsc)) {
			exec.setAttribute("fontSizeM", "10px");
			exec.setAttribute("fontSizeMS", "9px");
			exec.setAttribute("fontSizeS", "9px");
			exec.setAttribute("fontSizeXS", "8px");
		}
		return uri;
	}
	public String beforeWidgetCSS(Execution exec, String uri) {
```

```
            return uri;
      }
      /** Returns the font size specified in cooke. */
      private static String getFontSizeCookie(Execution exec) {
            Cookie[] cookies =
((HttpServletRequest)exec.getNativeRequest()).getCookies();
            if (cookies!=null)
                  for (int i=0; i<cookies.length; i++)
                        if ("myfontsize".equals(cookies[i].getName()))
                              return cookies[i].getValue();
            return "";
      }
      /** Returns the skin specified in cookie. */
      private static String getSkinCookie(Execution exec) {
            Cookie[] cookies =
((HttpServletRequest)exec.getNativeRequest()).getCookies();
            if (cookies!=null)
                  for (int i=0; i<cookies.length; i++)
                        if ("myskin".equals(cookies[i].getName()))
                              return cookies[i].getValue();
            return "";
      }
}
```

Notice that we return -1 when java.lang.String) ThemeProvider.getWCSCacheControl(org.zkoss.zk.ui.Execution, java.lang.String) [2] is called to disallow the browser to cache the CSS file. It is necessary since we manipulate the content of the CSS file by setting the attributes (based on the cookie). It means the content might be different with the same request URL. For a cacheable example, please refer to the next section.

Then, you configure `WEB-INF/zk.xml` by adding the following lines.

```
<desktop-config>
    <theme-provider-class>my.MyThemeProvider</theme-provider-class>
</desktop-config>
```

A Cacheable Example

To improve the performance, it is better to allow the browser to cache the CSS file as often as possible. With the theme provider, it can be done by returning a positive number when java.lang.String) ThemeProvider.getWCSCacheControl(org.zkoss.zk.ui.Execution, java.lang.String) [2] is called. However, because the browser will use the cached version, we have to ensure the browser gets a different URL for each different theme. Here we illustrate a technique called **fragment injection**.

The idea is simple: when java.util.List) ThemeProvider.getThemeURIs(org.zkoss.zk.ui.Execution, java.util.List) [3] is called, we **inject** a special fragment to denote the content, such that each different theme is represented with a different URL. The injection can be done easily with the inner class called Aide [4]. For example,

```
final String fsc = getFontSizeCookie(exec);
if (fsc != null && fsc.length() > 0) {
      for (ListIterator it = uris.listIterator(); it.hasNext();) {
            final String uri = (String)it.next();
```

```
            if (uri.startsWith(DEFAULT_WCS)) {
                  it.set(Aide.injectURI(uri, fsc));
                  break;
            }
      }
}
```

Then, we can retrieve the fragment we encoded into the URI later when java.lang.String) ThemeProvider.beforeWCS(org.zkoss.zk.ui.Execution, java.lang.String) [5] is called. It can be done easily by use of Aide.decodeURI(java.lang.String) [6]. Aide.decodeURI(java.lang.String) [6] returns a two-element array if the fragment is found. The first element is the URI without fragment, and the second element is the fragment. For example,

```
public String beforeWCS(Execution exec, String uri) {
      final String[] dec = Aide.decodeURI(uri);
      if (dec != null) {
            if ("lg".equals(dec[1])) {
                  exec.setAttribute("fontSizeM", "15px");
                  exec.setAttribute("fontSizeMS", "13px");
                  exec.setAttribute("fontSizeS", "13px");
                  exec.setAttribute("fontSizeXS", "12px");
            } else if ("sm".equals(dec[1])) {
                  exec.setAttribute("fontSizeM", "10px");
                  exec.setAttribute("fontSizeMS", "9px");
                  exec.setAttribute("fontSizeS", "9px");
                  exec.setAttribute("fontSizeXS", "8px");
            }
            return dec[0];
      }
      return uri;
}
```

Here is a complete example:

```
public class CacheableThemeProvider implements ThemeProvider{
      private static String DEFAULT_WCS = "~./zul/css/zk.wcs";

      public Collection getThemeURIs(Execution exec, List uris) {
            //font-size
            final String fsc = getFontSizeCookie(exec);
            if (fsc != null && fsc.length() > 0) {
                  for (ListIterator it = uris.listIterator();
it.hasNext();) {
                        final String uri = (String)it.next();
                        if (uri.startsWith(DEFAULT_WCS)) {
                              it.set(Aide.injectURI(uri, fsc));
                              break;
                        }
                  }
```

```
            }

            //slivergray
            if ("silvergray".equals(getSkinCookie(exec))) {
                  uris.add("~./silvergray/color.css.dsp");
                  uris.add("~./silvergray/img.css.dsp");
            }
            return uris;
      }

      public int getWCSCacheControl(Execution exec, String uri) {
            return 8760; //safe to cache
      }
      public String beforeWCS(Execution exec, String uri) {
            final String[] dec = Aide.decodeURI(uri);
            if (dec != null) {
                  if ("lg".equals(dec[1])) {
                        exec.setAttribute("fontSizeM", "15px");
                        exec.setAttribute("fontSizeMS", "13px");
                        exec.setAttribute("fontSizeS", "13px");
                        exec.setAttribute("fontSizeXS", "12px");
                  } else if ("sm".equals(dec[1])) {
                        exec.setAttribute("fontSizeM", "10px");
                        exec.setAttribute("fontSizeMS", "9px");
                        exec.setAttribute("fontSizeS", "9px");
                        exec.setAttribute("fontSizeXS", "8px");
                  }
                  return dec[0];
            }
            return uri;
      }

      public String beforeWidgetCSS(Execution exec, String uri) {
            return uri;
      }

      /** Returns the font size specified in cooke. */
      private static String getFontSizeCookie(Execution exec) {
            Cookie[] cookies =
((HttpServletRequest)exec.getNativeRequest()).getCookies();
            if (cookies!=null)
                  for (int i=0; i<cookies.length; i++)
                        if ("myfontsize".equals(cookies[i].getName()))
                              return cookies[i].getValue();
            return "";
      }
      /** Returns the skin specified in cookie. */
```

```
      private static String getSkinCookie(Execution exec) {
            Cookie[] cookies =
((HttpServletRequest)exec.getNativeRequest()).getCookies();
            if (cookies!=null)
                  for (int i=0; i<cookies.length; i++)
                        if ("myskin".equals(cookies[i].getName()))
                              return cookies[i].getValue();
            return "";
      }
}
```

How to Specify the Media Types

```
[since 5.0.3]
```

In addition to String instances, you can return instances of StyleSheet [7] in the returned collection of ThemeProvider.getThemeURIs(org.zkoss.zk.ui.Execution,java.util.List) [8], such that you can control more about the generated CSS link. For example, if you want to add a CSS link for the media type [9], say, print, handheld, then you can do as follows.

```
public Collection getThemeURIs(Execution exec, List uris) {
   uris.add(new StyleSheet("/theme/foo.css", "text/css", "print,
handheld", false));
   return uris;
}
```

Version History

Version	Date	Content
5.0.3	June 2010	The media type was allowed in StyleSheet [7].

References

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

[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ThemeProvider.html#getWCSCacheControl(org.zkoss.zk.ui.Execution,

[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ThemeProvider.html#getThemeURIs(org.zkoss.zk.ui.Execution,

[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ThemeProvider/Aide.html#

[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ThemeProvider.html#beforeWCS(org.zkoss.zk.ui.Execution,

[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ThemeProvider/Aide.html#decodeURI(java.lang.String)

[7] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/web/servlet/StyleSheet.html#

[8] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ThemeProvider.html#getThemeURIs(org.zkoss.zk.ui.Execution,java.util.List)

[9] http://www.w3.org/TR/CSS2/media.html

Internationalization

This chapter describes how to make ZK applications flexible enough to run in any locale.

First of all, ZK enables developers to embed Java code and EL expressions any way you like. You could use any Internationalization method you want, such as java.util.ResourceBundle.

However, ZK has some built-in support of internationalization that you might find them useful.

Locale

Overview

The locale used to process requests and events is, by default, determined by the browser's preferences (by use of the `getLocale` method of `javax.servlet.ServletRequest`). For example, `DE` is assumed if an user is using a DE-version browser (unless he changed the setting).

In this section, we'd like to discuss how to configure ZK to handle the locale differently. For example, you could configure ZK to use the same Locale for all users no matter how the browser is configured. Another example is that you could configure ZK to use the preferred locale that a user specified in his or her profile, if you maintain the user profiles in the server.

The Decision Sequence of Locale

The locale is decided in the following sequence.

1. It checks if an attribute called org.zkoss.web.preferred.locale defined in the HTTP session (or Session [2]). If so, use it.
2. It checks if an attribute called org.zkoss.web.preferred.locale defined in the Servlet context (or Application [1]). If so, use it.
3. It checks if a property called org.zkoss.web.preferred.locale defined in the library property (i.e., Library [2]). If so, use it.
4. If none of them is found, it uses the locale defined in the Servlet request (i.e., ServletRequest.getLocale()).

With this sequence in mind, you could configure ZK to use the correct locale based on the application requirements.

Application-level Locale

If you want to use the same locale for all users, you can specify the locale in the library property. For example, you could specify the following in `WEB-INF/zk.xml`:

```
<library-property>
    <name>org.zkoss.web.preferred.locale</name>
    <value>de</value>
</library-property>
```

Alternatively, if you prefer to specify it in Java, you could invoke java.lang.String) Library.setProperty(java.lang.String, java.lang.String) [3]. Furthermore, to avoid typo, you could use Attributes.PREFERRED_LOCALE [4] as follows.

```
Library.setProperty(Attributes.PREFERRED_LOCALE, "de");
```

Per-user Locale

Because ZK will check if a session attribute for the default locale, you could configure ZK to have per-user locale by specifying the attribute in a session.

For example, you can do this when a user logins.

```
import org.zkoss.web.Attributes;
...

 void login(String username, String password) {
     //check password
     ...
     Locale preferredLocale = ...; //decide the locale (from, say,
database)
     session.setAttribute(Attributes.PREFERRED_LOCALE,
preferredLocale);
     ...
 }
```

The Request Interceptor

Deciding the locale after the user logins may be a bit late for some applications. For example, you might want to use the same Locale that was used in the previous session, before the user logins. For a Web application, it is usually done by storing the information in a cookie. It can be done by registering a request interceptor, and then manipulating the cookies when the interceptor is called.

A request interceptor is used to intercept each request being processed. It must implement the RequestInterceptor [5] interface. For example,

```
import java.util.Locale;

import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;

import org.zkoss.web.Attributes;

public class MyLocaleProvider implements
org.zkoss.zk.ui.util.RequestInterceptor {
     public void request(org.zkoss.zk.ui.Session sess,
     Object request, Object response) {
        final Cookie[] cookies =
((HttpServletRequest)request).getCookies();
         if (cookies != null) {
             for (int j = cookies.length; --j >= 0;) {
                if (cookies[j].getName().equals("my.locale")) {
                     //determine the locale
                     String val = cookies[j].getValue();
                     Locale locale =
org.zkoss.util.Locales.getLocale(val);
                     sess.setAttribute(Attributes.PREFERRED_LOCALE,
```

```
locale);
                        return;
                    }
                }
            }
        }
}
```

To make it effective, you have to register it in WEB-INF/zk.xml as a listener. Once registered, the request method is called each time ZK receives a request.

```
<listener>
    <listener-class>MyLocaleProvider</listener-class>
</listener>
```

Note: An instance of the interceptor is instantiated when it is registered. It is then shared among all requests in the same application. Thus, you have to make sure it can be accessed concurrently (i.e., thread-safe).

Note: The `request` method is called at very early stage, before the request parameters are parsed. Thus, it is recommended to access them in this method, unless you configured the locale and character encoding properly for the request.

Change Locale at Run-time

When changing the locale dynamically at the run-time (i.e., under an AU request), it is important to notice:

1. The Locale-dependent messages have been sent to the client, and they have to be reloaded.
2. The current thread's default locale has to be changed since Locale-dependent components and functionality depend on it.

Reload with sendRedirect

The simplest way to solve the issues is to ask the browser to reload the whole page by use of Executions.sendRedirect(java.lang.String) [1].

For example,

```
session.setAttribute(Attributes.PREFERRED_LOCALE, locale);
Executions.sendRedirect(null); //reload the same page
```

Notice that Executions.sendRedirect(java.lang.String) [1] will cause the client to reload the page, so any updates to the current desktop will be lost.

Change without Reloading

If you prefer to keep the current desktop, you have to ask the browser to reload the messages, and change the default locale used by the current thread if you're going to access any component and functionality that depends on. The reloading of messages can be done by invoking Clients.reloadMessages(java.util.Locale) [6], while the setting of the default locale can be done by the use of Locales.setThreadLocal(java.util.Locale) [7]

For example,

```
session.setAttribute(Attributes.PREFERRED_LOCALE, locale);
Clients.reloadMessage(locale);
Locales.setThreadLocale(locale);
```

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Application.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/lang/Library.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/lang/Library.html#setProperty(java.lang.String,
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/web/Attributes.html#PREFERRED_LOCALE
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/RequestInterceptor.html#
[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Clients.html#reloadMessages(java.util.Locale)
[7] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/Locales.html#setThreadLocal(java.util.Locale)

Time Zone

Overview

The time zone used to process requests and events is, by default, determined by the JVM's default (by use of the `getDefault` method of `java.util.TimeZone`)[1].

In this section, we will discuss how to configure ZK to use a time zone other than JVM's default. For example, you might configure ZK to use the preferred time zone that a user specified in his or her profile.

[1] Unlike locale, there is no way to determine the time zone for each browser

The Decision Sequence of Time Zone

The time zone is decided in the following sequence.

1. It checks if an attribute called org.zkoss.web.preferred.timeZone defined in the HTTP session (aka., Session (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Session.html#)). If so, use it.
2. It checks if an attribute called org.zkoss.web.preferred.timeZone defined in the Servlet context (aka., Application (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Application.html#)). If so, use it.
3. It checks if a property called org.zkoss.web.preferred.timeZone defined in the library property (i.e., Library (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/lang/Library.html#)). If so, use it.
4. If none of them is found, JVM's default will be used.

With this sequence in mind, you could configure ZK to use the correct time zone based on the application requirements.

Application-level Time Zone

If you want to use the same time zone for all users of the same application, you can specify the time zone in the library property. For example, you could specify the following in `WEB-INF/zk.xml`:

```
<library-property>
    <name>org.zkoss.web.preferred.timeZone</name>
    <value>GMT-8</value>
</library-property>
```

where the value can be anything accepted by the getTimeZone method of java.util.TimeZone

Alternatively, if you prefer to specify it in Java, you could invoke java.lang.String) Library.setProperty(java.lang.String, java.lang.String) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/lang/Library. html#setProperty(java. lang. String,). Furthermore, to avoid typo, you could use Attributes.PREFERRED_TIME_ZONE (http:// www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ web/ Attributes. html#PREFERRED_TIME_ZONE) as follows.

```
Library.setProperty(Attributes.PREFERRED_TIME_ZONE, "PST");
```

Per-user Time Zone

Because ZK will check if a session attribute for the default time zone, you could configure ZK to have per-user time zone by specifying the attribute in a session.

For example, you can do this when a user logins.

```
 void login(String username, String password) {
     //check password
     ...
     TimeZone preferredTimeZone = ...; //decide the time zone (from,
say, database)
     session.setAttribute(Attributes.PREFERRED_TIME_ZONE,
preferredTimeZone);
     ...
 }
```

The Request Interceptor

Like configuring locale, you can prepare the time zone for the given session by the use of the request interceptor. Please refer to the Locale section for more information.

Version History

Version	Date	Content

Labels

Overview

For a multilingual application, it is common to display the content in the language that the end user prefers. Here we discuss the built-in support called *internationalization labels*.

However, if you prefer to use other approach, please refer to the Use Other Implementation section.

Internationalization Labels

The internationalization labels of an application are loaded from properties files based on the current locale[1] . A properties file is a simple text file encoded in UTF-8[2] . The file contains a list of `key=value` pairs, such as[3]

```
# This is the default LabelsBundle.properties file
s1=computer
s2=disk
s3=monitor
s4=keyboard
```

By default the property file must be placed under the `WEB-INF` directory and named as `zk-label_`*`lang_CNTY`*`.properties`.[4] , where *lang* is the language such as en and fr, and *CNTY* is the country, such as US and FR.

If you want to use one file to represent a language regardless the country, you could name it zk-label_*lang*.properties, such as `zk-label_ja.properties`. Furthermore, zk-label.properties is the default file if the user's preferred locale doesn't match any other file.

When an user accesses a page, ZK will load the properties files for the user's locale. For example, assume the locale is `de_DE`, then it will search the following files and load them if found:

1. zk-label_de_DE.properties
2. zk-label_de.properties
3. zk-label.properties

By default, one properties file is used to contain all labels of a given locale. If you prefer to split it to multiple properties files (such as one file per module), please refer to the Loading Labels from Multiple Resources section.

Also notice that all files match the given locale will be loaded and merged, and the property specified in, say, `zk-label_de_DE.properties` will override what are defined in `zk-label_de.properties` if replicated. It also means if a label is the same in both `de_DE` and `de`, then you need only to specify in `zk-label_de.properties` (and then it will be *inherited* when de_DE is used). Of course, you could specify it in both files.

[1] It is the value returned by Locales.getCurrent() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/Locales.html#getCurrent()). For more information, please refer to the Locale section.
[2] If you prefer a different charset, please refer to the Encoding Character Set section.
[3] Please refer to here for more details about the format of a properties file, such as the use of multiple lines and EL expressions.
[4] Notice the directory and filename is configurable. For more information, please refer ZK Configuration Reference: org.zkoss.util.label.web.location

Access Internationalization Labels In ZUML

Use `labels`

Since 5.0.7 and later, an implicit object called labels was introduced, such that you could access the internationalization labels (so-called internationalization labels) directly. For example, assume you have a label called `app.title`, and then you could:

```
<window title="${labels.app.title}">
...
</window>
```

The labels object is a map (`java.util.Map`), so you could access the label directly by the use of `labels.`*`whatever`* in an EL expression. Moreover, as shown above, you could access the label even if a key is named as *aa.bb.cc* (a string containing dot), such as app.title in the above example.

If the key is not a legal name, you could use `labels['key']` to access, such as `labels['foo-yet']`.

When an internationalization label is about to be retrieved, one of zk-label_*lang_CNTY*.properties will be loaded. For example, if the Locale is `de_DE`, then `WEB-INF/zk-label_de_DE.properties` will be loaded. If no such file, ZK will try to load `WEB-INF/zk-label_de.properties` and `WEB-INF/zk-label.properties` in turn.

Notice that ZK groups the segmented labels as map. For example, ${labels.app} was resolved as a map containing two entries (title and `description`).

```
app.title=Foo
app.description=A super application
```

If you have a key named as the prefix of the other keys, you have to use `$` to access it. For example, if the labels consist of keys `a`, `a.b`, etc., `${labels.a.$}` is required to resolve the label with key named `a`.

For example, in properties file:

```
app=Application
app.title=Foo
app.description=A super application
```

In ZUL:

```
<window title="${labels.app.$}"><!-- shows "Application" -->
...
</window>
<window title="${labels.app}"><!-- WRONG! -->
...
</window>
```

Use `c:l('key')`

With 5.0.6 or prior, you could, to get an internationalization label, use `${c:l('key')}` in EL expression. For example,

```
<?taglib uri="http://www.zkoss.org/dsp/web/core" prefix="c"?>

<window title="${c:l('app.title')}">
 ...
</window>
```

Notice that the l function belongs to the TLD file called http://www.zkoss.org/dsp/web/core, so we have to specify it with the taglib directive as shown above.

Use `c:l2('key')` to format the message

If you'd like to use the label as a pattern to generate concatenated message with additional arguments (like java.text.MessageFormat (http://download.oracle.com/javase/6/docs/api/java/text/MessageFormat.html) does), you could use the l2 function

For example, let us assume we want to generate a full name based on the current Locale, then we could use `${c:l2('key',args)}` to generate concatenated messages as follows.

```
<?taglib uri="http://www.zkoss.org/dsp/web/core" prefix="c"?>
<label value="${c:l2('fullname.format', fullname)}">
```

where we assume fullname is a string array (such as new String[] {"Jimmy", "Shiau"}).

java.lang.Object[] (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/resource/Labels.html#getLabel(java.lang.String,)) Labels.getLabel(java.lang.String, java.lang.Object[])] assumes the content is a valid pattern accepted by MessageFormat (http://download.oracle.com/javase/6/docs/api/java/text/MessageFormat.html), such as "{1}, {0}".

Access Internationalization Labels In Java

To access labels in Java code (including zscript), you could use Labels.getLabel(java.lang.String) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/resource/Labels.html#getLabel(java.lang.String)), java.lang.Object[] (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/resource/Labels.html#getLabel(java.lang.String,)) Labels.getLabel(java.lang.String, java.lang.Object[])] and others.

```
String username = Labels.getLabel("username");
```

Here is a more complex example. Let us assume we want to generate a full name based on the Locale, then we could use java.lang.Object[] (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/resource/Labels.html#getLabel(java.lang.String,)) Labels.getLabel(java.lang.String, java.lang.Object[])] to generate concatenated messages as follows.

```
public String getFullName(String firstName, String lastName) {
   return Labels.getLabel("fullname.format", new java.lang.Object[]
{firstName, lastName});
}
```

java.lang.Object[] (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/resource/Labels.html#getLabel(java.lang.String,)) Labels.getLabel(java.lang.String, java.lang.Object[])] assumes the content is a valid pattern accepted by MessageFormat (http://download.oracle.com/javase/6/docs/api/java/text/

MessageFormat.html), such as "{1}, {0}".

Encoding character set

By default, the encoding of properties files are assumed to be UTF-8. If you prefer another encoding, please specify it in a library property called org.zkoss.util.label.web.charset. It also means all properties files must be encoded in the same character set.

For more information, please refer to ZK Configuration Reference.

Loading Labels from Multiple Resources

It is typical to partition the properties file into several modules for easy maintenance. Since 5.0.7 and later, you could specify the location for each of these properties file with the label-location element. For example,

```
<system-config>
    <label-location>/WEB-INF/labels/order.properties</label-location>
    <label-location>/WEB-INF/labels/invoice.properties</label-location>
</system-config>
```

Notice that, once you specify `label-location`, the default loading of `/WEB-INF/zk-labels.properties` won't take place. In other words, only the properties files specified in the label-location elements are loaded. Thus, if you'd like to load `/WEB-INF/zk-labels.properties` too, you have to add it to `label-location` with others.

Also notice that you don't have to and shall not specify the language, such as `de_DE`, in the path. ZK will try to locate the most matched one as described in the previous section.

In addition to the servlet path, you could specify a file path by starting with `file://`[1] . For example, `file:///foo/labels.properties`. If the target environment is Windows, you could specify the drive too, such as `file:///C:/myapp/foo.properties`. The advantage is that additional properties files could be added after the project has been built into a WAR file.

```
<system-config>
    <label-location>file:///labels/order.properties</label-location>
    <label-location>file:///labels/invoice.properties</label-location>
</system-config>
```

Notice that the configuration with a path related to the file system is better not to be part of `WEB-INF/zk.xml`, since it is easy to cause errors when deploying the application. Rather, it is better to be specified in the additional configuration file. The additional configuration file is also specified at the run time and could be located in the file system (rather than the WAR file). It can be done by specifying the path of the configuration file in a library property called org.zkoss.zk.config.path.

For 5.0.6 and older, you could use the approach described in the following section to load multiple properties files.

---

[1] For more information about the URI of a file, please refer to File URI scheme (http://en.wikipedia.org/wiki/File_URI_scheme).

Loading from Database or Other Resources

If you prefer to put the internationalization labels in, say, database, you could extend the label loader to load labels from other locations, say database. It can be done by registering a locator, which must implement either LabelLocator (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ util/ resource/ LabelLocator. html#) or LabelLocator2 (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/resource/LabelLocator2.html#). Then,

invoke Labels.register(org.zkoss.util.resource.LabelLocator) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ util/ resource/ Labels. html#register(org. zkoss. util. resource. LabelLocator)) or Labels.register(org.zkoss.util.resource.LabelLocator2) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/resource/Labels.html#register(org.zkoss.util.resource.LabelLocator2)) to register it[1] .

If you can represent your resource in URL, you could use LabelLocator (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/resource/LabelLocator.html#) (as show below). If you have to load it by yourself, you could use LabelLocator2 (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/resource/LabelLocator2.html#) and return an input stream (java.io.InputStream).

Alernative 1: load as an input stream:

```
public class FooDBLocator extends org.zkoss.util.resource.LabelLocator2
 {
    private String _field;
    public FooDBLocator(String field) {
        _field = field;
    }
    public InputStream locate(Locale locale) {
        InputStream is = ... //load the properties from, say, database
        return is;
    }
    public String getCharset() {
        return "UTF-8"; //depending the encoding you use
    }
}
```

Alernative 2: load as an URL:

```
public class FooServletLocator extends
org.zkoss.util.resource.LabelLocator {
    private ServletContext _svlctx;
    private String _name;
    public FooServletLocator(SevletContext svlctx, String name) {
        _svlctx = svlctx;
        _name = name;
    }
    public URL locate(Locale locale) {
        return _svlctx.getResource("/WEB-INF/labels/" + name + "_" +
locale + ".properties");
    }
}
```

Then, we could register label locators when the application starts by use of WebAppInit (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/WebAppInit.html#) as follows.

```
public class MyAppInit implements org.zkoss.zk.ui.util.WebAppInit {
    public void init(WebApp wapp) throws Exception {
        Labels.register(new FooDBLocator(("moduleX");
        Labels.register(new FooDBLocator(("moduleY");
        Labels.register(new
```

```
FooServletLocator((ServletContext)wapp.getNativeContext(), "module-1");
        Labels.register(new
FooServletLocator((ServletContext)wapp.getNativeContext(), "module-2");
    }
}
```

where we assume `moduleX` and `moduleY` is the database table to load the properties, and `module-1.properties` and `module-2.properties` are two modules of messages you provide. Then, you configure it in `WEB-INF/zk.xml` as described in ZK Configuration Reference.

---

[1] For 5.0.7 and later, you could use the label-location element if the properties file is located in the file system or in the Web application as described in the previous section.

Reload Labels Dynamically

The internationalization labels are loaded when a locale is used at the first time. It won't be reloaded automatically if the file is modified. However, it is easy to force ZK to reload by the use of Labels.reset() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/resource/Labels.html#reset()).

For example, you could prepare a test paging for reloading as follows.

```
<zk>
<button label="Reload Labels" onClick="org.zkoss.util.resource.Labels.reset();execution.s
Test result: ${foo} ${another.whatever}
</zk>
```

Use Other Implementation

If you prefer to use other implementation (such as property bundle), you could implement a static method and map it with xel-method. Then, you could reference it in EL expressions. For example,

```
<?xel-method prefix="c" name="label" class="foo.MyI18Ns"
  signature="java.lang.String label(java.lang.String)"?>
<window title="${c:label('app.title')}">
....
${c:label('another.key')}
</window>
```

Version History

Version	Date	Content
5.0.5	October 2010	LabelLocator2 (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/resource/LabelLocator2.html#) was introduced.
5.0.7	March 2011	The `labels` object was introduced.

The Format of Properties Files

In this section, we will discuss the format of a properties file, such as `zk-label.properties`.

A properties file is a simple text file. The file contains a list of `key=value` pairs, such as

```
# This is the default LabelsBundle.properties file
s1=computer
s2=disk
s3=monitor
s4=keyboard
```

The default encoding of a properties file is assumed to be UTF-8. If you want to use a different encoding, please refer to the Use Encoding Other Than UTF-8 section.

A properties file is usually used to contain the internationalization labels of an application, but technically you could use it in any situation you'd like[1] .

[1] If it is used for internationalization labels, it will be loaded automatically. If you want to use it in other situation, you have to invoke java.io.InputStream, boolean) Maps.load(java.util.Map, java.io.InputStream, boolean) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/Maps.html#load(java.util.Map,) or similar to load it manually.

Specify a Value with Multiple Lines

By default, a property is the text specified right after the equal sign. If the property's value has multiple lines, you could use the following format:

```
key={
line 1
line 2
}
```

Notice that the curly braces must be followed by a line break immediately, and the right brace (}) must be the only character in the line.

Specify Segmented Keys

Since all internationalization labels are stored in the same scope, it is common to separate them by naming the key with dot (.) like the Java package name. For sake of description, we call them segmented key. For example,

```
order.fruit.name = Orange
order.fruit.description = A common fruit
```

It can be simplified by use of the following syntax:

```
order.fruit.  {
name = Orange
description = A common fruit
}
```

As shown, the segmented key could be specified by specifying the prefix and a following right brace ({).

The segmented key could be accessed in two ways.

First, with an implicit object called labels:

```
<textbox value="${labels.order.fruit.name}"/>
```

> **Under the hood:** The `labels` object is actually the map returned by Labels.getSegmentedLabels() (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ util/ resource/ Labels. html#getSegmentedLabels()). Furthermore, if the key of a property contains dot (.), i.e., segmented, all properties with the same prefix are grouped as another map. For example, `${labels.order}` (i.e., Labels.getSegmentedLables().get("order")) will return a map containing an entry (`fruit`) in the above example.

Second, with an EL function called [[ZUML Reference/EL Expressions/Core Methods/lll] and/or l2:

```
<?taglib uri="http://www.zkoss.org/dsp/web/core" prefix="c"?>
<label value="${c:l('order.fruit.name')}">
```

Specify a Comment

You could put a comment line by starting with the sharp sign (#), such as

```
#This is a comment line that will be ignored when loaded
```

Use EL Expressions

EL expressions are allowed for a property's value. For example, you could reference a property's value in another property, such as

```
first=the first label
second=come after ${first}
```

Segmented keys are also allowed[1] :

```
group1.first=the first group
group2.second=come after ${group1.first}
```

In addition to referencing another property, you could reference any implicit object specified in ZUML Reference: Implicit Objects if it is part of a HTTP request (excluding component/page).

For example, param references to a request's parameter:

```
message=Thank ${param.user} for using
```

[1] The segmented key was supported since 5.0.7

Use Encoding Other Than UTF-8

By default, the encoding of properties files are assumed to be UTF-8. If you prefer another encoding, please specify it in a library property called org.zkoss.util.label.web.charset. It also means all properties files must be encoded in the same character set.

For more information, please refer to ZK Configuration Reference.

Version History

Version	Date	Content
5.0.7	Mar 2011	labels implicit object was introduced to access properties without declaring taglib. Also allows label keys of a.b.c format.

Date and Time Formatting

Overview

By default, the format of date and time, especially the format of Datebox and Timebox, is determined by the JVM's default and the current locale.

In this section, we will discuss how to configure ZK to use the format other than the JVM. For example, you could configure ZK to use the preferred format based on the user's preferences.

The Decision Sequence of Format

The format of date and time is decided in the following sequence.

1. It checks if an attribute called `org.zkoss.web.preferred.dateFormatInfo` defined in the HTTP session (i.e., Session [2]). If so, it will be used by assuming the value is an instance or a class of DateFormatInfo [1].
2. It checks if an attribute called `org.zkoss.web.preferred.dateFormatInfo` defined in the servlet context (i.e., Application [1]). If so, it will be used by assuming the value is an instance or a class of DateFormatInfo [1].
3. It checks if a property called `org.zkoss.web.preferred.dateFormatInfo` defined in the library property (i.e., Library [2]). If so, it will be used by assuming the value is a class of DateFormatInfo [1].
4. If none of them is found, it uses the JVM's default based on the current locale

In other words, to configure ZK to use the format other than the JVM's default, you have to:

1. Implements DateFormatInfo [1] to provide the format you want
2. Specify the class or an instance of it in the session's attribute or application's attribute depending on the requirement of your application.

If a class or the class's name is specified, an instance of it is instantiated each time the server receives a request from the client. It means the implementation needs not to be thread safe, but at the cost of instantiation. On the other hand, if an instance is specified as the attribute value, it will be used for all requests, so it has to be thread safe.

Also notice that you could specify `short`, `long` and other *standard* styling in the format property of datebox and timebox, such that the corresponding format of the styling will be used instead of the default, `meidum`. For more information, please refer to the Per-component Format section.

Application-level Format

If you want to use the same format for all users, you could specify your implementation of DateFormatInfo [1] in the library property. For example,

```
<library-property>
    <name>org.zkoss.web.preferred.dateFormatInfo</name>
    <value>foo.MyDateFormatInfo</value>
</library-property>
```

where we assume the implementation is named foo.MyDateFormatInfo.

Per-user Format

If you'd like to configure ZK to allow each user (aka., session) has an independent format, you could store an instance of your implementation of DateFormatInfo [1] in the session's attribute.

For example, you could do this when a user logins.

```
import org.zkoss.web.Attributes;
...

 void login(String username, String password) {
     //check password
     ...
     session.setAttribute(Attributes.PREFERRED_DATE_FORMAT_INFO,
         new foo.MyDateFormatInfo(session));
     ...
 }
```

where we assume the implementation is named foo.MyDateFormatInfo.

Per-component Format

Datebox and Timebox allow a developer to specify any format he prefer for any instance. For example,

```
<datebox format="MM d, yyyy"/>
<timebox format="HH:mm"/>
```

However, it is usually better to design a page that depends on the configuration as described above, rather than specify the format explicitly in each page. It can be done by specifying the styling rather than the real format in the format property (Datebox.setFormat(java.lang.String) [2] and Timebox.setFormat(java.lang.String) [3]). There are totally four different styling: short, medium, long and full (representing the styling defined in java.text.DateFormat, SHORT, MEDIUM, LONG and FULL). For example,

```
<datebox format="short"/>
<datebox format="long"/>
<timebox format="medium"/>
```

Then, the real format will be decided by your implementation of DateFormatInfo [1], if any, or the JVM's default.

In additions, you could specify the date/time format in the syntax of styling_for_date+styling_for_time, such as:

```
<datebox format="long+medium"/>
```

which specifies the date/time format with the long styling for date and the medium styling for time.

Per-component Locale

In additions to the current locale, you could specify the locale for individual instances of datebox and timebox. Then, the real format will depend on the locale and the format you specified. For example,

```
<datebox format="medium" locale="de"/>
<timebox format="long" locale="fr"/>
```

Note: the language of the format and the datebox's calendar is the same as the locale you specified. [Since 5.0.8]

Version History

Version	Date	Content
5.0.7	April 2011	The per-session format of datebox/timebox was introduced. Prior to 5.0.7, the format depends only on locale.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/text/DateFormatInfo.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Datebox.html#setFormat(java.lang.String)
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Timebox.html#setFormat(java.lang.String)

The First Day of the Week

Overview

By default, the first day of the week depends on the locale (e.g., Sunday in US, Monday in France). More precisely, it is the value returned by the `getFirstDayOfWeek` method of the `java.util.Calendar` class.

However, you can configure it different, and it will affect how datebox and calendar components behave.

The decision sequence of the first day of the week

The first day of the week is decided in the following sequence.

1. It checks if an attribute called org.zkoss.web.preferred.firstDayOfWeek defined in the HTTP session (aka., Session [2]). If so, use it.
2. It checks if an attribute called org.zkoss.web.preferred.firstDayOfWeek defined in the Servlet context (aka., Application [1]). If so, use it.
3. It checks if a property called org.zkoss.web.preferred.firstDayOfWeek defined in the library property (i.e., Library [2]). If so, use it.
4. If none of them is found, JVM's default will be used (`java.util.Calendar`).

Application-level first-day-of-the-week

[since 5.0.3]

If you want to use the same first-day-of-the-week for all users of the same application, you can specify it in the library property. The allowed values include 1 (Sunday), 2 (Monday), .. and 7 (Saturday). For example, you could specify the following in WEB-INF/zk.xml:

```
<library-property>
    <name>org.zkoss.web.preferred.firstDayOfWeek</name>
    <value>7</value><!-- Saturday -->
</library-property>
```

Alternatively, if you prefer to specify it in Java, you could invoke java.lang.String) Library.setProperty(java.lang.String, java.lang.String) [3]. Furthermore, to avoid typo, you could use java.lang.Object) WebApp.setAttribute(java.lang.String, java.lang.Object) [1] and Attributes.PREFERRED_FIRST_DAY_OF_WEEK [2] as follows.

```
webApp.setAttribute(org.zkoss.web.Attributes.PREFERRED_FIRST_DAY_OF_WEEK,
 java.util.Calendar.SATURDAY);
```

As shown above, the allowed values of java.lang.Object) WebApp.setAttribute(java.lang.String, java.lang.Object) [1] include Calendar.SUNDAY, Calendar.MONDAY and so on.

Per-user first-day-of-week

[since 5.0.3]

By specify a value to the session attribute called org.zkoss.web.preferred.firstDayOfWeek (i.e., Attributes.PREFERRED_FIRST_DAY_OF_WEEK [2]), you can control the first day of the week for the given session. The allowed values include Calendar.SUNDAY, Calendar.MONDAY and so on.

```
session.setAttribute(org.zkoss.web.Attributes.PREFERRED_FIRST_DAY_OF_WEEK,
 java.util.Calendar.SATURDAY);
  //then, the current session's first day of the week will be Saturday
```

For example, you can do this when a user logins.

```
 void login(String username, String password) {
     //check password
     ...
     int preferredFDOW = ...; //decide the user's preference
     session.setAttribute(Attributes.PREFERRED_FIRST_DAY_OF_WEEK,
preferredFDOW);
...
```

Version History

Version	Date	Content
5.0.3	June 201	The first day of week is configurable

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/WebApp.html#setAttribute(java.lang.String,
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/web/Attributes.html#PREFERRED_FIRST_DAY_OF_WEEK

Locale-Dependent Resources

Overview

Many resources depend on the Locale and, sometimes, the browser. For example, you might need to use a larger font for Chinese characters to have better readability.

Specifying Locale- and browser-dependent URL

ZK can handle this for you automatically, if you specify the URL with "*". The algorithm is as follows.

1. If there is one "*" is specified in an URI such as `/my*.css`, then "*" will be replaced with a proper Locale depending on the preferences of user's browser.For example, user's preferences is `de_DE`, then ZK searches `/my_de_DE.css`, `/my_de.css`, and `/my.css` one-by-one from your Web site, until any of them is found. If none of them is found, `/my.css` is still used.
2. If two or more "*" are specified in an URI such as "/my*/lang*.css", then the first "*" will be replaced with `"ie"` for Internet Explorer, "saf" for Safari, and `"moz"` for other browsers[1] . Moreover, the last asterisk will be replaced with a proper Locale as described in the above step.In summary, the last asterisk represents the Locale, while the first asterisk represents the browser type.
3. All other "*" are ignored.

Note: The lat asterisk that represents the Locale must be placed right before the first dot ("."), or at the end if no dot at all. Furthermore, no following slash (/) is allowed, i.e., it must be part of the filename, rather than a directory. If the last asterisk doesn't fulfill this constraint, it will be eliminated (not ignored).

For example, "/my/lang.css*" is equivalent to "/my/lang.css".

In other words, you can consider it as neutral to the Locale.

Tip: We can apply this rule to specify an URI depending on the browser type, but not depending on the Locale. For example, "/my/lang*.css*" will be replaced with "/my/langie.css" if Internet Explorer is the current user's browser.

---

[1] In the future editions, we will use different codes for browsers other than Internet Explorer, Firefox and Safari.

Example

In the following example, we assume the preferred Locale is `de_DE` and the browser is Internet Explorer.

URI	Resources that are searched
/css/norm*.css	# /norm_de_DE.css 1. /norm_de.css 2. /norm.css
/css-*/norm*.css	# /css-ie/norm_de_DE.css 1. /css-ie/norm_de.css 2. /css-ie/norm.css
/img*/pic*/lang*.png	# /imgie/pic*/lang_de_DE.png 1. /imgie/pic*/lang_de.png 2. /imgie/pic*/lang.png
/img*/lang.gif	# /img/lang.gif
/img/lang*.gif*	# /img/langie.gif
/img*/lang*.gif*	# /imgie/lang*.gif

Locating Locale- and browser-dependent resources in Java

In addition to ZUML[1] , you could handle browser- and Locale-dependent resource in Java. Here are a list of methods that you could use.

- The `encodeURL`, `forward`, and `include` methods in Execution (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Execution.html#) for encoding URL, forwarding to another page and including a page. In most cases, these methods are all you need.
- The `locate`, `forward`, and `include` method in Servlets (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/web/servlet/Servlets.html#) for locating Web resouces. You rarely need them when developing ZK applications, but useful for writing a servlet, portlet or filter.
- The `encodeURL` method in Encodes (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/web/servlet/http/Encodes.html#) for encoding URL. You rarely need them when developing ZK applications, but useful for writing a Servlet, Portlet or Filter.
- The `locate` method in Locators (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/resource/Locators.html#) for locating class resources.

---

[1] It is also supported by all components that accept an URL.

Version History

Version	Date	Content

Warning and Error Messages

Overview

ZK's messages (such as warnings and errors) are packed in property files (*.properties) under the `/metainfo/mesg` directory of the classpath. These mesages are grouped into modules, such as zcommon, zweb, zk and zul. These files are Locale depedent. For example, the message file of `zk.jar` for Germany messages is `msgzk_de_DN.properties` or `msgzk_de.properties`.

Translate messages to another language

If you want to translate messages to another language, you can add your own property files named with the correct Locale, and put it to the `/metainfo/mesg` directory of the classpath. Of course, it is always better to contribute it back. Please take a look at ZK Messages for all available translations. If you'd like to contribute, just add the language to it and notice us at info at zkoss dot org.

Change particular message

```
[since 6.0.0]
```

If you want to change a particular message, you can add it to zk-label.properties. For example, let us see you want to customize MZk.NOT_FOUND in German translation, then you can add the following to WEB-INF/zk-label_de.properties:

```
MZk.3000=my customized message here
```

where 3000 is the error code and you can find it at ZK Messages

Version History

Version	Date	Content
6.0.0	n/a	Allows applications to override a particular message with zk-label.

Server Push

HTTP is a request-and-response protocol. Technically, there is no way to have the server to actively *push* data to the client. However, there are a few approaches [1] to emulate *push* -- it is also called Ajax Push. These approaches could be summarized in two categories, client polling and comet[2] , that are both supported in ZK.

Different approaches have different pros and cons, and we will discuss them in the Configuration section.

No matter which implementation you choose, the use is the same. The Event Queue is the high-level API, and this is a suggested approach for its simplicity. However, if you prefer to access the low-level API directly, you could refer to the Asynchronous Tasks and Synchronous Tasks sections, depending on whether you task can be executed asynchronously.

---

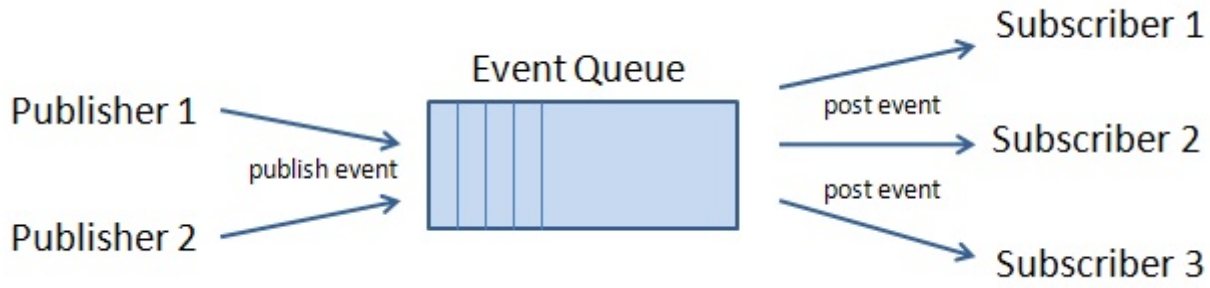
[1] http://en.wikipedia.org/wiki/Push_technology

[2] More precisely, it is so-called long polling.

Event Queues

An event queue is an event-based publish-subscribe solution for the application information delivery and messaging. It provides asynchronous communication for different modules/roles in a loosely-coupled and autonomous fashion.

By publishing, a module (publisher) sends out messages without explicitly specifying or having knowledge of intended recipients. By subscribing, a receiving module (subscriber) receives messages that the subscriber has registered an interest in, without explicitly specifying or knowing the publisher.



ZK generalizes the event queue to support the server push. The use is straightforward: specifying the scope of a given event queue as EventQueues.APPLICATION [2] (or EventQueues.SESSION [1], but rare). For example,

```
EventQueue que = EventQueues.lookup("chat", EventQueues.APPLICATION,
true);
```

For more information about event queues, please refer to the Event Handling: Event Queues section.

For the information about low-evel API, please refer to Asynchronous Tasks section, if the task can execute asynchronously; or Synchronous Tasks if it must execute synchronously.

Version History

Version	Date	Content
5.0.6	November 2010	The event queue won't start any working threads and they are serializable, so it is safe to use them in a clustering environment.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/EventQueues.html#SESSION

Synchronous Tasks

Server push is a technology to actively *push* data to the client. For ZK, the data is usually the UI updates or its variants. Thus, for sake of understanding, we could consider the task is about updating UI in parallel with regular Ajax requests (poll-type request). For example, in a chat application, once a message is entered by a participant, the server has to *push* it to all clients that involve in the conversation.

If the task of updating UI takes place in a working thread, it is generally more convenient to execute it synchronously as described later. On the other hand, if the task can be encapsulated as an event listener (EventListener [1]), you could execute it asynchronously -- please refer to the Asynchronous Tasks section for more information.

Enable Server Push

By default, the server push is disabled (for better performance). Before pushing data for a given desktop, you have to enable the server push for it.

It can be done by use of Desktop.enableServerPush(boolean) [2]:

```
desktop.enableServerPush(true);
```

After the server push of a given desktop is enabled, you could use any number of working thread to update the desktop concurrently as described in the following section[3] .

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

[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Desktop.html#enableServerPush(boolean)

[3] For better performance, it is suggested to disable the server push if it is no longer used in the give desktop.

Update UI in a Working Thread

To updating the UI synchronously in a working thread, we have to do as follows.

1. Invoke Executions.activate(org.zkoss.zk.ui.Desktop) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#activate(org.zkoss.zk.ui.Desktop)). It has two purposes:
   1. It grants the right to access the UI of the given desktop to the caller's thread[1] .
   2. It establishes a connection with the client (the browser window displaying the desktop), such that the update will be sent to the client after finished
2. Update UI any way you want, just like any regular event listener.
3. Invoke Executions.deactivate(org.zkoss.zk.ui.Desktop) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#deactivate(org.zkoss.zk.ui.Desktop)) to return the control, such that other thread could have a chance to update UI.

Here is the pseudo code that illustrates the flow[2] :

```
public class WorkingThread extends Thread {
    public void run() {
        try {
            while (anyDataToShow()) { //whatever you like
                //Step 1. Prepare the data that will be updated to UI
                collectData(); //whatever you like

                //Step 2. Activate to grant the access of the give
desktop
                Executions.activate(desktop);
                try {
                     //Step 3. Update UI
                     updateUI(); //whatever you like
                } finally {
                    //Step 4. Deactivate to return the control of UI
back
                     Executions.deactivate(_desktop);
                }
            }
        } catch (InterruptedException ex) {
            //Interrupted. You might want to handle it
        }
    }
}
```

Notice that the task between Executions.activate(org.zkoss.zk.ui.Desktop) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#activate(org.zkoss.zk.ui.Desktop)) and Executions.deactivate(org.zkoss.zk.ui.Desktop) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#deactivate(org.zkoss.zk.ui.Desktop)) has to be efficient, since it blocks others, including the user (of the desktop), from access the UI. It is suggested to prepare the data before Executions.activate(org.zkoss.zk.ui.Desktop) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#activate(org.zkoss.zk.ui.Desktop)), such that it can be done in parallel with other threads.

---

[1] Notice that, for each desktop, there is at most one thread is allowed to access at the same time.

[2] For a real example, please refer to small talks: Simple and Intuitive Server Push with a Chat Room Example and Server Push with a Stock Chart Example.

Version History

Version	Date	Content

Asynchronous Tasks

If the task of updating UI can be represented as a method, the push can be done easily. All you need to do is

1. Implement the UI updates in an event listener (implementing EventListener [1] or SerializableEventListener [2]).
2. Then, schedule it for executed asynchronously by the use of org.zkoss.zk.ui.event.EventListener, org.zkoss.zk.ui.event.Event) Executions.schedule(org.zkoss.zk.ui.Desktop, org.zkoss.zk.ui.event.EventListener, org.zkoss.zk.ui.event.Event) [1].

Here is the pseudo code:

```
Executions.schedule(desktop,
    new EventListener() {
        public void onEvent(Event event) {
            updateUI(); //whatever you like
        }
    }, event);
```

You could manipulate UI whatever you want in EventListener.onEvent(org.zkoss.zk.ui.Event) [2]. It is no different from any other event listener.

Notice that org.zkoss.zk.ui.event.EventListener, org.zkoss.zk.ui.event.Event) Executions.schedule(org.zkoss.zk.ui.Desktop, org.zkoss.zk.ui.event.EventListener, org.zkoss.zk.ui.event.Event) [1] can be called anywhere, including another event listener or a working thread. In other words, you don't have to fork a working thread to use this feature.

Notice that, since there is at most one thread to access the UI of a given desktop, the event listener's performance must be good. Otherwise, it will block other event listeners from execution. Thus, if you have a long operation to do, you could use event queue's asynchronous event listener, or implement it as a synchronous task and handle lengthy operation outside of the activation block.

Version History

Version	Date	Content
5.0.6	November 2010	This feature was introduced. With 5.0.5 or prior, you have to use Event Queues or Synchronous Tasks.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#schedule(org.zkoss.zk.ui.Desktop,
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/EventListener.html#onEvent(org.zkoss.zk.ui.Event)

Configuration

ZK have two implementations: PollingServerPush [1] and CometServerPush [2]. As their name suggest, they implement the Client-Polling and Comet (aka., long-polling) server pushes.

The default implementation depends on which ZK edition you use. ZK CE and PE will use PollingServerPush [1] ZK EE will use CometServerPush [2]. You configure ZK to use the one you prefer, even to use a custom server push.

Choose an Implementation

Client-polling is based on a timer that peeks the server continuously to see if any data to be *pushed* to the client, while Comet establishes a permanent connection for instant *push*. Client-polling will introduce more traffic due to the continuous peeks, but Comet will consume the network connections that a server allows.

Page-level Configuration

You could configure a particular ZK page to use a particular implementation by the use of DesktopCtrl.enableServerPush(org.zkoss.zk.ui.sys.ServerPush) [2]. For example,

```
((DesktopCtrl)desktop).enableServerPush(
    new org.zkoss.zk.ui.impl.PollingServerPush(2000,5000,-1));
```

Application-level Configuration

If you would like to change the default server push for the whole application, you could use the the server-push-class element as follows.

```
<device-config>
    <device-type>ajax</device-type>
    <server-push-class>org.zkoss.zkex.ui.impl.PollingServerPush</server-push-class>
</device-config>
```

where you could specify any implementation that implements ServerPush [3].

Customized Client-Polling

PollingServerPush [1] uses a timer to peek if the server has any data to *push* back. The period between two peeks is determined by a few factors.

- PollingServerPush.delay.min

  The minimal delay to send the second polling request (unit: milliseconds). Default: 1000.
- PollingServerPush.delay.max

  The maximal delay to send the second polling request (unit: milliseconds). Default: 15000.
- PollingServerPush.delay.factor

  The delay factor. The real delay is the processing time that multiplies the delay factor. For example, if the last request took 1 second to process, then the client polling will be delayed for 1 x factor seconds. Default: 5.

  The larger the factor is, the longer delay it tends to be.

It could be configured in WEB-INF/xml by use of the preference element as follows.

```
<preference>
      <name>PollingServerPush.delay.min</name>
```

```
      <value>3000</value>
</preference>
<preference>
      <name>PollingServerPush.delay.max</name>
      <value>10000</value>
</preference>
<preference>
      <name>PollingServerPush.delay.factor</name>
      <value>5</value>
</preference>
<!-- JavaScript code to start the server push; rarely required
<preference>
      <name>PollingServerPush.start</name>
      <value></value>
</preference>
<preference>
      <name>PollingServerPush.stop</name>
      <value></value>
</preference>
-->
```

In additions, you could specify them in the constructor: int, int) PollingServerPush.PollingServerPush(int, int, int) [4]. For example,

```
((DesktopCtrl)desktop).enableServerPush(
    new org.zkoss.zk.ui.impl.PollingServerPush(2000,10000,3));
```

Error Handling

The configuration of the errors is handled by [[ZK Configuration Reference/zk.xml/The client-config Element/The error-reload Element|the client-reload element], specified in WEB-INF/zk.xml. The markup below demonstrates an example of catching an error of the server push:

```
<error-reload>
    <device-type>ajax</device-type>
    <connection-type>server-push</connection-type>
    <error-code>410</error-code>
    <reload-uri>/login.zul</reload-uri>
</error-reload>
```

where the connection-type element specifies through which channel the requests are sent. By default it is the AU channel in which Ajax requests are sent by widgets running at the client. If you would like to specify the error page for server-push then connection-type must be set to server-push.

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/impl/PollingServerPush.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkmax/ui/comet/CometServerPush.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/sys/ServerPush.html#
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/impl/PollingServerPush.html#PollingServerPush(int,

Clustering

ZK components, pages and desktops are all serializable, so using ZK in a clustering environment is straightforward. However, it is a challenge to develop a sophisticated application that is ready for clustering. If you are not familiar with it, you might refer to J2EE clustering [1] and other resources.

Here we discuss how to configure ZK for a clustering environment, how to configure some servers, and the important notes when developing a clustering-ready application with ZK.

References

[1] http://www.javaworld.com/jw-02-2001/jw-0223-extremescale.html

ZK Configuration

Turn on Serializable UI Factory

```
[Required]
```

To use ZK in a clustering environment, you have to use the serializable UI factory. It could be done by specifying the following statement in `WEB-INF/zk.xml`:

```
<zk>
      <system-config>
            <ui-factory-class>org.zkoss.zk.ui.http.SerializableUiFactory</ui-factory-clas
      </system-config>
</zk>
```

SerializableUiFactory [1] is the UI factory that will instantiate serializable sessions such that the sessions, components, pages and desktops will be serialized when a session is about to deactivate.

Turn on ClusterSessionPath for WebLogic and GAE

```
[Required if WebLogic and GAE][No need for other servers]
[Since 5.0.8]
```

WebLogic clustering server and GAE (Google App Engine) minimize the synchronization of the session states by assuming nothing has changed, if no session attribute has been modified in a HTTP request. Thus, you have to specify the following configuration in `WEB-INF/zk.xml` to enforce the server to synchronize the states for each Ajax request.

```
<zk>
      <listener>
            <listener-class>org.zkoss.zkplus.cluster.ClusterSessionPatch</listener-class>
      </listener>
</zk>
```

Under the hub: ClusterSessionPatch [2] is an ExecutionCleanup [3] listener that updates an session attribute holding Session [2] for each request.

Turn on Log

```
[Optional]
```

If an attribute or a listener is not serializable, ZK will skip it, i.e., not to serialize it (similar to how a Servlet container serializes the attributes of sessions). It is sometimes hard to know what are ignored, since it is common for a developer to forget to declare a value or a listener as serializable.

To detect this problem, you could turn on the logger for `org.zkoss.io.serializable` to the DEBUG level[4] . The logger is the standard logger [5]. You could consult the configuration of the Web server you use. Or, you could run the following statement when your application starts[6] .

```
org.zkoss.util.logging.Log.lookup("org.zkoss.io.serializable").setLevel("DEBUG");
```

---

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/http/SerializableUiFactory.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/cluster/ClusterSessionPatch.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/util/ExecutionCleanup.html#
[4] Available in 5.0.7
[5] http://download.oracle.com/javase/6/docs/api/java/util/logging/Logger.html
[6] It can be done by use of the WebAppInit (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/WebAppInit.html#) listener. For more information, please refer to the Customization section.

Disable the Use of zscript

```
[Optional]
[since 5.0.8]
```

The interpreter (BeanShell) does not work well under the clustering environment, since the serializationis not stable. Thus, it is suggested to disable the use of zscript with the following statement. It is optional, but, by disabling, it is easy to prevent any possible use.

```
<system-config>
      <disable-zscript>true</disable-zscript>
</system-config>
```

Configuration Not Allowed

Here are a list of configuration that can not be used in the clustering environment. They are disabled by default. However, it is worth to double check if any of your members enables it accidentally.

Event Processing Thread

Do not enable the event processing thread. The event processing thread might be suspended, while the (suspended) thread cannot be migrated from one machine to another.

It is disabled by default. For more information, please refer to the Event Threads section.

Global Desktop Cache

Do not use GlobalDesktopCacheProvider (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/impl/GlobalDesktopCacheProvider.html#) (global desktop cache). The global desktop cache is stored in the servlet context, while only the data stored in sessions are migrated when failover takes place.

The desktop desktop cache is *not* used by default[1] . Just make sure you don't configure it wrong.

[1] Rather, the default is SessionDesktopCacheProvider (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/impl/SessionDesktopCacheProvider.html#).

Version History

Version	Date	Content
5.0.7	April 2011	The log called `org.zkoss.io.serializable` was introduced.
5.0.8	June 2011	The listener called `org.zkoss.zkplus.cluster.ClusterSessionPatch` was introduced.

Server Configuration

The configuration of a Web server really depends on the server itself. There is no standard approach.

Apache + Tomcat

For configuring Apache + Tomcat, please refer to

- How to Run ZK on Apache + Tomcat clustering, Part I
- How to Run ZK on Apache + Tomcat clustering, Part II

Google App Engine

For configuring Google App Engine, please refer to

- ZK Installation Guide: Google App Engine

Version History

Version	Date	Content

Programming Tips

Objects Referenced by UI Must be Serializable

Objects that are referenced by an UI object, such as components and pages, have to be serializable. Otherwise, they might have no value after de-serialized, or causes an exception (depending how it is used).

Attributes of UI Objects

If the value of an attribute is not serializable, it will be ignored. Thus, it will become null after de-serialized. So, it is better to make them all serializable (such as implementing java.io.Serializable), or handle the serialization manually (refer to the Clustering Listeners section below) .

zscript

It is OK, though not recommended, to use zscript in a clustering environment, but there are some limitations.

- BeanShell's function is not serializable. For example, the following won't work:

```
void foo() {
}
```

- The value of variables must be serializable

Notice that it is not recommended to use zscript in the clustering environment. After all, the performance of BeanShell is not good.

Event Listeners

Event listeners have to be serializable. Otherwise, it will be ignored after serialization.

A simplest way to make an event listener serializable is to implement SerializableEventListener [2] (available since 5.0.6), instead of EventListener [1].

For example,

```
button.addEventListener(Events.ON_CLICK,
  new SerializableEventListener() {
    public void onEvent(Event event) {
        ....
    }
  });
```

Data Models

The data models, such as ListModel [2] and ChartModel [1], have to be serializable. Otherwise, the UI object (such as grid) won't behave correctly. The implementations provided by ZK are serializable. However, the items to be stored in the data models have to be serializable too.

Composers

If you extend from GenericForwardComposer [5] or GenericAutowireComposer [1], you have to make sure all of its members are serializable (or transient), since the implementation will keep a reference in the applied component.

When implementing from Composer [2] directly, the composer could be non-serializable if you don't keep a reference in any UI object. In other words, the composer will be dropped after Composer.doAfterCompose(org.zkoss.zk.ui.Component) [3]

Clustering Listeners

If there are non-serializable objects, you could implement one of the clustering listeners to handle them manually as described below. Basically, there are two kinds of clustering listeners for different purpose:

- Serialization Listeners: they are called when an object is about to be serialized, and after it has been de-serialized. Example: ComponentSerializationListener [4] and PageSerializationListener [5]
- Activation Listeners: they are called when a session is about to be passivated, and after it has been activated. Examples: ComponentActivationListener [6] and PageActivationListener [7].

To register a listener is straightforward: just implements the corresponding listener interface. For example, you could implement ComponentActivationListener [6] if an object is stored in a component and wants to called on activation and passivation.

Passivation Flow

When a session is about to be passivated (such as moving to anther machine), the activation listeners will be called first to notify the passivation, and then the serialization listeners will be called before the object is serialized.

Sequence	Description
1	Invokes SessionActivationListener.willPassivate(org.zkoss.zk.ui.Session) [8] for each object referenced by the Session [2] that will be passivated
2	Invokes DesktopActivationListener.willPassivate(org.zkoss.zk.ui.Desktop) [9] for each object referenced by each Desktop [9] that will be passivated
3	Invokes PageActivationListener.willPassivate(org.zkoss.zk.ui.Page) [10] for each object referenced by each Page [8] that will be passivated
4	Invokes ComponentActivationListener.willPassivate(org.zkoss.zk.ui.Component) [11] for each object referenced by each Component [1] that will be passivated
5	Invokes SessionSerializationListener.willSerialize(org.zkoss.zk.ui.Session) [12] for each object referenced by the Session [2] that will be passivated
6	Serializes the session
7	Invokes DesktopSerializationListener.willSerialize(org.zkoss.zk.ui.Desktop) [13] for each object referenced by each Desktop [9] that will be passivated
8	Serializes desktops of the session
9	Invokes PageSerializationListener.willSerialize(org.zkoss.zk.ui.Page) [14] for each object referenced by each Page [8] that will be passivated
10	Serializes pages of each desktop
11	Invokes ComponentSerializationListener.willSerialize(org.zkoss.zk.ui.Component) [15] for each object referenced by each Component [1] that will be passivated
12	Serializes components of each page

Activation Flow

When a session is about to be activated (such as moving from another machine), the serialization listener is called after the object has been deserialized. After all objects are deserialized, the activation listener will be called to notify a session has been activated.

Sequence	Description
1	Deserializes the session
2	Deserializes desktops of the session
3	Deserializes pages of each desktop
4	Deserializes components of each page
5	Invokes ComponentSerializationListener.didDeserialize(org.zkoss.zk.ui.Component) [16] for each object referenced by each Component [1] that will be passivated
6	Invokes PageSerializationListener.didDeserialize(org.zkoss.zk.ui.Page) [17] for each object referenced by each Page [8] that will be passivated

7	Invokes DesktopSerializationListener.didDeserialize(org.zkoss.zk.ui.Desktop) [18] for each object referenced by each Desktop [9] that will be passivated
8	Invokes SessionSerializationListener.didDeserialize(org.zkoss.zk.ui.Session) [19] for each object referenced by the Session [2] that will be passivated
9	Invokes SessionActivationListener.didActivate(org.zkoss.zk.ui.Session) [20] for each object referenced by the Session [2] that will be passivated
10	Invokes DesktopActivationListener.didActivate(org.zkoss.zk.ui.Desktop) [21] for each object referenced by each Desktop [9] that will be passivated
11	Invokes PageActivationListener.didActivate(org.zkoss.zk.ui.Page) [22] for each object referenced by each Page [8] that will be passivated
12	Invokes ComponentActivationListener.didActivate(org.zkoss.zk.ui.Component) [23] for each object referenced by each Component [1] that will be passivated

Working Thread Cannot Last Two or More Requests

Since the thread cannot be migrated from one machine to another, you couldn't use a working thread that work across multiple requests. For example, you cannot start a working thread in one request, and then invoke it in another request, since the session might be passivated between the requests.

It also implies you cannot use a working thread to handle a long operation. Rather, you have to use the so-called Echo Event.

Users of ZK 5.0.5 or prior cannot deploy the event queues for the session and application scope. However, users of ZK 5.0.6 or later have no such limitation.

Version History

Version	Date	Content

References

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

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

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

[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComponentSerializationListener.html#

[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/PageSerializationListener.html#

[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComponentActivationListener.html#

[7] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/PageActivationListener.html#

[8] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/SessionActivationListener.html#willPassivate(org.zkoss.zk.ui.Session)

[9] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/DesktopActivationListener.html#willPassivate(org.zkoss.zk.ui.Desktop)

[10] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/PageActivationListener.html#willPassivate(org.zkoss.zk.ui.Page)

[11] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComponentActivationListener.html#willPassivate(org.zkoss.zk.ui.Component)

[12] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/SessionSerializationListener.html#willSerialize(org.zkoss.zk.ui.Session)

[13] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/DesktopSerializationListener.html#willSerialize(org.zkoss.zk.ui.Desktop)

[14] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/PageSerializationListener.html#willSerialize(org.zkoss.zk.ui.Page)

[15] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComponentSerializationListener.html#willSerialize(org.zkoss.zk.ui.Component)
[16] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComponentSerializationListener.html#didDeserialize(org.zkoss.zk.ui.Component)
[17] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/PageSerializationListener.html#didDeserialize(org.zkoss.zk.ui.Page)
[18] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/DesktopSerializationListener.html#didDeserialize(org.zkoss.zk.ui.Desktop)
[19] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/SessionSerializationListener.html#didDeserialize(org.zkoss.zk.ui.Session)
[20] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/SessionActivationListener.html#didActivate(org.zkoss.zk.ui.Session)
[21] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/DesktopActivationListener.html#didActivate(org.zkoss.zk.ui.Desktop)
[22] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/PageActivationListener.html#didActivate(org.zkoss.zk.ui.Page)
[23] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ComponentActivationListener.html#didActivate(org.zkoss.zk.ui.Component)

Integration

This chapter describes how to integrate ZK with other frameworks, including how to write a JSP/JSF tag with ZK components, how to access ZK components in foreign Ajax channel, how to work with form-based framework, and so on.

Use ZK in JSP

Employment/Purpose

Basically there are two approaches to use ZK in JSP pages.

1. Use <jsp:include> to include a ZUL page.
2. Use ZK JSP Tags [4] in a JSP page directly.

Here we discuss the general concepts applicable to both approaches. For information of ZK JSP Tags, please refer to ZK JSP Tags Essentials. It is also worth to take a look at the HTML Tags section.

Prerequisite

DOCTYPE

To use ZK components correctly, the JSP page must specify DOCTYPE as follows.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>
...
```

BODY Style

By default, ZK will set the CSS style of the BODY tag to width:100%;height:100% If you prefer to have the browser to decide the height (i.e., the browser's default) for you, you could specify height:auto to the BODY tag (optional).

```
<body style="height:auto">
...
```

Browser Cache

Though optional, it is suggested to disable the browser to cache the result page. It can be done as follows.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Pragma" content="no-cache" />
    <meta http-equiv="Expires" content="-1" />
```

In addition, you could invoke the following statement in JSP to tell ZK to drop desktops once the user navigates to other URL. It is optional but it saves memory since the browser page is not cached and safe to remove if the user navigates away.

```
<%
      request.setAttribute(org.zkoss.zk.ui.sys.Attributes.NO_CACHE,
Boolean.TRUE);
%>
```

Notice that it has to be invoked before ZK JSP's zkhead tag, if ZK JSP is used, or before the first jsp:include that includes a ZUL page.

HTML Form

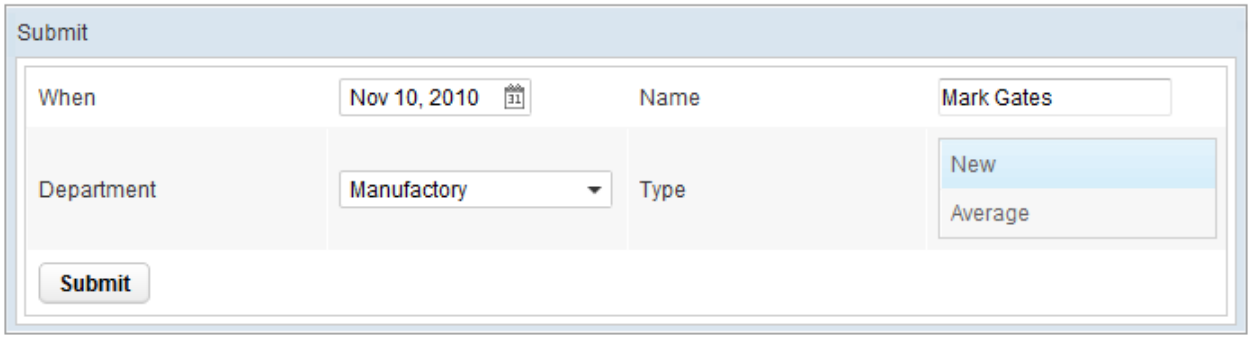
ZK input components (datebox, slider, listbox and so on) work seamlessly with HTML form. In addition to Ajax, you could process input in batch with legacy Servlets.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<%@ taglib uri="http://www.zkoss.org/jsp/zul" prefix="z" %>

<html>
  <body>
    <z:page>
    <form action="/foo/legacy">
      <table>
        <tr>
          <td>When</td><td><z:datebox name="when"/></td>
        </tr>
        <tr>
          <td>Which></td>
          <td>
            <z:listbox name="which">
                <z:listitem label="choice 1"/>
                <z:listitem label="choice 2"/>
```

```
            </z:listbox>
          </td>
        </tr>
        <tr>
          <td><z:button type="submit" label="Submit"/></td>
          <td><z:button type="reset" label="Reset"/></td>
        </tr>
    </form>
    </z:page>
  </body>
</html>
```

The name Property

If you want to submit the values of the ZK components, you have to place the component inside the form and then specify the `name` property. Thus, when the form is submitted, the value of, say, the datebox will be sent together with the name you specified. For example,




```
<window title="Submit" border="normal" xmlns:n="native">
      <n:form action="/fooLegacy">
            <grid>
                  <rows>
                        <row>
                              When
                              <datebox name="when" />
                              Name
                              <textbox name="name" />
                        </row>
                        <row>
                              Department
                              <combobox name="department">
                                    <comboitem label="RD" />
                                    <comboitem label="Manufactory" />
                                    <comboitem label="Logistics" />
                              </combobox>
                              Type
                              <listbox name="type">
                                    <listitem label="New" value="new"/>
                                    <listitem label="Average" value="avarage"/>
                              </listbox>
```

```
                        </row>
                        <row>
                              <button type="submit" label="Submit"/>
                        </row>
                    </rows>
                </grid>
            </n:form>
</window>
```

Once users press the submit button, a request is posted to the `/fooLegacy` servlet with the query string as follows.

```
?when=Nov+10%2C+2010&name=Mark+Gates&department=Manufactory&type=new
```

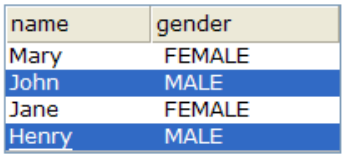
Thus, as long as you maintain the proper associations between name and value, your servlet could work as usual without any modification.

Components that Support the name Property

All input-types components support the `name` property, such as `textbox`, `datebox`, `decimalbox`, `intbox`, `combobox`, `bandbox`, `slider` and `calendar`.

In addition, the list boxes and tree controls are also support the `name` property. If the `multiple` property is true and users select multiple items, then multiple name/value pairs are posted.

```
<listbox name="who" multiple="true" width="200px">
    <listhead>
        <listheader label="name"/>
        <listheader label="gender"/>
    </listhead>
    <listitem value="mary>
        <listcell label="Mary"/>
        <listcell label="FEMALE"/>
    </listitem>
    <listitem value="john">
        <listcell label="John"/>
        <listcell label="MALE"/>
    </listitem>
    <listitem value="jane">
        <listcell label="Jane"/>
        <listcell label="FEMALE"/>
    </listitem>
    <listitem value="henry">
        <listcell label="Henry"/>
        <listcell label="MALE"/>
    </listitem>
</listbox>
```



If both John and Henry are selected, then the query string will contain:

```
who=john&who=henry
```

Notice that, to use the list boxes and tree controls with the `name` property, you have to specify the `value` property for `listitem` and `treeitem`, respectively. They are the values being posted to the servlets.

Rich User Interfaces

Because a `form` component could contain any kind of components, the rich user interfaces could be implemented independently of the existent servlets. For example, you could listen to the `onOpen` event and fulfill a tab panel as illustrated in the previous sections. Yet another example, you could dynamically add more rows to a grid control, where each row might control input boxes with the `name` property. Once user submits the form, the most updated content will be posted to the servlet.

Version History

Version	Date	Content

Spring

Overview

Spring [1] is a platform for building Java application, and it includes many easy-to-use solutions for building web-based application.

Here we discuss how to use Spring with ZK, especially the use of DelegatingVariableResolver [2]. It provides the basic support of Spring which allows a ZUML document to access variables defined in Spring. For more comprehensive support, such as Spring scopes, annotations and security, please refer to another product: ZK Spring [3].

Installing Spring

First you have to install Spring to your Web application. If you are familiar with Spring, you could skip this section. In this section we use Spring core 3.0.2 and Spring Security 3.0.2.

Copy Spring binaries into your Web library

Before using Spring, you have to download it, and put the jar file into the directory of your web application.

1. Download Spring Core framework 3.0.2 release binaries download [4]
   - org.springframework.aop-3.0.2.RELEASE.jar
   - org.springframework.asm-3.0.2.RELEASE.jar
   - org.springframework.beans-3.0.2.RELEASE.jar
   - org.springframework.context-3.0.2.RELEASE.jar
   - org.springframework.context.support-3.0.2.RELEASE.jar
   - org.springframework.core-3.0.2.RELEASE.jar
   - org.springframework.expression-3.0.2.RELEASE.jar

- org.springframework.transaction-3.0.2.RELEASE.jar
- org.springframework.web-3.0.2.RELEASE.jar
- org.springframework.web.servlet-3.0.2.RELEASE.jar

Put these jar files into your `$myApp/WEB-INF/lib/`

`Here $myApp represents the name of your web application.`

Configure web.xml

In your web.xml, you have to define `org.springframework.web.context.ContextLoaderListener`, and to specify the location of the configuration file to load bean definitions.

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

Create Spring Configuration File

Define bean definitions in `applicationContext.xml` file, and put it into your `WEB-INF` directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spri

<beans>
  <bean id="DataSource" class="test.DataSourceImpl"/>
</beans>
```

Creating Spring Bean Class

Then you have to define a DataSource interface and its implementation:

DataSource.java

```
package test;

public interface DataSource
{
    java.util.List getElementsList();
}
```

DataSourceImpl.java

```
package test;

import java.util.*;

public class DataSourceImpl implements DataSource
```

```
{
    public List getElementsList()
    {
        List list = new ArrayList();
        list.add("Tom");
        list.add("Henri");
        list.add("Jim");

        return list;
    }
}
```

Accessing Spring Bean in the ZUML page

There are two ways to access Spring-Managed beans in your ZUML page. One is using `variable-resolver`, and the other is using `SpringUtil`. Which to use depends on your usage, in the ZUML page, we suggest you to use `variable-resolver`.

Using variable-Resolver

Simply declare the `variable-resolver` with DelegatingVariableResolver [9] on top of your ZUML page, then, in the rest of your page, you can access any Spring-Managed beans directly using its bean-id.

```
<?variable-resolver class="org.zkoss.zkplus.spring.DelegatingVariableResolver"?>
<window>
  <grid>
    <rows>
      <row forEach="${DataSource.elementsList}">
        <label value="${each}"/>
      </row>
    </rows>
  </grid>
</window>
```

`variable-resolver` will look-up the bean named `DataSource` automatically for you, and returned a list to the `forEach` loop.

Use with Composer

GenericAutowireComposer [1] will wire the variables defined in the variable resolvers. Thus, we could declare DelegatingVariableResolver [9] in the ZUML document, and then declare the Spring-managed bean as a control directly in a composer. For example,

```
<?variable-resolver class="org.zkoss.zkplus.spring.DelegatingVariableResolver"?>
<window apply="foo.MyComposer">
...
    <textbox id="password"/>
...
    <button id="submit" label="Change"/>
</window>
```

Then, if a data member's name matches a Spring-managed bean, it will be wired automatically too. For example,

```
public class PasswordSetter extends GenericFowardComposer {
    private User user; //wired automatically if user is a
spring-managed bean
    private Textbox password; //wired automatically if there is a
textbox named password

    public void onClick$submit() {
        user.setPassword(password.getValue());
    }
}
```

For more information, please refer to MVC: Controller.

Using SpringUtil

`org.zkoss.zkplus.spring.SpringUtil` is a utility class which allows you to get Spring-managed beans in Java code with ease.

```
<window>
 <zscript><![CDATA[
   import org.zkoss.zkplus.spring.SpringUtil;
   import test.*;

   DataSource dataSource = SpringUtil.getBean("DataSource");
   List list = dataSource.getElementsList();
 ]]></zscript>

 <grid>
   <rows>
     <row forEach="${list}">
       <label value="${each}"/>
     </row>
   </rows>
 </grid>
</window>
```

Where the `forEach` loop is looping over the collection to print the `${each}` attribute on each object in the collection.

Version History

Version	Date	Content

References

[1] http://www.springframework.org/
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/DelegatingVariableResolver.html#
[3] http://www.zkoss.org/documentation/zkspring
[4] http://www.springframework.org/download

JDBC

ZK aims to be as thin as the presentation tier. In addition, as the code executes at the server, so connecting database is no different from any desktop applications. In other words, ZK doesn't change the way you access the database, no matter you use JDBC or other persistence framework, such as Hibernate [1].

Simplest Way to Use JDBC (but not recommended)

The simplest way to use JDBC, like any JDBC tutorial might suggest, is to use `java.sql.DriverManager`. Here is an example to store the name and email into a MySQL [2] database.

```
<window title="JDBC demo" border="normal">
     <zscript><![CDATA[
     import java.sql.*;
     void submit() {
         //load driver and get a database connetion
         Class.forName("com.mysql.jdbc.Driver");
         Connection conn = DriverManager.getConnection(

"jdbc:mysql://localhost/test?user=root&password=R3f@ct0r");
         PreparedStatement stmt = null;
         try {
             stmt = conn.prepareStatement("INSERT INTO user values(?,
?)");

             //insert what end user entered into database table
             stmt.setString(1, name.value);
             stmt.setString(2, email.value);

             //execute the statement
             stmt.executeUpdate();
         } finally { //cleanup
             if (stmt != null) {
                 try {
                     stmt.close();
                 } catch (SQLException ex) {
```

```
                    log.error(ex); //log and ignore
                }
            }
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException ex) {
                    log.error(ex); //log and ignore
                }
            }
        }
    }
    ]]>
    </zscript>
    <vbox>
        <hbox>Name : <textbox id="name"/></hbox>
        <hbox>Email: <textbox id="email"/></hbox>
        <button label="submit" onClick="submit()"/>
    </vbox>
</window>
```

Though simple, it is not recommended. After all, ZK applications are Web-based applications, where loading is unpredictable and treasurable resources such as database connections have to be managed more effectively.

Luckily, all J2EE frameworks and Web servers support a utility called connection pooling. It is straightforward to use, while managing the database connections well. We will discuss more in the next section.

> **Tip:** Unlike other Web applications, it is possible to use `DriverManager` with ZK, though *not recommended*.

First, you could cache the connection in the desktop, reuse it for each event, and close it when the desktop becomes invalid. It works just like traditional Client/Server applications. Like Client/Server applications, it works efficiently only if there are at most tens concurrent users.

To know when a desktop becomes invalid, you have to implement a listener by use of DesktopCleanup [3].

Use with Connection Pooling (recommended)

Connection pooling is a technique for creating and managing a pool of connections that are ready for use by any thread that needs them. Instead of closing a connection immediately, it keeps them in a pool such that the next connection request could be served very efficiently. Connection pooling, in addition, has a lot of benefits, such as control resource usage.

There is no reason not to use connection pooling when developing Web-based applications, including ZK applications.

The concept of using connection pooling is simple: configure, connect and close. The way to connect and close a connection is very similar the ad-hoc approach, while the configuration depends on what Web server and database server are in use.

Connect and Close a Connection

After configuring the connection pooling (which will be discussed in the following section), you could use JNDI to retrieve an connection as follows.

```
<window title="JDBC demo" border="normal">
     <zscript><![CDATA[
          import java.sql.*;
          import javax.sql.*;
          import javax.naming.InitialContext;
          void submit() {
               DataSource ds = (DataSource) new InitialContext()
                         .lookup("java:comp/env/jdbc/MyDB");

               Connection conn = null;
               PreparedStatement stmt = null;
               try {
                    conn = ds.getConnection();
                    //remember that we specify autocommit as false
in the context.xml
                    conn.setAutoCommit(true);
                    stmt = conn.prepareStatement("INSERT INTO user
values(?, ?)");
                    stmt.setString(1, name.value);
                    stmt.setString(2, email.value);
                    stmt.executeUpdate();

                    stmt.close();
                    stmt = null;
               } catch (SQLException e) {
                    conn.rollback();
                    //(optional log and) ignore
               } finally { //cleanup
                    if (stmt != null) {
                         try {
                              stmt.close();
                         } catch (SQLException ex) {
                              //(optional log and) ignore
                         }
                    }
                    if (conn != null) {
                         try {
                              conn.close();
                         } catch (SQLException ex) {
                              //(optional log and) ignore
                         }
                    }
               }
```

```
            }
      ]]>
     </zscript>
      <vbox>
            <hbox>Name :<textbox id="name" />
            </hbox>
            <hbox>Email:<textbox id="email" />
            </hbox>
            <button label="submit" onClick="submit()" />
      </vbox>
</window>
```

Notes:

- It is important to close the statement and connection after use.
- You could access multiple databases at the same time by the use of multiple connections. Depending on the configuration and J2EE/Web servers, these connections could even form a distributed transaction.

Configure Connection Pooling

The configuration of connection pooling varies from one J2EE/Web/Database server to another. Here we illustrate some of them. You have to consult the document of the server you are using.

Tomcat 5.5 (and above) + MySQL

To configure connection pooling for Tomcat 5.5, you have to edit `$TOMCAT_DIR/conf/context.xml`[4] , and add the following content under the `<Context>` element. The information that depends on your installation and usually need to be changed is marked in the blue color.

```
<!-- The name you used above, must match _exactly_ here!
    The connection pool will be bound into JNDI with the name
    "java:/comp/env/jdbc/MyDB"
-->
<Resource name="jdbc/MyDB" username="someuser" password="somepass"
    url="jdbc:mysql://localhost:3306/test"
    auth="Container" defaultAutoCommit="false"
    driverClassName="com.mysql.jdbc.Driver" maxActive="20"
    timeBetweenEvictionRunsMillis="60000"
    type="javax.sql.DataSource" />
```

Then, in `web.xml`, you have to add the following content under the `<web-app>` element as follows.

```
<resource-ref>
  <res-ref-name>jdbc/MyDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

Notes

[1] http://www.hibernate.org/
[2] http://www.mysql.com/
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/DesktopCleanup.html#

[4] Thanks Thomas Muller ((http://asconet.org:8000/antville/oberinspector)) for correction.

See also (http://tomcat.apache.org/tomcat-5.5-doc/jndi-resources-howto.html) and (http://en.wikibooks.org/wiki/ZK/How-Tos/HowToHandleHibernateSessions#Working_with_the_Hibernate_session) for more details.

JBoss + MySQL

The following instructions is based on section 23.3.4.3 of the reference manual of MySQL 5.0.

To configure connection pooling for JBoss, you have to add a new file to the directory called deploy (`$JBOSS_DIR/server/default/deploy`). The file name must end with "`*-ds.xml`" (* means the database, please refer to `$JBOSS_DIR/docs/examples/jca/`), which tells JBoss to deploy this file as JDBC Datasource. The file must have the following contents. The information that depends on your installation and usually need to be changed is marked in the blue color.

`mysql-ds.xml`:

```
<datasources>
    <local-tx-datasource>
        <!-- This connection pool will be bound into JNDI with the name
             "java:/MyDB" -->
        <jndi-name>MyDB</jndi-name>
        <connection-url>jdbc:mysql://localhost:3306/test</connection-url>
        <driver-class>com.mysql.jdbc.Driver</driver-class>
        <user-name>someuser</user-name>
        <password>somepass</password>

        <min-pool-size>5</min-pool-size>

        <!-- Don't set this any higher than max_connections on your
         MySQL server, usually this should be a 10 or a few 10's
         of connections, not hundreds or thousands -->

        <max-pool-size>20</max-pool-size>

        <!-- Don't allow connections to hang out idle too long,
         never longer than what wait_timeout is set to on the
         server...A few minutes is usually okay here,
         it depends on your application
         and how much spikey load it will see -->

        <idle-timeout-minutes>5</idle-timeout-minutes>

        <!-- If you're using Connector/J 3.1.8 or newer, you can use
             our implementation of these to increase the robustness
             of the connection pool. -->

        <exception-sorter-class-name>com.mysql.jdbc.integration.jboss.ExtendedMysqlExcep
        <valid-connection-checker-class-name>com.mysql.jdbc.integration.jboss.MysqlValid

    </local-tx-datasource>
```

```
</datasources>
```

To specify the JNDI name at which the datasource is available , you have to add a `jboss-web.xml` file under the WEB-INF folder.

`jboss-web.xml`

```
<jboss-web>
<resource-ref>
    <res-ref-name>jdbc/MyDB</res-ref-name>
    <jndi-name> java:/MyDB </jndi-name>
</resource-ref>
</jboss-web>
```

In `web.xml`, you have to add the following content under the `<web-app>` element as follows.

```
<resource-ref>
  <res-ref-name>jdbc/MyDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

JBoss + PostgreSQL

```
<datasources>
    <local-tx-datasource>
    <!-- This connection pool will be bound into JNDI with the name
         "java:/MyDB" -->
    <jndi-name>MyDB</jndi-name>

    <!-- jdbc:postgresql://[servername]:[port]/[database name] -->
    <connection-url>jdbc:postgresql://localhost/test</connection-url>

    <driver-class>org.postgresql.Driver</driver-class>
    <user-name>someuser</user-name>
    <password>somepass</password>
    <min-pool-size>5</min-pool-size>
    <max-pool-size>20</max-pool-size>
    <track-statements>false</track-statements>
    </local-tx-datasource>
</datasources>
```

download

- Please download the source(Tomcat 5.5 (and above) + MySQL) (https://sourceforge.net/projects/zkforge/files/Small Talks/JDBC(JNDI sample)/Mysql_tomcat.war/download)
- Please download the source(JBoss + MySQL) (https://sourceforge.net/projects/zkforge/files/Small Talks/JDBC(JNDI sample)/jboss+mysql.zip/download)

Version History

Version	Date	Content

Hibernate

Overview

Hibernate is an object-relational mapping (ORM) solution for the Java language. The main feature of Hibernate is that it simplifies the job of accessing a relational database.

Example here we use with Hibernate 3.3 and hsqldb 1.8. It should work with the newer version.

Installing Hibernate

Before using Hibernate, you have to install it into your application first.

1. Download hibernate core and hibernate annotations from Hibernate [1]
2. Put *.jar files from hibernate core and hibernate annotations into your $myApp/WEB-INF/lib/

$myApp represents the name of your web application. ex. event

Configuring the ZK Configuration File

To make ZK work with Hibernate smoothly, you can use the following utilities. These two utilities help applying the *open session In view* design pattern for you. They will open and close the hibernate session in each HTTP request automatically. If you don't want to use them, you will have to manage the Hibernate session yourself.

1. Create `zk.xml` under `$myApp/WEB-INF/`(if not exists)
2. Copy the following lines into your `zk.xml`

```
<!-- Hibernate SessionFactory life cycle -->
<listener>
<description>Hibernate SessionFactory life cycle</description>
<listener-class>org.zkoss.zkplus.hibernate.HibernateSessionFactoryListener</listener-cla
</listener>

<!-- Hibernate OpenSessionInView life cycle -->
<listener>
<description>Hibernate Open Session In View life cycle</description>
<listener-class>org.zkoss.zkplus.hibernate.OpenSessionInViewListener</listener-class>
</listener>
```

`$myApp` represents the name of your web application. ex. `event`

Creating the Java Objects

You have to create simple JavaBean class with some properties.

1. Create your first Java class (`Event.java`)

```
package events;

import java.util.Date;

public class Event {
    private Long id;
    private String title;
    private Date date;

    public Event() {}
        public Long getId() {
            return id;
    }
    private void setId(Long id) {
        this.id = id;
    }
    public Date getDate() {
        return date;
    }
    public void setDate(Date date) {
        this.date = date;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
}
```

1. You have to compile the Java source, and place the class file in a directory called `classes` in the Web development folder, and in its correct package. (ex.`$myApp/WEB-INF/classes/event/Event.class`)

The next step is to tell Hibernate how to map this persistent class with database.

Mapping the Java Objects

There are two ways to tell Hibernate how to load and store objects of the persistent class, one is using Java Annotation, and the other older way is using Hibernate mapping file.

Using Java Annotation

The benefit of using Java annotation instead of Hibernate mapping file is that no additional file is required. Simply add Java annotation on your Java class to tell Hibernate about the mappings.

```
package events;

import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="EVENTS")
public class Event {
    private Long id;
    private String title;
    private Date date;

    @Id
    @GeneratedValue(strategy=GenerationType.SEQUENCE)
    @Column(name = "EVENT_ID")
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }
    @Column(name = "EVENT_DATE")
    public Date getDate() {
        return date;
    }
    public void setDate(Date date) {
        this.date = date;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
```

```
    }
}
```

- @Entity declares this class as a persistence object
- @Table(name = "EVENTS") annotation tells that the entity is mapped with the table EVENTS in the database
- @Column element is used to map the entities with the column in the database.
- @Id element defines the mapping from that property to the primary key column.

Using the Mapping Files

1. Simply create `Event.hbm.xml` for the persistent class `Event.java`.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"[http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd
http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd]">

<hibernate-mapping>
    <class name="events.Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/>
        <property name="title"/>
    </class>
</hibernate-mapping>
```

1. Place this `Event.hbm.xml` in the directory called `src` in the development folder, and its correct package. (ex.`$myApp/WEB-INF/src/event/Event.hbm.xml`)

Creating the Hibernate Configuration File

The next step is to setup Hibernate to use a database. HSQL DB, a java-based SQL DBMS, can be downloaded from the HSQL DB website(http://hsqldb.org/ [2]).

1. Unzip it to a directory, say c:/hsqldb. (note: For hsqldb, the directory must be at the same partition of your web application, and under root directory)
2. Copy hsqldb.jar to `$myApp/WEB-INF/lib`
3. Open a command box and change to c:/hsqldb/lib
4. In the command prompt, execute `java -cp ../lib/hsqldb.jar org.hsqldb.Server -database.0 mydb -dbname.0 EVENTS`

After installing the database, we have to setup Hibernate Configuration file. Create `hibernate.cfg.xml` in the directory called `src` in the development folder(ex.`$myApp/WEB-INF/src/hibernate.cfg.xml`). Copy the following lines into your `hibernate.cfg.xml`. It depends on how you map your Java objects. (Note: if you're using eclipse, hibernate.cfg.xml should be placed under src folder of Java Resources)

Using Java Annotation

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <!-- Database connection settings -->
        <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
        <property name="connection.url">jdbc:hsqldb:hsql://localhost</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- Enable Hibernate's automatic session context management -->
        <property name="current_session_context_class">thread</property>

        <!-- Disable the second-level cache -->
        <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</proper

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create</property>

        <mapping class="events.Event" />
    </session-factory>
</hibernate-configuration>
```

Using the Mapping Files

```
 <?xml version='1.0' encoding='utf-8'?>
 <!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

 <hibernate-configuration>
     <session-factory>
         <!-- Database connection settings -->
         <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
         <property name="connection.url">jdbc:hsqldb:hsql://localhost</property>
         <property name="connection.username">sa</property>
```

```
        <property name="connection.password"></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

        <!-- Enable Hibernate's automatic session context management -->
        <property name="current_session_context_class">thread</property>

        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</prope

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create</property>

        <mapping resource="events/Event.hbm.xml"/>

    </session-factory>
</hibernate-configuration>
```

We continue with how to create a class to handle data accessing jobs.

Creating DAO Objects

For ease of maintenance, we used to create another Java class to handle data accessing jobs.

1. Create `EventDAO.java`

```
package events;

import java.util.Date;
import java.util.List;

import org.hibernate.Session;
import org.zkoss.zkplus.hibernate.HibernateUtil;

public class EventDAO {
    Session currentSession() {
        return HibernateUtil.currentSession();
    }
    public void saveOrUpdate(Event anEvent, String title, Date date) {
        Session sess =  currentSession();
        anEvent.setTitle(title);
        anEvent.setDate(date);
```

```
        sess.saveOrUpdate(anEvent);
    }
    public void delete(Event anEvent) {
        Session sess =  currentSession();
        sess.delete(anEvent);
    }
    public Event findById(Long id) {
        Session sess =  currentSession();
        return (Event) sess.load(Event.class, id);
    }
    public List findAll() {
        Session sess =  currentSession();
        return sess.createQuery("from Event").list();
    }
}
```

1. You have to compile the Java source, and place the class file in a directory called `classes` in the Web development folder, and in its correct package.
   (ex.`$myApp/WEB-INF/classes/event/EventDAO.class`)

Accessing Persistence Objects in ZUML Page

To access persistence objects in ZUML page is simple, simply declare a persistence object, and uses it to get data from database.

1. Create a `event.zul` in the root directory of web development. (ex. `$myApp/event.zul`)

```
<zk>
<zscript><![CDATA[

    import java.util.Date;
    import java.text.SimpleDateFormat;
    import events.Event;
    import events.EventDAO;

    //fetch all allEvents from database
    List allEvents = new EventDAO().findAll();

]]></zscript>
<listbox id="lbxEvents">
    <listhead>
        <listheader label="Title" width="200px"/>
        <listheader label="Date" width="100px"/>
    </listhead>
    <listitem forEach="${allEvents}" value="${each}">
        <listcell label="${each.title}"/>
        <zscript>String datestr = new
SimpleDateFormat("yyyy/MM/dd").format(each.date);</zscript>
        <listcell label="${datestr}"/>
```

```
    </listitem>
</listbox>
</zk>
```

1. Open a browser and visit the ZUML page. (ex. http://localhost:8080/event/event.zul)

Download

Download the sample [3]

Version History

Version	Date	Content

References

[1] http://www.hibernate.org
[2] http://hsqldb.org/
[3] http://sourceforge.net/projects/zkforge/files/Small%20Talks/Integration%20hibernate/ZkHibernate.war/download

Struts

The use of Struts [1] with ZK is straightforward: just replace JSP pages with ZUL pages. You don't need to modify action handlers, data models and others. All you need to do is to map the result view to a ZUL page instead of JSP. In addition, EL expressions will work the same way even in the ZUL page.

Use ZUL instead of JSP

First, let us take the Hello World example in Struts tutorial [2] as an example. We could provide a ZUL page called `HelloWorld.zul` to replace `HelloWorld.jsp` as follows.

```
<?page title="Hello World!"?>

<h:h2 xmlns:h="xhtml">
${messageStore.message}
</h:h2>
```

As shown, you could use the same EL expression to access the data provided by Struts and your action handler.

Then, you map the `hello` action to `HelloWorld.zul` by modifying `WEB-INF/classes/struts.xml` as follows.

```
<action name="hello" class="org.apache.struts.helloworld.action.HelloWorldAction" method=
      <result name="success">/HelloWorld.zul</result>
</action>
```

Then, you could visit http://localhost:8080/Hello_World_Struts2_Ant/hello.action as you are used to and have the same result.

Of course, it is a ZUL document. You could have any Ajax behavior you'd like.

Access Data Model of Struts in Composer

The data (so-called model) provided by Struts (or the action) can be retrieved by invoking Execution.getAttribute(java.lang.String) [3]. For example,

```
package foo;
import org.zkoss.zk.ui.util.Composer;
import org.zkoss.zk.ui.*;
import org.zkoss.zul.*;
import org.apache.struts.helloworld.model.MessageStore;

public class FooComposer implements org.zkoss.zk.ui.util.Composer {
      public void doAfterCompose(Component comp) {
            MessageStore mstore =
Executions.getCurrent().getAttribute("messageStore");
            comp.appendChild(new Label(":"+mstore.getMessage()));
      }
}
```

Submit Form

By replacing JSP with ZUML, you could enable a *static* page with ZK's power. And, you could do what any ZUML documents can do. In other words, Struts is used only for Model and Controller, while ZK for View. However, sometimes you have to redirect back to submit-based URL (maybe another action with parameters). It can be done easily by enclosing the input components with HTML FORM. For example,

```
<?taglib uri="http://www.zkoss.org/dsp/web/core" prefix="c"?>
<n:form action="${c:encodeURL('/login.action')}" method="POST" xmlns:n="native">
<grid>
  <rows>
    <row>
      User: <textbox name="user"/>
    </row>
    <row>
      Password: <textbox name="password"/>
    </row>
    <row>
      <button label="Login" type="submit"/>
    </row>
  </rows>
</grid>
</n:form>
```

As shown above, notice that

- Every input (including listbox and tree) shall be assigned with a name that will become the parameter's name when submitting the form.
- You could use the encodeURL function to encode an URL.

For more information, please refer to ZK Developer's Reference/Integration/Use_ZK_in_JSP#HTML_Form the Use ZK in JSP section.

Version History

Version	Date	Content

References

[1] http://struts.apache.org/
[2] http://struts.apache.org/2.x/hello-world-using-struts-2.html
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Execution.html#getAttribute(java.lang.String)

Portal

Configuration

Here we describe the standard configuration. Depending on the portal server, you might have more than one configuration to set. For more information, please refer to ZK Installation Guide.

WEB-INF/portlet.xml

To use it, you first have to add the following portlet definition for DHtmlLayoutPortlet [1] into `WEB-INF/portlet.xml`. Notice that `expiration-cache` must be set to zero to prevent portals from caching the result.

```
<portlet>
    <description>ZK loader for ZUML pages</description>
    <portlet-name>zkPortletLoader</portlet-name>
    <display-name>ZK Portlet Loader</display-name>

    <portlet-class>org.zkoss.zk.ui.http.DHtmlLayoutPortlet</portlet-class>

    <expiration-cache>0</expiration-cache>

    <supports>
        <mime-type>text/html</mime-type>
        <portlet-mode>VIEW</portlet-mode>
    </supports>

    <supported-locale>en</supported-locale>

    <portlet-info>
        <title>ZK</title>
        <short-title>ZK</short-title>
        <keywords>ZK,ZUML</keywords>
    </portlet-info>
</portlet>
```

WEB-INF/web.xml

ZK portlet loader actually delegates the loading of ZUML documents to ZK Loader (DHtmlLayoutServlet [2]). Thus, you have to configure `WEB-INF/web.xml` as specified in ZK Installation Guide, even if you want to use only portlets.

Use ZK Portlet

The zk_page and zk_richlet Parameter and Attribute

ZK portlet loader is a generic loader. To load a particular ZUML page, you have to specify either a request parameter, a portlet attribute or a portlet preference called `zk_page`, if you want to load a ZUML page, or `zk_richlet`, if you want to load a richlet.

More precisely, ZK portlet loader first checks the following locations for the path of the ZUML page or the richlet. The lower the number, the higher the priority.

1. The request parameter (`RenderRequest`'s `getParameter`) called `zk_page`. If found, it is the path of the ZUML page.
2. The request attribute (`RenderRequest`'s `getAttribute`) called `zk_page`. If found, it is the path of the ZUML page.
3. The request preference (`RenderRequest`'s `getPortletPreferences`'s `getValue`) called `zk_page`. If found, it is the path of the ZUML page.
4. The request parameter (`RenderRequest`'s `getParameter`) called `zk_richlet`. If found, it is the path of the richlet.
5. The request attribute (`RenderRequest`'s `getAttribute`) called `zk_richlet`. If found, it is the path of the richlet.
6. The request preference (`RenderRequest`'s `getPortletPreferences`'s `getValue`) called `zk_richlet`. If found, it is the path of the richlet.
7. The initial parameter (`PortletConfig`'s `getInitParameter`) called `zk_page`. If found, it is the path of the ZUML page.

Examples

How to pass a request parameter or attribute to a portlet depends on the portal. You have to consult the user's guide of your favorite portal for details, or refer to ZK Installation Guide.

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/http/DHtmlLayoutPortlet.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/http/DHtmlLayoutServlet.html#

ZK Filter

If you prefer to *Ajax*-ize a dynamically generated HTML page (e.g., the output of a Velocity Servlet), you could use ZK Filter to process the generated page. The content of the generated page will be interpreted by ZK Filter as a ZUML document. Thus, please make sure the output is a valid ZUML document, such as it must be a valid XML. If the output is HTML, it must be a valid XHTML document.

To enable ZK Filter, you have to configure `WEB-INF/web.xml`, as shown below.

```
<filter>
    <filter-name>zkFilter</filter-name>
    <filter-class>org.zkoss.zk.ui.http.DHtmlLayoutFilter</filter-class>
    <init-param>
        <param-name>extension</param-name>
        <param-value>html</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>zkFilter</filter-name>
    <url-pattern>/my/dyna.jsp</url-pattern>
</filter-mapping>
<filter-mapping>
    <filter-name>zkFilter</filter-name>
    <url-pattern>/my/dyna/*</url-pattern>
</filter-mapping>
```

where `url-pattern` is the servlets that you would like ZK Filter to process.

The `extension` parameter (`init-param`) defines the language of the dynamical output. By default, it is `html`, since most of legacy servlet generates a HTML document. If the output is a ZUL document, you could specify `zul` as the extension.

Notice that, if you want to filter the output from include and/or forward, remember to specify the dispatcher element with REQUEST and/or INCLUDE. Consult the Java Servlet Specification for details. For example,

```
<filter-mapping>
    <filter-name>zkFilter</filter-name>
    <url-pattern>/my/dyna/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

Performance Consideration for Filtering XHTML

If the extension is html (and the output is XHTML), it means each HTML tag will be converted to a XHTML component. It is convenient if you want to manipulate them dynamically. However, it costs more memory since ZK has to maintain the states of each HTML tags. Thus, it is suggested to use the native namespace for the portion whose content won't be changed dynamically.

ZK Filter versus UI Factory

ZK Filter is designed to handle the output of a legacy servlet. If you would like to load a ZUML document from resources other than Web pages, such as from the database, you could implement UiFactory [1]. It is generally done by extending from AbstractUiFactory [2] and overriding java.lang.String) AbstractUiFactory.getPageDefinition(org.zkoss.zk.ui.sys.RequestInfo, java.lang.String) [3]. For more information, please refer to ZK Configuration Reference.

Version History

Version	Date	Content

References

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

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

[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/impl/AbstractUiFactory.html#getPageDefinition(org.zkoss.zk.ui.sys.RequestInfo,

CDI

CDI (JSR-299 [1]) is an emerging standard for contexts and dependency injection for Java EE.

Here we discuss how to use CDI with ZK, especially the use of DelegatingVariableResolver [2]. It provides the basic support of CDI which allow a ZUML document to access variables defined in CDI. For more comprehensive support, please refer to another product: ZK CDI [24].

For more information, please refer to the following blogs:

- Integrate ZK and JSR-299 [3]
- Handling ZK Events using CDI event notification model [4]

Example

Here is a *Hello World* example. Suppose we have a Java class called HelloWorld as shown below.

```
@Named
@SessionScoped
public class HelloWorld implements Serializable {
    private final String text = "Hello World";
    public String getText() {
        return text;
    }
}
```

Then, we could access it by specifying the variable resolver: DelegatingVariableResolver [2] as shown below:

```
<?variable-resolver class="org.zkoss.zkplus.cdi.DelegatingVariableResolver"?>
<window title="ZK + CDI: Hello World" width="300px">
    My weld-injected bean says: ${helloWorld.text}
</window>
```

DelegatingVariableResolver [2] resolves all variables defined by CDI (with Java annotations). In other words, it makes them visible to the ZUML document, including EL expressions, data binding and zscript.

Setup Tomcat + Weld

Weld [5] is an implementation of CDI. Here is a brief installation instructions:

- Copy `weld-servlet.jar` to your application's `WEB-INF/lib` folder. You can find the jar file in [Weld 1.0 SP1 [6].
- Add in your application's `WEB-INF/web.xml` the following listener. This makes Weld bind with Tomcat.

```
<listener>
    <listener-class>org.jboss.weld.environment.servlet.Listener</listener-class>
</listener>
```

- In your application's `META-INF` folder, creates a `context.xml` file with following contents. This provides a JNDI reference `java:comp/env/BeanManager` for the accessing to the Weld bean manager. The ZK CDI variable resolver will need this.

```
<Context>
    <Resource name="BeanManager" auth="Container"
        type="javax.enterprise.inject.spi.BeanManager"
        factory="org.jboss.weld.resources.ManagerObjectFactory"/>
</Context>
```

- Add **WEB-INF/beans.xml** with empty content to the web root.

Version History

Version	Date	Content

References

[1] http://jcp.org/en/jsr/detail?id=299
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/cdi/DelegatingVariableResolver.html#
[3] http://blog.zkoss.org/index.php/2010/01/07/integrate-zk-and-jsr-299weld/
[4] http://blog.zkoss.org/index.php/2010/02/11/handling-zk-events-using-cdi-event-notification-model/
[5] http://docs.jboss.org/weld/reference/1.0.0/en-US/html/
[6] https://sourceforge.net/projects/jboss/files/Weld/1.0.0.SP1/weld-1.0.0.SP1.zip/download

EJB and JNDI

Enterprise JavaBeans (EJB) technology is the server-side component architecture for Java EE. Here we describe how to use it in a ZUML document.

Here we use JBoss [1] as the example. The configuration of the server might vary from one server to another, but the ZUML document is the same.

Notice that if you would like to access EJB in Java (such as in a composer or in a richlet), you could skip this section (since you could use the approach described in any EJB guide).

Use JndiVariableResolver to Resolve EJB in EL Expressions

Referencing an EJB in an EL expression is straightforward: specifying JndiVariableResolver [2] in the variable-resolver directive. For example,

```
<?variable-resolver class="org.zkoss.zkplus.jndi.JndiVariableResolver" ?>
<window>
...
</window>
```

Depending your configuration, you might have to pass extra information about JNDI to it such as:

```
<?variable-resolver class="org.zkoss.zkplus.jndi.JndiVariableResolver"
  arg0="ZkEJB3Demo"
  arg1="mail=java:comp/env/mailing,sec=java:comp/security/module" ?>
<!--
    arg0: prepend - the prepended part of JDNDI name
    arg1: mapping - the key-value pairs for JNDI names and the
corresponding variable names
```

```
-->
<window>
...
</window>
```

JndiVariableResolver [2] will resolve variables in the following order:

1. java:comp/env
2. java:comp
3. java:
4. Variable could be found as a session beans with the `prepend` argument (arg0).
5. The key-value pairs which is defined in the `mapping` argument (arg1)

Example: Retrieve Session Beans

The session beans are bound to the `java:comp/env` configured by `jboss-web.xml` and `web.xml`. For example, suppose we have them as follows:

`jboss-web.xml`:

```
<ejb-local-ref>
    <ejb-ref-name>personLocalBean</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local>demo.PersonBeanLocal</local>
    <local-jndi-name>ZkEJB3Demo/PersonBean/local</local-jndi-name>
</ejb-local-ref>
```

`web.xml`:

```
<ejb-local-ref>
    <ejb-ref-name>personLocalBean</ejb-ref-name>
    <ejb-ref-type>Session</ejb-ref-type>
    <local-home>demo.PersonBeanLocal</local-home>
    <local>demo.PersonBeanLocal</local>
</ejb-local-ref>
```

Then, we could access them as follows.

```
<?variable-resolver class="org.zkoss.zkplus.jndi.JndiVariableResolver" ?>
<listbox width="600px">
    <listhead sizable="true">
        <listheader label="name" sort="auto"/>
        <listheader label="email" sort="auto"/>
    </listhead>
    <listitem forEach="${personLocalBean.allPersons}"> <!-- resolve personLocalBean from
        <listcell label="${each.name}"/>
        <listcell label="${each.email}"/>
    </listitem>
</listbox>
```

The variables provided by a variable resolver is also available to the Java code in zscript. For example,

```
<zscript>
personLocalBean.createDemoData();
</zscript>
```

Example: Retrieve EntityManagerFactory

Persistence units are not bound into JNDI by default, so we have to define JBoss specific properties in persistence.xml to bind them into JNDI. For example,

```
</persistence-unit>
    <properties>
        <property name="jboss.entity.manager.factory.jndi.name" value="java:comp/entityMa
    </properties>
</persistence-unit>
```

Then, we could retrieve the entity manager factory by use of JndiVariableResolver [2].

Version History

Version	Date	Content

References

[1] http://jboss.org
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/jndi/JndiVariableResolver.html#

Google Analytics

To track the Ajax traffic with Google Analytics [1] or other statistic services, you have to override a client-side API: zk.Event, zk.Desktop) zAu.beforeSend(_global_.String, zk.Event, zk.Desktop) [2]. This method will be called each time ZK Client is about to send an Ajax request to the server. You could override it to record the requests on any statistic service you prefer.

Here we use Google Analytics as an example to illustrate how to override it.

```
try {
var pageTracker = _gat._getTracker("UA-xxxx"); //whatever code your
website is assigned
pageTracker._setDomainName("zkoss.org");
pageTracker._initData();
pageTracker._trackPageview();

zk.override(zAu, "beforeSend", function (uri, req) {
      try {
            var t = req.target;
            if (t && t.id && (!req.opts || !req.opts.ignorable)) {
                  var data = req.data||{},
                        value = data.items &&
data.items[0]?data.items[0].id:data.value;
                  pageTracker._trackPageview(uri +"/" + t.id + "/" +
req.name + (value?"/"+value:""));
            }
      } catch (e) {
      }
      return zAu.$beforeSend(uri, req);
});
}catch(e){
}
```

Of course, you could only record the information you are interested by examining Event.name [3].

Version History

Version	Date	Content

References

[1] http://www.google.com/analytics/
[2] http://www.zkoss.org/javadoc/latest/jsdoc/_global_/zAu.html#beforeSend(_global_.String,
[3] http://www.zkoss.org/javadoc/latest/jsdoc/zk/Event.html#name

Embed ZK Component in Foreign Framework

Employment/Purpose

Here we describe how to embed ZK component(s) as a native element of a foreign framework. For example, how ZK components can be embedded as a native JSF component. It allows application developers to use the native element without knowing the existence of ZK.

For the sake of description, we call it the embedded component.

> Note: if it is OK for your developers to work on ZUL directly. it is more convenient and powerful to use the inclusion (such as <jsp:include>) or ZK JSP Tags [4], and you don't have to wrap them into a native element.

Prerequisite

DOCTYPE

To use ZK components correctly, the pages generated by the foreign framework (JSP, JSF...) must generate the doc type as follows.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Browser Cache

Though optional, it is suggested to disable the browser to cache the result page. It can be done as follows.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Pragma" content="no-cache" />
    <meta http-equiv="Expires" content="-1" />
```

Embed a component directly

The simplest way to embed is to invoke javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse, org.zkoss.zk.ui.Component, java.lang.String, java.io.Writer) Renders.render(javax.servlet.ServletContext, javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse, org.zkoss.zk.ui.Component, java.lang.String, java.io.Writer) [1] when it is time to output the content of the native element.

For example, if you are implementing a JSP tag, then you can invoke the render method in doTag() as follows.

<syntax lang="java" high="16"> public void doTag() throws JspException, IOException {

```
   //prepare variables
   final JspContext jspctx = getJspContext();
   final PageContext pgctx = Jsps.getPageContext(jspctx);
   final ServletContext svlctx = pgctx.getServletContext();
   final HttpServletRequest request = (HttpServletRequest)pgctx.getRequest();
   final HttpServletResponse response = (HttpServletResponse)pgctx.getResponse();
```

```
   //create components
   Listbox listbox = new Listbox();
   listbox.appendChild(new Listitem("Item 1"));
   listbox.appendChild(new Listitem("Item 2"));
```

```
//render the result
   final StringWriter out = new StringWriter();
   Renders.render(svlctx, request, response, listbox, null, out);
   getJspBody().invoke(out);
```

} </syntax>

Embed by implementing a richlet

If you want to have more control, such as applying a composer provided by users or creating components from a ZUL page, you could implement a richlet (Richlet [1]) and then invoke javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse, org.zkoss.zk.ui.Richlet, java.lang.String, java.io.Writer) Renders.render(javax.servlet.ServletContext, javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse, org.zkoss.zk.ui.Richlet, java.lang.String, java.io.Writer) [1] instead.

<syntax lang="java"> Renders.render(svlctx, request, response,

```
 new GenericRichlet() {
   public void service(Page page) throws Exception {
       //execution is ready
       //... do whatever you want
       Window main = new Window();
       main.setPage(page); //associate to the page
       Executions.createComponents("/WEB-INF/template/foo.zul", main, null);
       composer.doAfterCompose(main); //assume user assigned a composer
   }
 }, null, out);
```

</syntax>

where we use GenericRichlet [2] to simplify the implementation of a richlet.

Example

Embed as a native JSF component

ZK Component as a native JSF component can be easily achieved by wrapping it as a custom JSF component [2] and rendering it in *Render Response Phase* of JSF lifecycle by invoking javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse, org.zkoss.zk.ui.Richlet, java.lang.String, java.io.Writer) Renders.render(javax.servlet.ServletContext, javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse, org.zkoss.zk.ui.Richlet, java.lang.String, java.io.Writer) [1]

<syntax lang="java" high="13"> @FacesComponent(value = "window") public class WindowTag extends UIComponentBase { private static final Log log = Log.lookup(WindowTag.class); private Window window; public void encodeBegin(FacesContext context) throws IOException { ServletContext svlctx = (ServletContext)context.getExternalContext().getContext(); HttpServletRequest request = (HttpServletRequest)

context.getExternalContext().getRequest(); HttpServletResponse response = (HttpServletResponse) context.getExternalContext().getResponse(); ResponseWriter responseWriter = context.getResponseWriter();

try { Renders.render(svlctx, request,response, new GenericRichlet() { public void service(Page page) throws Exception { window = new Window(); window.setPage(page); applyProperties(); doAfterCompose(); } }, null, responseWriter); } catch (ServletException e) { throw new IOException(e.getMessage()); } }

/** apply ZK component properties as retrieved from JSF custom component tag */ private void applyProperties() { Map<String, Object> attrs = getAttributes(); Set<String> attrNames = attrs.keySet(); for (Iterator iterator = attrNames.iterator(); iterator.hasNext();) { String attrName = (String) iterator.next(); if(!"apply".equals(attrName)) { try { Property.assign(window, attrName, attrs.get(attrName).toString()); } catch(PropertyNotFoundException pnfe) { log.debug(pnfe.getMessage()); } } } } /** apply composer by calling doAfterCompose after ZK component is composed */ private void doAfterCompose() throws Exception { Object o = getAttributes().get("apply"); if(o instanceof String) { o = Classes.newInstanceByThread(o.toString()); } if(o instanceof Composer) { ((Composer)o).doAfterCompose(window); } }

.... } </syntax>

Version History

Version	Date	Content
5.0.5	September 2010	Renders [3] was introduced to simplify the making of a native element for a foreign framework.

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/embed/Renders.html#render(javax.servlet.ServletContext,

[2] http://weblogs.java.net/blog/driscoll/archive/2009/10/09/jsf-2-custom-java-components-and-ajax-behaviors

[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/embed/Renders.html#

Start Execution in Foreign Ajax Channel

Employment/Purpose

Here we describe how to start a ZK execution in a foreign Ajax channel. For example, JSF 2 allows developers to send back JavaScript code to update the browser in JSF's Ajax channel.

Bridge [1]

Starting an execution in a foreign Ajax channel is straightforward: invoke javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse, org.zkoss.zk.ui.Desktop) Bridge.start(javax.servlet.ServletContext, javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse, org.zkoss.zk.ui.Desktop) [2]. Then, you are allowed to access the components, post events and do anything you like. At the end, you invoke Bridge.getResult() [3] to retrieve the JavaScript code snippet and send it back to the client to execute. Finally, you invoke Bridge.close() [4] to close the execution.

<syntax lang="java"> Bridge bridge = Bridge.start(svlctx, request, response, desktop); try {

```
   //execution started, do whatever you want
```

```
   String jscode = bridge.getResult();
```

```
//send jscode back with the foreign Ajax channel.
```

} finally {

```
   bridge.close(); //end of execution and cleanup
```

} </syntax>

Example

Start Execution in JSF 2 ActionListener

In JSF 2.0 developers can initiate Ajax request using jsf.ajax.request [5] For e.g. <syntax lang="xml"> ...

```
<h:commandButton id="save" value="Save"
  onclick="jsf.ajax.request(this, event, {execute:'@all'}); return false;" actionListener="${myBean.saveDetails}">
</h:commandButton>
```

... </syntax>

and in your ActionListener

<syntax lang="java" high="12,22,28"> @ManagedBean @SessionScoped public class MyBean {

```
 public void saveDetails(ActionEvent e)  throws IOException {
```

```
    ExternalContext ec = FacesContext.getCurrentInstance().getExternalContext();
    ServletContext svlctx = (ServletContext) ec.getContext();
    HttpServletRequest request = (HttpServletRequest) ec.getRequest();
    HttpServletResponse response = (HttpServletResponse) ec.getResponse();
    Component comp = getComponent();
    Bridge bridge = Bridge.start(svlctx, request, response,comp.getDesktop());
```

```
     try {
              // update ZK component(s) state here
              //comp.appendChild(new SomethingElse()); ...
```

```
             //Send back bridge.getResult() with the response writer (eval)
             PartialResponseWriter responseWriter =
               FacesContext.getCurrentInstance().getPartialViewContext().getPartialResponseWriter();
             responseWriter.startDocument();
             responseWriter.startEval();
             responseWriter.write(bridge.getResult());
             responseWriter.endEval();
             responseWriter.endDocument();
             responseWriter.flush();
             responseWriter.close();
     } finally {
             bridge.close();
     }
 }
```

```
private Component getComponent() {
     //locate the component that you want to handle
 }
```

} </syntax>

---

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/embed/Bridge.html#

[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/embed/Bridge.html#start(javax.servlet.ServletContext,

[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/embed/Bridge.html#getResult()

[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/embed/Bridge.html#close()

[5] For more information on jsf.ajax.request (https://javaserverfaces.dev.java.net/nonav/docs/2.0/jsdocs/symbols/jsf.ajax.html#.request) read official JSF Javascript docs for jsf.ajax (https://javaserverfaces.dev.java.net/nonav/docs/2.0/jsdocs/symbols/jsf.ajax.html).

Version History

Version	Date	Content
5.0.5	September 2010	Bridge (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/embed/Bridge.html#) was introduced to simplify the starting of an execution in foreign Ajax channel

Use ZK as Fragment in Foreign Templating Framework

Employment/Purpose

Here we describe how to make a ZUL page to be assembled at the client by using Ajax to request ZUL pages separately in a foreign templating framework[1] .

You could skip this chapter if you'd like to use ZK's templating technology, such as Templating: composition, Servlet's inclusion (javax.servlet.RequestDispatcher's include) and macro components.

ZK also supports many powerful layout components, such as portallayout, borderlayout, tablelayout, columnlayout and so on[2] . You could use them to have similar or better effect, and skip this chapter.

---

[1] Apache Tiles (http://tiles.apache.org/) is a typical templating framework and allows developers to assemble UI at both server and client.

[2] For more information, please refer to ZK Component Reference.

Prerequisite

DOCTYPE

To use ZK components correctly, the templating page must specify DOCTYPE as follows.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html>
...
```

Browser Cache

Though optional, it is suggested to disable the browser to cache the result page. It can be done as follows.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="Pragma" content="no-cache" />
    <meta http-equiv="Expires" content="-1" />
```

Make a ZUL page as a fragment

Include a ZUL page when receiving a request

By default, if a ZUL page is requested by the browser directly, it will generate a complete HTML structure, including HTML, HEAD and BODY tags. On the other hand, if the assembling is done by inclusion (javax.servlet.RequestDispatcher's include), a ZUL page will be generated as a HTML fragment without HTML, HEAD, and BODY. For example, if a ZUL page is included by jsp:include, then it won't generate HTML/HEAD/BODY, such that the following JSP page will be rendered correctly.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<%-- a JSP page --%>
<html>
  <body>
    <jsp:include page="frag.zul"/>
...
```

In other words, if the result page is assembled when the request is received, you don't need to do anything specially[1] . However, if the assembling is done at the client side by using Ajax to request fragments after loaded, you have to read the following section.

---

[1] You might take a look at Use ZK in JSP for more information.

Load a ZUL page with an Ajax request

As described above, if a ZUL page is requested by the browser directly, it will, by default, generate a complete HTML structure, including HTML, HEAD and BODY tags. To disable it, you could specify a special parameter called `zk.redrawCtrl=page`. For example, you might have a HTML page that loads a ZUL page at the client with jQuery as follows.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtm
<html>
      <head>
            <title>Mash-up of ZUML apges</title>
            <script src="http://code.jquery.com/jquery-1.4.2.min.js">
            </script>
      </head>
      <body>
            <div id="anchor"></div>
            <button onclick="$('#anchor').load('foo.zul?zk.redrawCtrl=page')">Load the fr
      </body>
</html>
```

The `zk.redrawCtrl` parameter is used to control how a ZUL page is specified. In this case, since `page` is specified, the generation of HTML, HEAD and BODY tags are disabled.

Alternative: using the request-scoped attribute called `org.zkoss.zk.ui.page.redrawCtrl`

If a ZUL page is always loaded as a fragment by the client, you could specify the request-scoped attribute called `org.zkoss.zk.ui.page.redrawCtrl` (Attributes.PAGE_REDRAW_CONTROL (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/sys/Attributes.html#PAGE_REDRAW_CONTROL)) with `page`, such that the generation of HTML, HEAD and BODY tags are always disabled no matter if the `zk.redrawCtrl` parameter is specified or not.

For example,

```
<window title="whatever content you want"/>
  <custom-attributes scope="request" org.zkoss.zk.ui.page.redrawCtrl="page"/>
  ...
</window>
```

Then, you don't need to specify the `zk.redrawCtl` parameter when loading it at the client (e.g., $('#anchor').load('foo.zul')).

Of course, if the fragment itself is a JSP page and then use inclusion to include a ZUL page (or use ZK JSP Tags), then the generated HTML structure is already a correct HTML fragment (and you don't need to anything described above).

Server-side memory optimization: turn off browser cache

As described in [ZK in JSP (http:/ / books. zkoss. org/ wiki/ ZK_Developer's_Reference/ Integration/ Use_ZK_in_JSP#Browser_CachelUse)], the memory footprint at the server can be improved by turning off the browser cache for the HTML page that will load ZUL pages later. For example, we could add `no-cache` and `expires` as follows (line 4 and 5):

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtm
<html>
      <head>
            <meta http-equiv="Pragma" content="no-cache" />
            <meta http-equiv="Expires" content="-1" />
            <title>Mash-up of ZUML apges</title>
            <script src="http://code.jquery.com/jquery-1.4.2.min.js">
            </script>
      </head>
      <body>
            <div id="anchor"></div>
            <button onclick="$('#anchor').load('foo.zul')">Load the fragment</button>
      </body>
</html>
```

In addition, we have to specify a request-scoped attribute called `org.zkoss.zk.desktop.nocache` in the ZUL page being loaded as follows:

```
<window title="whatever content you want"/>
  <custom-attributes scope="request" org.zkoss.zk.desktop.nocache="true"
    org.zkoss.zk.ui.page.redrawCtrl="page"/>
  ...
</window>
```

Note: since 5.0.8, assigning `page` to the `zk.redrawCtrl` parameter implies *no-cache*, i.e., `zk.redrawCtrl=true` implies `org.zkoss.zk.desktop.nocache="true"`.

ID Generator

Each ZUL page we request by Ajax as described above will be an independent desktop. It means the browser window will have several desktops, if we assemble UI this way. Thus, the component's UUID must be unique across different desktops (of the same session[1]). The default ID generator can handle it well.

However, if you use a customized IdGenerator (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/sys/IdGenerator. html#), you have to generate component's UUID (org.zkoss.zk.ui.Component) IdGenerator.nextComponentUuid(org.zkoss.zk.ui.Desktop, org.zkoss.zk.ui.Component) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zk/ ui/ sys/ IdGenerator. html#nextComponentUuid(org. zkoss. zk. ui. Desktop,)) correctly. A typical trick is to encode desktop's ID as part of component's UUID.

[1] In short, component's UUID must be unquie in the same session. It is OK to be duplicated in different session.

Communicate among ZUL pages

If a ZUL page is loaded separately with Ajax, an independent desktop is created. For example, the following HTML page will create three desktops.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Pragma" content="no-cache" />
<meta http-equiv="Expires" content="-1" />
<script type="text/javascript"
      src="http://code.jquery.com/jquery-1.4.2.js"></script>

<title>Assembling at the client with Ajax</title>
</head>
<body>
<table>
      <tr>
            <td id="top" colspan="2">top</td>
      </tr>
      <tr>
            <td id="left">left</td>
            <td id="right">right</td>
      </tr>
</table>
<script>
      $(function() {
            $.get("/frags/banner.zul",
                        {width : "600px"},
                        function(response) {
                          $("#top").html(response);
                    }
            );
            $.get("/frags/leftside.zul",
                        {width : "300px"},
                    function(response) {
                          $("#left").html(response);
                    }
            );
            $.get("/frags/rightside.zul",
                {width : "300px"},
                function(response) {
                  $("#right").html(response);
                }
            );
```

```
        });
</script>
</body>
</html>
```

Since they are in different desktops, you have to use the *group-scoped* event queue[1] if you want to send events from one desktop (such as leftside.zul) to another (such as rightside.zul). For more information, please refer to Event Queues.

[1] The group-scoped event queue is available only in ZK EE. For ZK CE, you have to use the session-scoped event queue.

Version History

Version	Date	Content
5.0.5	October, 2010	ZUL page is able to be generated as a HTML fragment.

Performance Tips

This chapter describes the tips to make your ZK application running faster. For information about identifying the bottleneck, please refer to the Performance Monitoring section.

Use Compiled Java Codes

Not to Use zscript for Better Performance

It is convenient to use zscript in ZUML, but it comes with a price: slower performance. The degradation varies from one application from another. It is suggested to use zsript only for fast prototyping, POC, or small projects. For large website, it is suggested to use ZK MVC instead. For example,

<syntax lang="xml">

```
<window apply="foo.MyComposer">
```

//omitted </syntax>

Then, you can handle all events and components in foo.MyComposer. By the use of auto-wiring, it is straightforward to handle events and components.

Event Handler Is zscript

In addition to the `zscript` element, the event handler declared in a ZUL page is also interpreted at the runtime. For example,

<syntax lang="xml">

```
 <button label="OK" onClick="doSomething()"/>
```

</syntax>

where `doSomething()` is interpreted as `zscript`. Thus, for better performance, they should be replaced too.

Turn off the use of zscript

```
[since 5.0.8]
```

If you decide not to use zscript at all, you could turn on the disable-script configuration as follows, such that an exception will be thrown if zscript is used.

```
<system-config>
      <disable-zscript>true</disable-zscript>
</system-config>
```

Use the deferred Attribute

If you still need to write zscript codes, you can specify the `deferred` attribute to defer the evaluation of zscript codes as follows.

<syntax lang="xml" > <zscript deferred="true"> </zscript> </syntax>

By specifying the `deferred` attribute, the zscript codes it contains will not be evaluated when ZK renders a page. It means that the interpreter won't be loaded when ZK renders a page. This saves memory and speeds up page rendering.

In the following example, the interpreter is loaded only when the button is clicked:

<syntax lang="xml" > <window id="w">

```
   <zscript deferred="true">
    void addMore() {
        new Label("More").setParent(w);
    }
   </zscript>
   <button label="Add" onClick="addMore()"/>
```

</window> </syntax>

The deferred Attribute and the onCreate Event

It is worth to notice that, if the `onCreate` event listener is written in zscript, the deferred option mentioned in the previous second becomes *useless*. It is because the `onCreate` event is sent when the page is loaded. In other words, all deferred zscript will be evaluated when the page is loaded if the `onCreate` event listener is written in zscript as shown below.

<syntax lang="xml" > <window onCreate="init()"> </syntax>

Rather, it is better to rewrite it as

<syntax lang="xml" > <window use="my.MyWindow"> </syntax>

Then, prepare `MyWindow.java` as shown below.

<syntax lang="java" >

```
package my;
public class MyWindow extends Window {
    public void onCreate() { //to process the onCreate event
...
```

</syntax>

If you prefer to do the initialization right after the component (and all its children) is created, you can implement the AfterCompose [1] interface as shown below. Note: the `afterCompose` method of the `AfterCompose` interface is evaluated at the Component Creation phase, while the `onCreate` event is evaluated in the Event Processing Phase.

<syntax lang="java" >

```
package my;
public class MyWindow extends Window implements org.zkoss.zk.ui.ext.AfterCompose {
    public void afterCompose() { //to initialize the window
...
```

</syntax>

Use the forward Attribute

To simplify the event flow, ZK components usually send the events to the component itself, rather than the parent or other targets. For example, when an user clicks a button, the `onClick` event is sent to the button. Developers usually forward the event to the window by the use of the `onClick` event listener as follows.

<syntax lang="xml" > <window id="w">

```
   <button label="OK" onClick="w.onOK"/>
```

</syntax> As suggested in the previous sections, the performance can be improved by *not* using zscript at all. Thus, you can rewrite the above code snippet either with `EventListener` or by specifying the `forward` attribute as follows.

<syntax lang="xml" > <window>

```
   <button label="OK" forward="onOK"/>
```

</syntax>

Version History

Version	Date	Content

References

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

Use Native Namespace instead of XHTML Namespace

ZK creates a component (one of the derives of AbstractTag [1]) for each XML element specified with the XHTML component set. In other words, ZK will maintain their states on the server. However, if you won't change their states dynamically (i.e., after instantiated), you could use the native namespace instead.

For example, the following code snippet creates five components (one Table [2], Tr [3], Textbox [4] and two Td [5]).

<syntax lang="xml" > <h:table xmlns:h="xhtml">

```
<h:tr>
    <h:td>Name</h:td>
    <h:td>
    <textbox/>
    </h:td>
</h:tr>
```

</h:table> </syntax>

On the other hand, the following code snippet won't create components for any elements specified with the native space (with prefix n:)[6] .

<syntax lang="xml" > <n:table xmlns:n="native">

```
<n:tr>
    <n:td>Name</n:td>
    <n:td>
    <textbox/>
    </n:td>
</n:tr>
```

</n:table> </syntax>

Notice that `table`, `tr` and `td` are generated directly to the client, so they don't have no counterpart at the client either. You cannot change their states dynamically. For example, the following code snippet is incorrect.

<syntax lang="xml" > <n:ul id="x" xmlns:n="native"/> <button label="add" onClick="new Li().setParent(x)"/> </syntax>

If you have to change them dynamically, you still have to use the XHTML component set, or you could use Html [7] alternatively, if the HTML tags won't contain any ZUL component.

Notice that you could create the native components in Java too. For more information, please refer to the native namespace section.

```
    HtmlNativeComponent n =
        new HtmlNativeComponent("table", "<tr><td>When:</td><td>", "</td></tr>");
    n.setDynamicProperty("border", "1");
    n.setDynamicProperty("width", "100%");
    n.appendChild(new Datebox());
    parent.appendChild(n);
```

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zhtml/AbstractTag.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zhtml/Table.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zhtml/Tr.html#
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Textbox.html#
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zhtml/Td.html#
[6] In fact, it will still create some components for the rerender purpose, such as Component.invalidate() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#invalidate()). However, since they shall not be accessed, you could image there are not created at all.
[7] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Html.html#

The Stub-izing of Native Components

By default, a native component will be *stub-ized*, i.e., they will be replaced with a stateless component called StubComponent (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/StubComponent.html#), such that the memory footprint will be minimized[1]

Though rarely, you could disable the stubing by setting a component attribute called org.zkoss.zk.ui.stub.native (i.e., Attributes.STUB_NATIVE (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zk/ ui/ sys/ Attributes. html#STUB_NATIVE)). A typical case is that supposed you have a component that has a native descendant, and you'd like to detach it and re-attach later. Then, you have to set this attribute to false, since the server does not maintain the states of stub-ized components (thus, it cannot be restored when attached back).

```
<div>
    <custom-attributes org.zkoss.zk.ui.stub.native="false"/>
    <n:table xmlns:n="native"> <!-- won't be stub-ized -->
...
```

Once set, descendant components unless it was set explicitly.

[1] Non-native components could be stub-ized too by use of Component.setStubonly(java.lang.String) (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#setStubonly(java.lang.String)). For more information, please refer here.

Version History

Version	Date	Content
5.0.6	January, 2011	The attribute called org.zkoss.zk.ui.stub.native was introduced to disable the *stub-ization*.

Use ZK JSP Tags instead of ZK Filter

The ZK filter actually maps each HTML tag to the corresponding XHTML components. As described in the previous section, it consumes more memory than necessary since ZK has to maintain the states of all ZK components (including XUL and XHTML components).

Include ZUL pages in a JSP page

If some part of UI is made of HTML tags (such as header and banner), you could use JSP as the main page, implement the parts with dynamic content in ZUL, and then put them together with <jsp:include>. For example,

<syntax lang="xml"> <%-- main.jsp --%> <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http:/ / www. w3. org/ TR/ xhtml1/ DTD/ xhtml1-transitional. dtd"> <html xmlns="http://www.w3.org/1999/xhtml">

```
<head>
</head>
<body>
  <%-- the static part such as header --%>
```

any content you like

```
<%-- include the dynamic part --%>
  <jsp:include page="foo.zul"/>
```

...

```
</body>
```

</head> </syntax>

Use ZK components directly in a JSP page with ZK JSP tags

If you prefer to use ZK components directly in a JSP page, you could use ZK JSP tags [4]. For example,

<syntax lang="xml" > <%-- another.jsp --%> <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"

```
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

<%@ taglib uri="http://www.zkoss.org/jsp/zul" prefix="z" %>

<html xmlns="http://www.w3.org/1999/xhtml">

```
<head>
</head>
<body>
 <%-- any JSP content --%>
```

<z:page>

Name <z:textbox/>

</z:page> </body> </head> </syntax>

where z:page declares a ZK page and then ZK tags can be used inside it.

The above example is equivalent to the following code snippet, if a ZUL page is used,

<syntax lang="xml" > <?page complete="true"?> <n:html xmlns="http://www.w3.org/1999/xhtml" xmlns:n="http://www.zkoss.org/2005/zk/native">

```
<n:head>
</n:head>
<n:body>
```

<n:table>

```
   <n:tr>
       <n:td>Name</n:td>
       <n:td><textbox/></n:td>
   </n:tr>
```

</n:table> </syntax>

where <?page complete="true"?> declares that this page is a *complete* page, i.e., it will provide HTML's html, head and body tags as shown above.

Version History

Version	Date	Content

Defer the Creation of Child Components

For sophisticated pages, the performance can be improved if we defer the creation of child components until they are becoming visible. The simplest way to do this is by the use of the `fulfill` attribute. In the following example, the children of the second tab panel are created only if it becomes visible.

<syntax lang="xml" > <tabbox>

```
   <tabs>
       <tab label="Preload" selected="true"/>
       <tab id="tab2" label="OnDemand"/>
   </tabs>
   <tabpanels>
       <tabpanel>
```

This panel is pre-loaded since no fulfill specified

```
       </tabpanel>
       <tabpanel fulfill="tab2.onSelect">
```

This panel is loaded only tab2 receives the onSelect event

```
       </tabpanel>
   </tabpanels>
```

</tabbox> </syntax>

For more information, please refer to the On-demand Evaluation section.

Version History

Version	Date	Content

Defer the Rendering of Client Widgets

```
[since 5.0.2]
```

In addition to Defer the Creation of Child Components, you can defer the rendering of the widgets at the client by the use of the renderdefer attribute. It is a technique to make a sophisticated page to appear earlier.

For example, we can defer the rendering of the inner window for 100 milliseconds as shown below

<syntax lang="xml"> <window title="Render Defer" border="normal"> The following is rendered after 100 milliseconds. <window title="inner" width="300px" height="200px" border="normal" renderdefer="100"> Enter something <datebox onChange='i.value = self.value + ""'/> <separator/> <label id="i"/> <separator bar="true"/> <button label="say hi" onClick='alert("Hi")'/> </window> </window> </syntax>

Unlike the fulfill attribute, the components on the server and the widgets at the client are created no matter renderdefer is specified. It only defers and the rendering of the widgets into DOM elements.

Here is another example to use it with pure Java.

<syntax lang="java"> Tabpanel tp = new Tabpanel(); tp.setRenderdefer(0); </syntax>

The render-defer technique is useful to improve the response time of showing a sophisticated page in a slow client. The total time required to render is not reduced (since all widgets have to render later), but it allow the page to show up sooner and it makes the user feel more responsive.

Version History

Version	Date	Content

Client Render on Demand

```
Available in ZK EE only [1]
[Since 5.0.0]
```

With Enterprise Edition, widgets[1] will delay the rendering of DOM elements until really required. For example, the DOM elements of `comboitem` won't be created until the drop down is shown up. It improved the performance a lot for a sophisticated user interface.

This feature is transparent to the application developers. All widgets are still instantiated (though DOM elements might not), so they can be accessed without knowing if this feature is turned on.

---

[1] A widget is the (JavaScript) object running at the client to represent a component

Client ROD: Tree

Client ROD is enabled only if a tree item is closed. Thus, to have the best performance (particularly for a huge tree), it is better to make all tree item closed initially.

<syntax lang="xml"> <treeitem forEach="${data}" open="false"> <treerow> <treecell label="${each.name}"/> <treecell label="${each.description}"/> </treerow>

```
   <treechildren>
       <treeitem forEach="${each.detail}" open="false">
```

<treerow> <treecell label="${each.name}"/> <treecell label="${each.description}"/> </treerow>

```
           <treechildren>
               <treeitem forEach="${each.fine}" open="false">
```

<treerow> <treecell label="${each.name}"/> <treecell label="${each.description}"/> </treerow> </treeitem> </treechildren> </treeitem> </treechildren> </treeitem> </syntax>

Client ROD: Groupbox

Client ROD is enabled only if a groupbox is closed. Thus, to have the best performance (particularly with sophisticated content), it is better to make the groupbox closed initially if proper.

Client ROD: Panel

Client ROD is enabled only if a panel is closed. Thus, to have the best performance (particularly for with sophisticated content), it is better to make the panel closed initially if proper.

Enable or Disable Client ROD

If you want to disable Client ROD for the whole application, you can specify a library property called `org.zkoss.zul.client.rod` rod with false. For example, specify the following in zk.xml:

<syntax lang="xml"> <library-property> <name>org.zkoss.zul.client.rod</name> <value>false</value> </library-property> </syntax>

Or, if you prefer to disable it for a particular page, then specify false to a page's attribute called `org.zkoss.zul.client.rod` rod, such as

<syntax lang="xml"> <custom-attributes org.zkoss.zul.client.rod="false" scope="page"/> </syntax>

Or, if you prefer to disable it for all descendants of a particular component, then specify false to a component's attribute. And, you can enable it for a subset of the descendants. For example,

<syntax lang="xml"> <window>

```
<custom-attributes org.zkoss.zul.client.rod="false"/>
```

```
  <custom-attributes org.zkoss.zul.client.rod="true"/>
```

..

</window> </syntax>

Version History

Version	Date	Content

Listbox, Grid and Tree for Huge Data

This section we will discuss how to make a listbox, grid and tree to serve huge amount of data effectively.

Use Live Data and Paging

Sending out a listbox/grid/tree with a lot of items to the client is expensive. In addition, the JavaScript engines of some browsers are not good for initializing a listbox/grid/tree with a lot of items. A better solution is to use the live data, i.e., by assigning a model (such as ListModel [2]) to it. Then, the items are sent to the client only if they become visible.

In addition, the performance will be improved more if you also use the paging mold such as

```
<listbox model="${mymodel}" mold="paging">
...
```

For more information of using and implementing a model, please refer to the Model section and ZK Component Reference: Listbox.

Version History

Version	Date	Content

Turn on Render on Demand

```
[ZK EE]
[Since 5.0.0]
```

With ZK EE, you can enable **Render on Demand** for Grid and Listbox to boost performance to show huge amount of data. Grid and Listbox will load only the necessary data chunk from associated ListModel, render required Row(s)/Listitem(s) on the server, then create only the required corresponding widgets and render the DOM elements really needed in browser. It improves the performance and saves memory significantly on both the server and browser sides.

Note: ROD actually brings performance boost on both the client side and server side. However, if you use `forEach` to populate Rows or Listitems, the components will be all in memory, which does not give you any performance benefits on server side. (The client side still enjoys a boost.) If you want to fully leverage the power of ROD, the use of model is recommended.

ROD: Grid

If you want to enable Grid ROD for the whole application, you can specify a library property called grid (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/grid.html#) rod with `true`. For example, specify the following in zk.xml:

<syntax lang="xml"> <library-property> <name>org.zkoss.zul.grid.rod</name> <value>true</value> </library-property> </syntax>

Or, if you prefer to enable it for a particular page, then specify `true` to a page's attribute called grid (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/grid.html#) rod, such as

<syntax lang="xml"> <custom-attributes org.zkoss.zul.grid.rod="true" scope="page"/> </syntax>

Or, if you prefer to enable it for all descendant grids of a particular component, then specify `true` to the component's attribute. You can enable it for a subset of the descendant grids. For example,

<syntax lang="xml"> <window>

```
<custom-attributes org.zkoss.zul.grid.rod="true"/>
<grid ...>
  ..
</grid>
```

```
  <custom-attributes org.zkoss.zul.grid.rod="false"/>
    <grid ...>
      ..
    </grid>
    ..
```

</window> </syntax>

Note that Grid ROD is not working unless the Grid is configured with a limited **view port**; i.e. you have to set `height` or `vflex` attribute of the Grid or set the Grid to `paging` mold so the user can see only a portion of the content of the Grid.

Specifies the number of rows rendered

```
[default: 100]
[inherit: true]
[since 5.0.8]
```

Specifies the minimum number of rows rendered on the client. It is only considered if Grid is using live data (Grid.setModel(ListModel) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zul/ Grid. html#setModel(ListModel))) and not using paging mold (Grid.getPagingChild() (http://www.zkoss.org/javadoc/ latest/ zk/ org/ zkoss/ zul/ Grid. html#getPagingChild())). <syntax lang="xml"> <custom-attributes org.zkoss.zul.grid.initRodSize="30"/> </syntax>

ROD: Listbox

If you want to enable Listbox ROD for the whole application, you can specify a library property called listbox (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zul/ listbox. html#) rod with `true`. For example, specify the following in zk.xml:

<syntax lang="xml"> <library-property> <name>org.zkoss.zul.listbox.rod</name> <value>true</value> </library-property> </syntax>

Or, if you prefer to enable it for a particular page, then specify `true` to a page's attribute called listbox (http:// www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/listbox.html#) rod, such as

<syntax lang="xml"> <custom-attributes org.zkoss.zul.listbox.rod="true" scope="page"/> </syntax>

Or, if you prefer to enable it for all descendant listboxs of a particular component, then specify `true` to the component's attribute. And, you can enable it for a subset of the descendant listboxs. For example,

<syntax lang="xml"> <window>

```
<custom-attributes org.zkoss.zul.listbox.rod="true"/>
<listbox ...>
  ..
</listbox>
```

```
  <custom-attributes org.zkoss.zul.listbox.rod="false"/>
    <listbox ...>
      ..
    </listbox>
    ..
```

</window> </syntax>

Note that Listbox ROD is not working unless the Listbox is configured with a limited **view port**; i.e. you have to set `height`, `vflex`, or `rows` attribute of the Listbox or set the Listbox to `paging` mold so the user sees only a portion of the content of the Listbox.

Specifies the number of items rendered

```
[default: 100]
[inherit: true]
[since 5.0.8]
```

Specifies the number of items rendered when the Listbox first render. It is used only if live data (Listbox.setModel(ListModel) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zul/ Listbox. html#setModel(ListModel))) and not paging (Listbox.getPagingChild() (http://www.zkoss.org/javadoc/latest/ zk/ org/ zkoss/ zul/ Listbox. html#getPagingChild())). <syntax lang="xml"> <custom-attributes org.zkoss.zul.listbox.initRodSize="30"/> </syntax>

Version History

Version	Date	Content

Implement ListModel and TreeModel

The default implementation of models, such as ListModelList [1] and DefaultTreeModel [4] assumes all data are available in the memory. It is not practical if a model has a lot of data. For huge amount of data, it is suggested to implement your own model by loading and caching only one portion of data at a time.

To implement your own model, you could extend from AbstractListModel [6], AbstractGroupsModel [4] and DefaultTreeModel [4] as described in the Model section. To implement a model that supports sorting, you have to implement Sortable [2] too. Each time an user requires sorting, boolean) Sortable.sort(java.util.Comparator, boolean) [3] will be called and the implementation usually clears the cache and re-generate the SQL statement accordingly.

Here is some pseudo code:

```
public class FooListModel extends AbstractListModel implements Sortable
 {
    private int _size = -1;
    private Object[] _cache;
    private int _beginOffset;
    private String _orderBy;
    private boolean _ascending, _descending;
    private Comparator _sorting;

    public int getSize() {
        if (_size < 0)
            _size = /**SELECT COUNT(*) FROM ...*/
        return _size;
    }
    public Object getElementAt(int index) {
        if (_cache == null || index < _beginOffset || index >= _beginOffset + _cache.leng
           loadToCache(index, 100); //SELECT ... FROM .... OFFSET index
 LIMIT 100
               //if _ascending, ORDER BY _orderBy ASC
```

```
                //if _descending, ORDER BY _orderBy DSC
        }
        return _cache[index - _beginOffset];
    }
    @Override
    public void sort(Comparator cmpr, boolean ascending) {
        _cache = null; //purge cache
        _size = -1; //so size will be reloaded
        _descending = !(_ascending = ascending);
        _orderBy = ((FieldComparator)cmpr).getRawOrderBy();
        _sorting = cmpr;
             //Here we assume sort="auto(fieldName)" is specified in
ZUML, so cmpr is FieldComparator
             //On other hand, if you specifies your own comparator,
such as sortAscending="${mycmpr}",
             //then, cmpr will be the comparator you assigned
        fireEvent(ListDataEvent.CONTENTS_CHANGED, -1, -1);
    }
    @Override
    public String getSortDirection(Comparator cmpr) {
        if (Objects.equals(_sorting, cmpr))
            return _ascending ?  "ascending" : "descending";
        return "natural";
    }
}
```

The implementation of boolean) Sortable.sort(java.util.Comparator, boolean) [3] generally has to purge the cache, store the sorting direction and field, and then fire ListDataEvent.CONTENTS_CHANGED [4] to reload the content.

The field to sort against has to be retrieved from the given comparator. If you specify "auto(fieldName)" to Listheader.setSort(java.lang.String) [5], then the comparator is an instance of FieldComparator [19], and you could retrieve the field's name from FieldComparator.getRawOrderBy() [23].

If you'd like to use your own comparator, you have to carry the information in it and then retrieve it back when boolean) Sortable.sort(java.util.Comparator, boolean) [3] is called.

Also notice that we cache the size to improve the performance, since ListModel.getSize() [3] might be called multiple times.

For a real example, please refer to Small Talk: Handling sortable huge data using ZK and/or Small Talk: Handling huge data using ZK.

Version History

Version	Date	Content
6.0.0	02/03/2012	Sortable interface

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModelList.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Sortable.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ext/Sortable.html#sort(java.util.Comparator,
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/event/ListDataEvent.html#CONTENTS_CHANGED
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Listheader.html#setSort(java.lang.String)

Minimize Number of JavaScript Files to Load

Overview

ZK loads the required JavaScript files only when necessary. It is similar to Java Virtual Machine's class loader, but ZK's JavaScript loader loads a JavaScript package once at a time. It minimizes the number of bytes to be loaded to a browser. However, with an Internet connection, a Web page is loaded faster if the number of files to load is smaller (assuming the total number of bytes to transmit is the same).

ZK, by default, loads both `zul` and `zul.wgt` packages when the `zk` package is loaded, since they are most common packages a ZK page might use. A ZK page generally uses more packages than that, and you, as an application developer, can pack them together to minimize the number of JavaScript files.

Notice that the more packages you packed, the larger the file it is. It will then slow down the load time if some of packages are not required. Thus, you only packed the packages that will be required by most of users.

Minimize the Number of JavaScript Files for a ZUL Page

Case Study: ZK Sanbox

In index.zul [1] of ZK Sanbox [2] there are about 15 JavaScript files that will be initially loaded:

```
* http://www.zkoss.org/zksandbox/zkau/web/947199ea/js/zk.wpd
* http://www.zkoss.org/zksandbox/zkau/web/947199ea/js/zul.lang.wpd
* http://www.zkoss.org/zksandbox/macros/category.js
* http://www.zkoss.org/zksandbox/zkau/web/_zv2010062914/js/zkmax.wpd
* http://www.zkoss.org/zksandbox/zkau/web/_zv2010062914/js/zul.wgt.wpd
* http://www.zkoss.org/zksandbox/zkau/web/_zv2010062914/js/zul.wpd
* http://www.zkoss.org/zksandbox/zkau/web/_zv2010062914/js/zul.utl.wpd
* http://www.zkoss.org/zksandbox/zkau/web/_zv2010062914/js/zul.layout.wpd
* http://www.zkoss.org/zksandbox/zkau/web/_zv2010062914/js/zul.wnd.wpd
* http://www.zkoss.org/zksandbox/zkau/web/_zv2010062914/js/zul.tab.wpd
* http://www.zkoss.org/zksandbox/zkau/web/_zv2010062914/js/zul.inp.wpd
* http://www.zkoss.org/zksandbox/zkau/web/_zv2010062914/js/zul.box.wpd
* http://www.zkoss.org/zksandbox/zkau/web/_zv2010062914/js/zul.sel.wpd
* http://www.zkoss.org/zksandbox/zkau/web/_zv2010062914/js/zk.fmt.wpd
* http://www.zkoss.org/zksandbox/zkau/web/_zv2010062914/js/zul.mesh.wpd
```

This means that the browser will trigger 15 requests to load the 15 JavaScript files. Even if each file is not too big, it still takes more time to connect to the server and download it. However, we can specify a DSP file to include several JavaScript into one and declare it at the top of the index.zul.

For example, **/macros/zksandbox.js.dsp**

<syntax lang="xml"> <%@ page contentType="text/javascript;charset=UTF-8" %> <%@ taglib uri="http://www.zkoss.org/dsp/web/core" prefix="c" %> <%@ taglib uri="http://www.zkoss.org/dsp/zk/core" prefix="z" %> ${z:setCWRCacheControl()} <c:include page="~./js/zk.fmt.wpd"/> <c:include page="~./js/zul.mesh.wpd"/> <c:include page="~./js/zul.utl.wpd"/> <c:include page="~./js/zul.layout.wpd"/> <c:include page="~./js/zul.wnd.wpd"/> <c:include page="~./js/zul.tab.wpd"/> <c:include page="~./js/zul.inp.wpd"/> <c:include page="~./js/zul.box.wpd"/> <c:include page="~./js/zul.sel.wpd"/> <c:include page="/macros/category.js"/> </syntax>

Note:

1. The included JavaScript files have their own sequence, so you cannot place them in randomly.
2. The *zk.wpd* is a ZK core JavaScript file hence you don't need to include it
3. The *zul.lang.wpd* is an I18N message, so you don't need to include it.
4. In ZK 5.0.4 we introduced a new feature(#System-wide_Minimizing_the_Number_of_JavaScript_Files). However, since the release of this new feature the packages **zul, zul.wgt, and zkmax** will be merged automatically into the ZK package, so you don't specify them in the the *zksandbox.js.dsp* file.
5. int) DspFns.setCacheControl(java.lang.String, int) [3] is used to set the Cache-Control and Expires headers to 24 hours, so the JavaScript file will be cached for a day.

index.zul <syntax lang="xml"> <?script type="text/javascript" src="/macros/zksandbox.js.dsp"?> // omitted </syntax>

System-wide Minimizing the Number of JavaScript Files

```
[since 5.0.4]
```

If a package is used by all your pages, you could configure it system wide by specifying the packages in the language add-on. Please refer to ZK Configuration Reference/zk.xml/The language-config Element for how to specify a language add-on.

For example, if the zul.wnd package (Window [4]) is required for all pages, then you could add the following to the language add-on.

<syntax lang="xml"> <javascript package="zul.wnd" merge="true"/> </syntax>

Notice that you have to specify the merge attribute which indicates that the JavaScript code of the package will be loaded with the `zk` package. In other words, the `~./js/zk.wpd` will contain all the packages specified with the merge attribute.

Also notice that if you use several DSP/JSP file to load multiple packages in a file as described in the previous section, you generally don't specify them here. Otherwise, you will load the same package twice (though it is safe, it wastes time).

Note: if you merge several JavaScript file into your own lang-addon.xml, but some JavaScript files need to be counted on *zul.lang.wpd*, such as the package *zul.inp*, thus, you can't include this package into your lang-addon.xml.(it will be fixed in ZK 5.0.5+)

Turn Off the Merging of JavaScript Packages

As described above, both `zul` and `zul.wgt` packages are merged into the `zk` package. If you prefer to load them separately, you could disable it by specifying the ondemand attribute as follows.

<syntax lang="xml"> <javascript package="zul" ondemand="true"/> <javascript package="zul.wgt" ondemand="true"/> </syntax>

Notice that all packages are default to load-on-demand, you rarely need to specify the ondemand attribute, unless you want to undo the package that has been specified with the `merge` attribute.

Version History

Version	Date	Content

References

[1] http://zk1.svn.sourceforge.net/viewvc/zk1/releases/5.0.7/zksandbox/src/archive/index.zul?view=log
[2] http://www.zkoss.org/zksandbox
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/fn/DspFns.html#setCacheControl(java.lang.String,
[4] http://www.zkoss.org/javadoc/latest/jsdoc/zul/wnd/Window.html#

Load JavaScript and CSS from Server Nearby

If some of the client machines are far away from the application server, we could set up a server nearby the clients to host ZK's JavaScript and CSS files, and then configure the application server to generate the URLs of JavaScript and CSS (and iamges it refers) from the the sever nearby the clients.

***Notice :** the ZK static resource server is a simple server which **deploy official ZK library**, not your whole application.

How to

1. Implement the URLEncoder [1]
2. Add **library-property** configuration to the *zk.xml*

   Document : ZK Configuration Reference/zk.xml/The Library Properties/org.zkoss.web.servlet.http.URLEncoder.
3. Host ZK static resouce server

Following is a sample :

Configuration

```
<library-property>
    <name>org.zkoss.web.servlet.http.URLEncoder</name>
    <value>org.zkoss.test.TestEncoder</value> <!-- Where the Implementation Class is -->
</library-property>
```

Implementation

```
package org.zkoss.test;
import javax.servlet.ServletContext;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import org.zkoss.web.servlet.http.Encodes.URLEncoder;
public class TestEncoder implements URLEncoder {

      @Override
      public String encodeURL(ServletContext ctx, ServletRequest
request, ServletResponse response,
      String uri, URLEncoder defaultEncoder) throws Exception {
            if (isStaticResource(uri)) {
                  return getResourceHost() + uri.replace("~./", "");
            } else {
                  return defaultEncoder.encodeURL(ctx, request,
response, uri, defaultEncoder);
            }
      }
      /**
       * file .wcs : CSS File
       * file .wpd : Javscript File
       */
      private boolean isStaticResource(String url) {
            return url.startsWith("~./") && (url.endsWith(".wpd") ||
url.endsWith(".wcs"));
      }

      /**
       * Detect where the ip is/ who is login / what kind of resouce
server will
       *
```

```
        * @return the host name include protocol prefix. (Client will
retrieve resource from it)
        */
       private String getResourceHost() {
             return "http://SomeWhereNearbyMe/DefaultContext/zkau/web/";
       }

}
```

Hosting ZK Static Resource

Simply deploy **ZK Library** to a server (near your customer) and add the URL to your implementation of URLEncoder.

Don't know how to deploy on server ? Please refer to Installation Guide.

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/web/servlet/http/Encodes/URLEncoder.html#

Specify Stubonly for Client-only Components

Overview

```
[since 5.0.4][ZK EE]
```

It is common that the states of some components are not required to maintain on the server. A typical example is that an application might use some components, such as `hbox`, for layout and won't access it again after rendered. To minimize the memory footprint, ZK supports a special property called `stubonly` (Component.setStubonly(java.lang.String) [1]). Once specified with `true`, its states won't be maintained on the server (and all states are maintained at the client). For example,

<syntax lang="xml"> <hbox stubonly="true"> </hbox> </syntax>

- Notice this feature is available since ZK 5.0.4 EE.

Values of Stubonly: true, false and inherit

The default value of the `stubonly` property is `inherit`. It means the value is the same as its parent's, if any, or `false`, if no parent at all. Thus, if a component's `stubonly` is specified with `true`, all its descendants are stub-only too, unless `false` is specified explicitly. For example, in the following snippet, only `textbox` is *not* stub-only, while `hbox`, `splitter`, `listbox`, `listitem` and labels are all stub-only.

<syntax lang="xml"> <hbox stubonly="true">

```
a stub-only label
<textbox stubonly="false"/>
<splitter/>
<listbox>
  <listitem label="also stubonly"/>
</listbox>
```

</hbox> </syntax>

Limitation of Stub Components

When a component is stub only, it will be replaced with a special component called a stub component (StubComponent [2]) after rendered. In addition, the adjacent stub components might be merged to minimize the memory further. Thus, the application should not access the component again on the server, if it is specified as stub only.

Invalidation

While a stub component cannot be invalidated directly, it is safe to invalidate its parent. ZK will rerender all non-stub components and retain the states of stub components at the client. For example, in the following snippet, it is safe to click the `invalidate` button. From an end user's point of view, there is no difference whether `stubonly` is specified or not.

<syntax lang="xml">

```
<window>
 <button label="self.parent.invalidate()"/>
 <vbox stubonly="true">
 stubonly <textbox/>
 </vbox>
```

</window> </syntax>

Event Handling

ZK will preserve all registered event listeners and handlers, when converting a stub-only component to a stub component. In other words, the listener will be called if the corresponding event is fired. However, since the original component no longer exists, the event is fired in the most generic format: an instance of Event [3], rather than a derived class.

For example, in the following snippet, "org.zkoss.zk.ui.event.Event:onChange" will be generated to `System.out`.

<syntax lang="xml"> <textbox stubonly="true" onChange='System.out.println(event.getClass().getName()+":"+event.getName())'/> </syntax>

In addition, the target (Event.getTarget() [4]) is the stub component rather than the original one (`text`).

Client-side Programming

The client-side widget of a component is the same no matter if it is stub only. Thus, the application can have the full control by registering the client side event listener, such as

<syntax lang="xml"> <textbox stubonly="true" w:onChange="doSomething(this.value)" xmlns:w="client"/> </syntax>

In other words, the stub-only components behave the same at the client.

Refer to Client Side Programming and ZK Client-side Reference: General Control for more information.

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#setStubonly(java.lang.String)
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/StubComponent.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Event.html#
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/Event.html#getTarget()

Reuse Desktops

```
[Since 5.0.0]
```

By default, a desktop is purged when the user browses to another URI or refreshes the page. Thus, the user can have the most updated information. However, if a page takes too long to generate, you can provide a plugin so-called *desktop recycle*.

First, you implement the DesktopRecycle [1] interface to cache and reuse the desktops which are supposedly being removed. Second, specify the class in `WEB-INF/zk.xml`. For example, let us assume the class you implement is called `foo.MyRecycle`, then add the following to `zk.xml`

<syntax lang="xml"> <listener> <listener-class>foo.MyRecycle</listener-class> </listener> </syntax>

org.zkoss.zkmax.zk.ui.util.DesktopRecycle

```
[Enterprise Edition]
[Since 5.0.0]
```

ZK provides a default implementation, the DesktopRecycle [2] class, to simplify the use. You can use it directly or extends from it. By default, it caches all desktops for all URI. You can extend it to limit to certain paths by overriding the `shallRecycle` method, or not to use desktops older than particular time by overriding the `shallReuse` method.

For example, we can limit the URL to cache to "/long-op/*", and re-generate the page if it has been served for more than 5 minutes.

<syntax lang="java"> public class MyRecycle extends org.zkoss.zkmax.zk.ui.util.DesktopRecycle {

```
 protected boolean shallCache(Desktop desktop, String path, int cause) {
   return path.startsWith("/long-op");
 }
 protected boolean shallReuse(Desktop desktop, String path, int secElapsed) {
   return secElapsed >= 300;
 }
```

} </syntax>

Implement Your Own Desktop Recycle

```
[Since 5.0.0]
```

It is straightforward to implement the DesktopRecycle [1] interface from scratch, if you prefer. The basic idea is to cache the desktop when the `beforeRemove` method is invoked, and to reuse the cached desktop when the `beforeService` method is called.

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/DesktopRecycle.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkmax/ui/util/DesktopRecycle.html#

Miscellaneous

Button: use the `os` mold if there are a lot of buttons

The `trendy` mold of a button provides a better and consistent look for, especially, Internet Explorer 6. Unfortunately, the browser (particularly, Internet Explorer) will be slowed down if there are a lot of button (with `trendy`) in the same page.

```
Notice that the default mold is os in ZK 5, while trendy in ZK 3.6.
```

The default mold can be changed easily. For example,

<syntax lang="xml"> <library-property> <name>org.zkoss.zul.Button.mold</name> <value>trendy</value> </library-property> </syntax>

Refer to ZK Configuration Reference for more information.

Prolong the Period to Check Whether a File Is Modified

ZK caches the parsed result of a ZUML page and re-compiles it only if it is modified. In a production system, ZUML pages are rarely modified so you can prolong the period to check whether a page is modified by specifying `file-check-period` in `WEB-INF/zk.xml` as shown below. By default, it is 5 seconds.

<syntax lang="java" > <desktop-config>

```
   <file-check-period>600</file-check-period>
```

</desktop-config> </syntax>

Version History

Version	Date	Content

Security Tips

This chapter describes how to make ZK applications secure. ZK is designed to provide enterprise-grade security. However, there are still several discussions worth to take a look.

Cross-site scripting

Overview

Cross-site scripting [1] (XSS) is a type of computer security vulnerability typically found in web applications that enables malicious attackers to inject client-side script into web pages viewed by other users. Because HTML documents have a flat, serial structure that mixes control statements, formatting, and the actual content, any non-validated user-supplied data included in the resulting page without proper HTML encoding may lead to markup injection.

To prevent from XSS attack, ZK component encodes any value that might be input by an user, such as the value of label and textbox, by escaping & and other unsafe characters. For example, the following statement is totally safe no matter what the value of any_value might be:

```
<textbox value="${any_value}"/>
```

However, there are still some notes worth to pay attention to.

The content Property of html and comboitem

The content property of the html and combitem components (Html.setContent(java.lang.String) [1] and Comboitem.setContent(java.lang.String) [2]) are designed to allow applications to generate HTML content directly. In other words, it is not encoded. Thus, it is better to carry the value input by an user, unless it is encoded property. For example, if the value of any_content is, in the following example, generated directly and vulnerable to XSS attack if it is the value provided by an user and without proper encoding.

```
<html>${any_content}</html>
```

- Java API: Html.setContent(java.lang.String) [1] and Comboitem.setContent(java.lang.String) [2]

Client-side Actions

The client-side action is not encoded and the options is interpreted as a JSON object. Thus, you could encode it by yourself, if you allow the end user to specify it (which is generally not suggested at all).

Version History

Version	Date	Content

References

[1] http://en.wikipedia.org/wiki/Cross-site_scripting
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Comboitem.html#setContent(java.lang.String)

Block Request for Inaccessible Widgets

Inaccessible widgets (such as disabled or invisible) can be accessed easily with a debugging tool running at the browser. For example, a hostile user can make an invisible button visible and then click on it to trigger unexpected actions. Thus, it is recommended not to create a widget if it is not supposedly accessible. For example, the first statement is safer than the second one in the following example:

```
<button unless="${accessible}"/>
<button visible="${accessible}"/>
```

Block with `InaccessibleWidgetBlockService`

```
[since 5.0.0]
[Enterprise Edition]
```

If you want to block a request for inaccessible widgets for the whole application or for a particular desktop, you can implement the org.zkoss.zk.au.AuService interface to filter out unwanted requests. ZK Enterprise Edition has provided a simple blocked called InaccessibleWidgetBlockService [1]. To apply it to the whole application, just specify the following in `WEB-INF/zk.xml` as follows.

```
<listener>
      <listener-class>org.zkoss.zkmax.au.InaccessibleWidgetBlockService$DesktopInit</list
</listener>
```

Then, each time a desktop is created, an instance of `InaccessibleWidgetBlockService` is added to the desktop to block the requests from the inaccessible widgets.

In many cases, you just want to block particular events, not all events. For example, you want to receive `onOpen` when a `menupopup` is going to show up. Then, you can specify a library property called events [2] to control the behavior of [%s %s]. For example,

```
<library-property>
      <name>org.zkoss.zkmax.au.IWBS.events</name>
      <value>onClick,onChange,onSelect</value>
</library-property>
```

Implement Your Own Block

The implementation of `AuService` is straightforward. For example, the following example blocks only `button` and `onClick`:

```
public class MyBlockService implements org.zkoss.zk.au.AuService {
      public boolean service(AuRequest request, boolean everError) {
            final Component comp = request.getComponent();
            return (comp instanceof Button) &&
"onClick".equals(request.getCommand());
                  //true means block
      }
}
```

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkmax/au/InaccessibleWidgetBlockService.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/au/IWBS/events.html#

Performance Monitoring

To improve the performance of an Ajax application, it is better to monitor the performance for identifying the bottleneck. Depending on the information you'd like to know, there are a few approaches.

- PerformanceMeter [1]: Monitoring the performance from the network speed, server-processing time and the client-rendering time.
- EventInterceptor [2]: Monitoring the performance of each event listener.
- Monitor [3]: Monitoring the number of desktops, sessions and other system load.
- There are a lot of performance monitor tools, such as VisualVM [4] and JProfiler [5]. They could provide more insightful view of your application.

For sample implementations, you might take a look at the following articles:

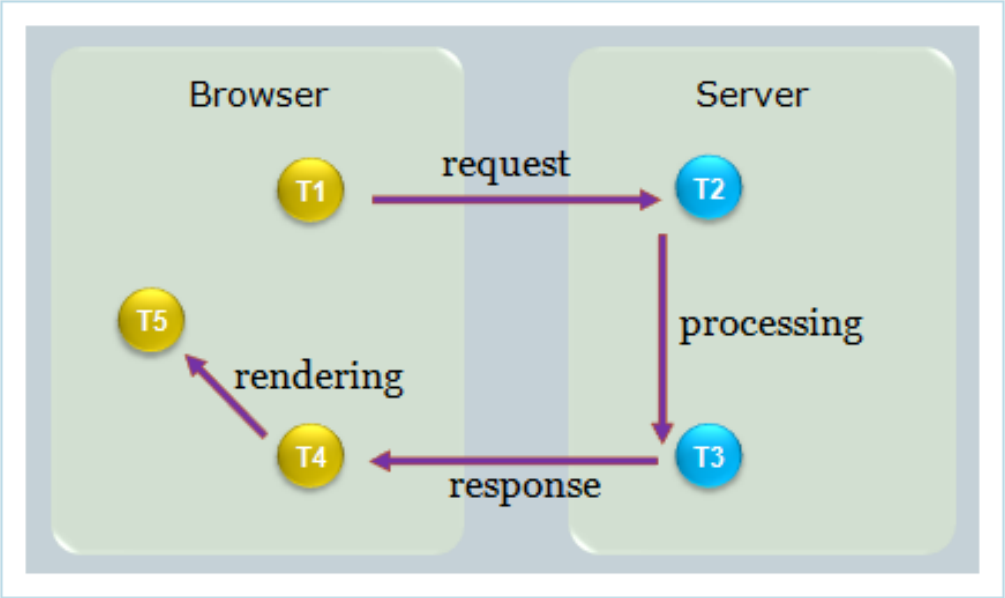
- Performance Monitoring of ZK Applicaiton
- A ZK Performance Monitor
- Real-time Performance Monitoring of Ajax Event Handlers

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/PerformanceMeter.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/EventInterceptor.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Monitor.html#
[4] http://visualvm.dev.java.net/
[5] http://www.ej-technologies.com/products/jprofiler/overview.html

Performance Meters

PerformanceMeter [1] is a collection of callbacks that the implementation could know when a request is sent, arrives or is is processed.



As show above, T1-T5 identifies the following callbacks.

- T1: (java.lang.String, org.zkoss.zk.ui.Execution, long) PerformanceMeter.requestStartAtClient (java.lang.String, org.zkoss.zk.ui.Execution, long) [1]
- T2: org.zkoss.zk.ui.Execution, long) PerformanceMeter.requestStartAtServer(java.lang.String, org.zkoss.zk.ui.Execution, long) [2]
- T3: org.zkoss.zk.ui.Execution, long) PerformanceMeter.requestCompleteAtServer(java.lang.String, org.zkoss.zk.ui.Execution, long) [3]
- T4: org.zkoss.zk.ui.Execution, long) PerformanceMeter.requestReceiveAtClient(java.lang.String, org.zkoss.zk.ui.Execution, long) [4]
- T5: org.zkoss.zk.ui.Execution, long) PerformanceMeter.requestCompleteAtClient(java.lang.String, org.zkoss.zk.ui.Execution, long) [5]

Thus,

- Server Execution Time: T3 - T2
- Client Execution Time: T5 - T4
- Network Latency Time: (T4 - T3) + (T2 - T1)

Notice that, when we make a connection to load a page for the first time, only Server Execution Time is available. T4 and T5 will be saved on the client side, and sent back along with the next request.

Once implemented, you could register it by specifying the following in `WEB-INF/zk.xml` (assume the class is called foo.MyMeter):

```
<zk>
    <listener>
        <listener-class>foo.MyMeter</listener-class>
    </listener>
</zk>
```

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/PerformanceMeter.html#requestStartAtClient
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/PerformanceMeter.html#requestStartAtServer(java.lang.String,
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/PerformanceMeter.html#requestCompleteAtServer(java.lang.String,
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/PerformanceMeter.html#requestReceiveAtClient(java.lang.String,
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/PerformanceMeter.html#requestCompleteAtClient(java.lang.String,

Event Interceptors

Though EventInterceptor [2] is designed to allow developer to intercept how an event is processed, you could use it as callback to know how long it takes to process an event. The event processing time can be calculated by subtracting the time between EventInterceptor.beforeProcessEvent(org.zkoss.zk.ui.event.Event) [1] and EventInterceptor.afterProcessEvent(org.zkoss.zk.ui.event.Event) [2]

Once implemented, you could register it by specifying the following in `WEB-INF/zk.xml` (assume the class is called foo.MyEventMeter):

```
<zk>
    <listener>
        <listener-class>foo.MyEventMeter</listener-class>
    </listener>
</zk>
```

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/EventInterceptor.html#beforeProcessEvent(org.zkoss.zk.ui.event.Event)
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/EventInterceptor.html#afterProcessEvent(org.zkoss.zk.ui.event.Event)

Loading Monitors

To know the loading of an application, you could implement Monitor [3] to count the number of desktops, sessions and requests.

Once implemented, you could register it by specifying the following in `WEB-INF/zk.xml` (assume the class is called foo.MyStatistic):

```
<zk>
    <listener>
        <listener-class>foo.MyStatistic</listener-class>
    </listener>
</zk>
```

Version History

Version	Date	Content

Testing

ZK is a Java framework. Technically you could use any Java test tools you prefer. Here we describe the testing tips and ZTL (the official test tool based on Selenium).

For information of particular test tools, please refer to small talks:

- Sahi: Making ZK Functional Tests With Sahi
- Selenium: How to Test ZK Application with Selenium and ZK Unit Testing
- zunit: ZK Unit Testing Project - zunit

Testing Tips

ID and UUID

By default, the desktop's ID and component's UUID are randomized, such that the chance to pick up a wrong component is minimized if the sever is restarted. Since component's UUID will become DOM element's ID at the browser, it means the DOM element's IDs will change from one test run to another.

If your test code runs at the server (such jUnit), it is not an issue at all (since DOM elements are available at the client only). However, if your test tool runs at the browser, you have to resolve it with one of the following solutions:

1. Not to depend on DOM element's ID. Rather, use component's ID and/or component's parent-child-sibling relationship.
2. Implement IdGenerator [1] to generate UUID in a predictable and repeatable way

Approach 1: Use Widget's ID

With Server+client architecture, ZK maintains an *identical* world at the client. If your test tool is able to access JavaScript at the client, your test code could depend on the widget's ID and widget's parent-child-relationship as your application code depends on the component's ID and component's parent-child-relationship. They are *identical*, except one is JavaScript and called Widget [2], while the other is Java and called Component [1].

This is a suggested approach, since it is much easier to test an application in the same abstract level -- the component level, aka., the widget level (rather than DOM level).

To retrieve widgets at the client, you could use jq [3] and/or _global_.Map) Widget.$(zk.Object, _global_.Map) [4] (they are all client-side API). jq [3] allows your test code to access the components directly, so the test code could depend on widget's ID (Widget.id [5]) and the widget tree (Widget.firstChild [6], Widget.nextSibling [7] and so on).

```
jq('@window[border="normal"]') //returns a list of window whose border
is normal
jq('$x'); //returns the widget whose ID is x
jq('$x $y'); //returns the widget whose ID is y and it is in an ID
space owned by x
```

With this approach, you still can verify the DOM structure if you want, since it can be retrieved from widget's Widget.$n() [8].

ZTL [9] is a typical example that takes this approach. For more information, please refer to the ZTL section.

Approach 2: Implement ID Generator

If your test tool running at the client cannot access JavaScript, you could implement an ID generator to generate desktop's ID and component's UUID in a predictable and repeatable matter.

To implement a custom ID generator, you have to do the following:

- Implement a Java class that implements IdGenerator [1].
- Specify the Java class in WEB-INF/zk.xml with the id-generator-class element. For example,

```
<system-config>
    <id-generator-class>my.IdGenerator</id-generator-class>
</system-config>
```

Different Configuration for Different Environment

If you prefer to have a different configuration for the testing environment (such as specifying ID generator for testing), you could put the configuration in a separated file, say, `WEB-INF/config/zk-testing.xml` with the following content.

```
<zk>
  <system-config>
    <id-generator-class>my.IdGenerator</id-generator-class>
  </system-config>
</zk>
```

Then, you could you could specify -Dorg.zkoss.zk.config.path=/WEB-INF/config/zk-testing.xml as one of the arguments when starting the Web server.

Disabled UUID recycle

If you want to generate uuid with some conditional , you might also want to disable UUID recycle. (It will reuse all the uuids from removed components.) You could set the properties org.zkoss.zk.ui.uuidRecycle.disabled in zk.xml .

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/sys/IdGenerator.html#
[2] http://www.zkoss.org/javadoc/latest/jsdoc/zk/Widget.html#
[3] http://www.zkoss.org/javadoc/latest/jsdoc/_global_/jq.html#
[4] http://www.zkoss.org/javadoc/latest/jsdoc/zk/Widget.html#$(zk.Object,
[5] http://www.zkoss.org/javadoc/latest/jsdoc/zk/Widget.html#id
[6] http://www.zkoss.org/javadoc/latest/jsdoc/zk/Widget.html#firstChild
[7] http://www.zkoss.org/javadoc/latest/jsdoc/zk/Widget.html#nextSibling
[8] http://www.zkoss.org/javadoc/latest/jsdoc/zk/Widget.html#$n()

ZTL

ZTL [9] is an automatic testing tool based on jUnit [1] and Selenium [2]. It is supported officially by ZK team. Here is a short introduction. For more information, please refer to ZTL Documentation [3].

The ZTL language is XML based which describes the operation of a test case for the Selenium Remote Control (RC) [4]. For example,

```
<test tags="button">
    <case id="Click">
        <server><!--
            <zk>
                <button id="btn" label="Click Me to Show a Message" onClick='alert("Hello
            </zk>
        --></server>
        <client>
             click(btn);
             waitResponse();
             verifyTrue(jq("@window").exists());
        </client>
    </case>
</test>
```

The root tag of the ZTL file is test and encloses one or many test case(s) (similar to each method of the jUnit test case). The case can enclose one or many server and client tags. The content of the server is run on the ZK server, and the content of the client is run on the Selenium server (we called client).

Notice that the client code is Java, and it runs on the Selenium server that provides an 'emulated' browser environment in Java. In other words, the client code will be eventually 'converted' to JavaScript code, and then delivered to the browser for execution. Also notice that the Selenium server is a server independent of the application server. It is used to run the client code (without changing anything run on the application server).

In the above example, the content of the server creates a button (`id=btn`) which when clicked on will show a "Hello!" message. The content of the client uses `btn` which is an instance of org.zkoss.ztl.Widget [5], to fire a Click event to the browser, and then waits for the response from ZK server. Once the response is received the code then checks whether `@window` exists or not by using the jq API which is implemented by the class org.zkoss.ztl.JQuery [6].

Unit testing using image differences

In additions to Java, ZTL also allows developers to do unit testing by comparing the screenshot.

For more information, please refer to Vision Test for ZTL [7].

Version History

Version	Date	Content

References

[1] http://www.junit.org
[2] http://seleniumhq.org/
[3] http://code.google.com/p/zk-ztl/#Documentation
[4] http://seleniumhq.org/projects/remote-control/
[5] http://zk-ztl.googlecode.com/svn/trunk/javadoc/org/zkoss/ztl/Widget.html
[6] http://zk-ztl.googlecode.com/svn/trunk/javadoc/org/zkoss/ztl/JQuery.html
[7] http://blog.zkoss.org/index.php/2011/03/22/vision-test-for-ztl/

Customization

Here describes how to customize ZK, such as initializing components with different properties, loading ZUML document from database and so on.

Packing Code

There are two ways to pack the customization code: part of the Web application, or an independent JAR file. Packing as part of the Web application is straightforward. All you have to do is to specify the customization in `WEB-INF/zk.xml` as described in ZK Configuration Reference.

In many cases, it is better to pack the customization code as an independent JAR file, such that it can be managed separately and reused in multiple Web applications.

Where to Configure a JAR File

The configuration of a JAR file can be placed in a file called `config.xml`, and it must be placed under `/metainfo/zk`. If the JAR file also provides the component definitions, you have to prepare another fiile called `lang-addon.xml` under the same directory[1] .

The content of `/metainfo/zk/config.xml` is similar to `WEB-INF/zk.xml`, except only a subset of configurations is allowed. Here is a sample (zkex.jar 's config.xml)[2] :

```
<config>
	<config-name>zkex</config-name><!-- used to resolve dependency -->
	<depends>zk</depends>

	<version>
		<version-class>org.zkoss.zkex.Version</version-class>
		<version-uid>5.0.6</version-uid>
```

```
            <zk-version>5.0.0</zk-version><!-- or later -->
      </version>

      <listener>
            <listener-class>org.zkoss.zkex.init.WebAppInit</listener-class>
      </listener>

      <library-property>
            <name>org.zkoss.zul.chart.engine.class</name>
            <value>org.zkoss.zkex.zul.impl.JFreeChartEngine</value>
      </library-property>
      <library-property>
            <name>org.zkoss.zul.captcha.engine.class</name>
            <value>org.zkoss.zkex.zul.impl.JHLabsCaptchaEngine</value>
      </library-property>
</config>
```

[1] For more information, please refer to ZK Client-side Reference: Language Definition.

[2] For more information, please refer to ZK Configuration Reference: JAR File's config.xml.

How to Initialize a JAR File

Sometimes you have to initialize a JAR file. It can be done by implementing WebAppInit (http://www.zkoss.org/javadoc/ latest/ zk/ org/ zkoss/ zk/ ui/ util/ WebAppInit. html#), and then specifying it as a listener in `/metainfo/zk/config.xml`. For example,

```
public class MyJARInit implements WebAppInit {
    public void init(WebApp wapp) throws Exception {
        //do whatever init you need
    }
}
```

Notice that many configuration can be done by accessing Configuration (http://www.zkoss.org/javadoc/latest/zk/org/ zkoss/ zk/ ui/ util/ Configuration. html#) directly. If you want to access it in WebAppInit.init(org.zkoss.zk.ui.WebApp) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zk/ ui/ util/ WebAppInit.html#init(org.zkoss.zk.ui.WebApp)) by invoking WebApp.getConfiguration() (http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/WebApp.html#getConfiguration()) as follows.

```
    public void init(WebApp wapp) throws Exception {
        Configuration config = wapp.getConfiguration();
...
```

Version History

Version	Date	Content

Component Properties

With component definitions, we could specify the initial values for the properties, attributes and annotations of a component.

Properties

Depending on the requirement, you could change the initial value of a property for a particular ZUML document or for the whole application.

Notice that the initial values are applicable only to the component instantiated by ZK Loaders. It has no effect if you instantiate it in pure Java (unless you invoke Component.applyProperties() [4] after instantiating a component).

Page-wide Initialization

Suppose we want to assign `normal` to the border property (Window.setBorder(java.lang.String) [1]) of all windows in a ZUML document, then we could use the component directive as follows.

```
<?component name="window" extends="window" border="normal"?>
<window title="Border"/>
```

Application-wide Initialization

If you prefer to have the same initial value for all ZUML documents, you could specify it in a language addon. For example, we could prepare a file called `WEB-INF/lang-addon.xml` with the following content:

```
<language-addon>
	<addon-name>myapp</addon-name>
	<component>
		<component-name>window</component-name>
		<extends>window</extends>
		<property>
			<property-name>border</property-name>
			<property-value>normal</property-value>
		</property>
	</component>
</language-addon>
```

Then, we could specify this file by adding the following content to `WEB-INF/zk.xml`:

```
	<language-config>
		<addon-uri>/WEB-INF/lang-addon.xml</addon-uri>
	</language-config>
```

For more information, please refer to ZK Configuration Reference.

Molds

A mold is yet another property (Component.setMold(java.lang.String) [1]), so you could change the initial value as described in the previous section. However, since it is common to change the value, we allow developers to specify the mold for a given component in a library property. As shown, the library is named as *ClassName*.mold. For example, if you would like to specify trendy as the initial mold of a button, then you could add the following to WEB-INF/zk.xml:

```
	<library-property>
		<name>org.zkoss.zul.Button.mold</name>
		<value>trendy</value>
	</library-property>
```

Attributes

Like properties, you could assign the initial value of an attribute for a given component in the whole application.

Notice that the initial values are applicable only to the component instantiated by ZK Loaders. It has no effect if you instantiate it in pure Java (unless you invoke Component.applyProperties() [4] after instantiating a component).

Page-wide Initialization

Unlike the initial value of a property, there is no way to specify the initial value of a custom attribute in a ZUML document.

Application-wide Initialization

Similar to customizing the initial value of a property, you could specify the following in a language addon to assign an initial value of a attribute to a component.

```
<language-addon>
	<addon-name>myapp</addon-name>
	<component>
		<component-name>panel</component-name>
		<extends>panel</extends>
		<custom-attribute>
			<attribute-name>any.attribute</attribute-name>
			<attribute-value>any value</attribute-value>
		</custom-attribute>
	</component>
</language-addon>
```

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/Window.html#setBorder(java.lang.String)

UI Factory

UiFactory [1] is used to instantiate all UI objects, such as session, desktop, and components, and to load ZUML documents. You could customize it to provide the functionality you want.

For example, SerializableUiFactory [1] is the factory used to instantiate sessions that are serializable[1] , while SimpleUiFactory [2], the default factory, instantiates non-serializable sessions.

Here are a list of customization you could do with UI Factory:

- Load a ZUML document from, say, a database
  - It can be done by overriding java.lang.String) UiFactory.getPageDefinition(org.zkoss.zk.ui.sys.RequestInfo, java.lang.String) [3]
- Instantiate a component by using a different implementation
  - It can be done by overriding org.zkoss.zk.ui.Component, org.zkoss.zk.ui.metainfo.ComponentInfo) UiFactory.newComponent(org.zkoss.zk.ui.Page, org.zkoss.zk.ui.Component, org.zkoss.zk.ui.metainfo.ComponentInfo) [4] and org.zkoss.zk.ui.Component, org.zkoss.zk.ui.metainfo.ComponentInfo, java.lang.String) UiFactory.newComponent(org.zkoss.zk.ui.Page, org.zkoss.zk.ui.Component, org.zkoss.zk.ui.metainfo.ComponentInfo, java.lang.String) [4].
  - Instantiate a desktop by using a different implementation
  - It can be done by overriding java.lang.String, java.lang.String) UiFactory.newDesktop(org.zkoss.zk.ui.sys.RequestInfo, java.lang.String, java.lang.String) [5]
  - Instantiate a page by using a different implementation
  - It can be done by overriding org.zkoss.zk.ui.metainfo.PageDefinition, java.lang.String) UiFactory.newPage(org.zkoss.zk.ui.sys.RequestInfo, org.zkoss.zk.ui.metainfo.PageDefinition, java.lang.String) [6] and/or org.zkoss.zk.ui.Richlet, java.lang.String) UiFactory.newPage(org.zkoss.zk.ui.sys.RequestInfo, org.zkoss.zk.ui.Richlet, java.lang.String) [6]

Notice that it is suggested to extend from either SerializableUiFactory [1] or SimpleUiFactory [2], rather than to implement UiFactory [1] from scratch.

[1] Then, the application is able to run in a clustering environment. Fore more information, please refer to the Clustering section
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/http/SimpleUiFactory.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/sys/UiFactory.html#getPageDefinition(org.zkoss.zk.ui.sys.RequestInfo,
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/sys/UiFactory.html#newComponent(org.zkoss.zk.ui.Page,
[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/sys/UiFactory.html#newDesktop(org.zkoss.zk.ui.sys.RequestInfo,
[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/sys/UiFactory.html#newPage(org.zkoss.zk.ui.sys.RequestInfo,

Load ZUML from Database

The default implementation of java.lang.String) AbstractUiFactory.getPageDefinition(org.zkoss.zk.ui.sys.RequestInfo, java.lang.String) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zk/ ui/ impl/ AbstractUiFactory. html#getPageDefinition(org. zkoss. zk. ui. sys. RequestInfo,) loads the ZUML document from the Web application's resources (i.e., the files found in a Web application). If you prefer to load from other sources, such as a database, you could override it.

The pseudo code will look like the following:

```
public class MyUiFactory extends SimpleUiFactory {
    @Override
    public PageDefinition getPageDefinition(RequestInfo ri, String
path) {
        PageDefinition pgdef = getFromCache(path); //your cache
implementation
        if (pgdef == null) {
            String content = loadFromDatabase(path); //your resource
loading
            pgdef = getPageDefinition(ri, content, "zul"); //delegate
to SimpleUiFactory
            setCache(path, pgdef); //cache the result
        }
        return pgdef;
    }
}
```

where we assume you implemented `loadFromDatabase` to load the ZUML document from a database. In addition, you have to implement `getFromCache` and `setCache` to cache the result in order to improve the performance of retrieving the document from the database.

On the other hand, the parsing of the ZUML document can be done easily by calling java.lang.String, java.lang.String) AbstractUiFactory.getPageDefinitionDirectly(org.zkoss.zk.ui.sys.RequestInfo, java.lang.String, java.lang.String) (http:/ / www. zkoss. org/ javadoc/ latest/ zk/ org/ zkoss/ zk/ ui/ impl/ AbstractUiFactory. html#getPageDefinitionDirectly(org.zkoss.zk.ui.sys.RequestInfo,).

Version History

Version	Date	Content

Init and Cleanup

You could have some custom initialization and cleanup when an application, a session, a desktop or an execution is instantiated or about to being destroyed.

There are two steps:

1. Implements the corresponding interface. For example, WebAppInit [1] for application's initialization
2. Register it in `WEB-INF/zk.xml`, or in Java.

Interfaces

Task	Interface
Application Init	WebAppInit [1]
Application Cleanup	WebAppCleanup [2]
Session Init	SessionInit [3]
Session Cleanup	SessionCleanup [4]
Desktop Init	DesktopInit [5]
Desktop Cleanup	DesktopCleanup [3]
Execution Init	ExecutionInit [6]
Execution Cleanup	ExecutionCleanup [3]

Notice that ZK will instantiate an object from the class you registered for each callback. For example, an object is instantiated to invoke java.lnag.Object) DesktopInit.init(org.zkoss.zk.ui.Desktop, java.lnag.Object) [7], and another object instantiated to invoke DesktopCleanup.cleanup(org.zkoss.zk.ui.Desktop) [8], even if you register a class that implements both DesktopInit [5] and DesktopCleanup [3].

If you have something that is initialized in the init callback and have to clean it up in the cleanup callback, you cannot store it as a data member. Rather, you have to maintain it by yourself, such as storing it in the desktop's attributes (java.lang.Object) Desktop.setAttribute(java.lang.String, java.lang.Object) [9]), session's attributes or application's attributes.

Registration

The registration in `WEB-INF/zk.xml` is the same, no matter what interface you implement:

```
<listener>
    <listener-class>my.MyImplementation</listener-class>
</listener>
```

The registration in Java is done by Configuration.addListener(java.lang.Class) [10].

```
webapp.getConfiguration().addListener(my.MyImplementation.class);
```

Version History

Version	Date	Content

References

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

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

[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/SessionInit.html#

[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/SessionCleanup.html#

[5] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/DesktopInit.html#

[6] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/ExecutionInit.html#

[7] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/DesktopInit.html#init(org.zkoss.zk.ui.Desktop,

[8] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/DesktopCleanup.html#cleanup(org.zkoss.zk.ui.Desktop)

[9] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Desktop.html#setAttribute(java.lang.String,

[10] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Configuration.html#addListener(java.lang.Class)

AU Services

An AU service (AuService [1]) is a plugin used to intercept the AU requests (AuRequest [2]) sent from the client.

By plugging in an AU service, you could

- Ignore some AU requests (such as hostile requests)
- Change the default way of handling an AU request
- Handle application-specific AU requests

To plug an AU service to a desktop, you could invoke Desktop.addListener(java.lang.Object) [3]. You could plug as many AU services as you want. Once plugged, all AU requests will go through the AU services (unless it was ignored by other AU service).

If you want to plug a particular component, you could invoke Component.setAuService(org.zkoss.zk.au.AuService) [4]. Unlike desktops, a component can have at most one AU service.

If you want to plug an AU service, you could implement DesktopInit [5] and register it in zk.xml as described in the Init and Cleanup section.

```
public class MyDesktopInit implements DesktopInit {
    public void init(Desktop desktop, Object request) {
        desktop.addListener(new MyAuService()); //assume you have an AU
 service called MyAuService
    }
}
```

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/au/AuService.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/au/AuRequest.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/sys/Desktop.html#addListener(java.lang.Object)
[4] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Component.html#setAuService(org.zkoss.zk.au.AuService)

AU Extensions

An AU extension (AuExtension [1]) is a small program that can be plugged into ZK Update Engine (DHtmlUpdateServlet [2]) and extend its functionality. Actually our file upload and multimedia viewing are implemented as an AU extension that you can replace with your implementation.

An AU extension is associated with a name starting with slash, such as "/upload". Then each time a request targeting /zkau/upload will be forwarded to this extension for service.

To register an AU extension, you could specify the name and the class name as the initial parameter of the declaration of ZK Update Engine in `WEB-INF/web.xml`. For more information, please refer to ZK Configuration Reference.

If you want to register it in Java, you could use org.zkoss.zk.au.http.AuExtension) DHtmlUpdateServlet.addAuExtension(java.lang.String, org.zkoss.zk.au.http.AuExtension) [3] instead.

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/au/http/AuExtension.html#
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/au/http/DHtmlUpdateServlet.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/au/http/DHtmlUpdateServlet.html#addAuExtension(java.lang.String,

How to Build ZK Source Code

SVN Repository

Depending on the branch you want, you could check out the source codes from the following paths.

Version	URL	Description
5.0.* (current)	[1]	The 5.0 branch. It is the working repository for the most up-to-date source codes for current ZK 5
6.0.* (upcoming)	[2]	The 6.0 branch. It is the working repository for the most up-to-date source codes for upcoming ZK 6
3.6.* (maintaining)	[3]	The 3.6 branch. It is the working repository for the most up-to-date source codes for maintaining ZK 3.6. Though it is named *trunk*, it is used only for the 3.6 branch now.
3.0.* (maintaining)	[4]	The 3.0 branch. It is the working repository for the most up-to-date source codes for maintaining ZK 3.0
2.4.* (maintaining)	[5]	The 2.4 branch. It is the working repository for the most up-to-date source codes for maintaining ZK 2.4
Releases (frozen)	https://zk1.svn.sourceforge.net/svnroot/zk1/releases/*x.y.z*	The releases. We won't change the code in this repository. The URL depends on the version you want to check out. For a complete list, please visit [6]

Maven Build

Since ZK 5.0.5 release, we put an Eclipse project setting into each submodule folder, such as *zk*, *zul*, *zkdemo*, and so on. After that time, you can be able to check out the source code from the SVN path above as a Eclipse Maven project to develop/build it. Or you can use Maven command line to build the ZK Jar files.

For example,

```
$ svn checkout
https://zk1.svn.sourceforge.net/svnroot/zk1/releases/5.0.5/zul zul
$ cd zul
$ mvn clean package
```

See Also

- ZK Installation Guide: Setting up IDE/Maven

Version History

Version	Date	Content

References

[1] https://zk1.svn.sourceforge.net/svnroot/zk1/branches/5.0/
[2] https://zk1.svn.sourceforge.net/svnroot/zk1/branches/6.0/
[3] https://zk1.svn.sourceforge.net/svnroot/zk1/trunk/
[4] https://zk1.svn.sourceforge.net/svnroot/zk1/branches/3.0/
[5] https://zk1.svn.sourceforge.net/svnroot/zk1/branches/2.4/
[6] http://zk1.svn.sourceforge.net/viewvc/zk1/releases/.

Supporting Utilities

In this section we will discuss the utilities that ZK are built on. You don't need them to develop ZK applications, but you might find them useful if they are applicable. Here we provide the basic information of them. Interested readers might refer to Javadoc [1] for detailed API.

References

[1] http://www.zkoss.org/javadoc/latest/zk/

Logger

In this section we describe how to configure the logging of ZK internal functions. You general can ignore it, unless you'd like to know how ZK operates internally.

Notice that, if you are using Google App Engine, you can *not* configure the logging as described in this chapter. For more information, please refer to Setting up Google App Engine.

How to Configure Logging

ZK uses the standard logger [1] to log messages. You could control what to log by configuring the logging of the Web server you are using. The configuration usually varies from one server to another. However, you could use the configuration mechanism provided by ZK as described in this section. It shall work with most Web servers.

There are basically two steps to configure the standard logger with ZK's configuration mechanism:

1. Prepare a logging configuration file
2. Specify the configuration file in a library property

Prepare a logging configuration file

A logging configuration file is a standard properties file. Each line is a key-value pair in the following format:

```
''a.package.or.a.class'' = ''level''
```

Here is an example of a configuration file.

```
 org.zkoss.zk.ui.impl.UiEngineImpl=FINER
    #Make the log level of the specified class to FINER
 org.zkoss.zk.ui.http=DEBUG
    #Make the log level of the specified package to DEBUG
 org.zkoss.zk.ui=OFF
    #Turn off the log for the specified package
 org.zkoss=WARNING
    #Make all log levels of ZK classes to WARNING except those
specified here
```

Allowed Levels

Level	Description
OFF	Indicates no message at all.
SEVERE	Indicates providing error messages.
WARNING	Indicates providing warning messages. It also implies ERROR.
INFO	Indicates providing informational messages. It also implies ERROR and WARNING.
FINE	Indicates providing tracing information for debugging purpose. It also implies ERROR, WARNING and INFO.
FINER	Indicates providing fairly detailed tracing information for debugging purpose. It also implies ERROR, WARNING, INFO and DEBUG

Specify the handler for Jetty and servers that don't turn on the standard logger

Some Web servers, such as Jetty, don't turn on the standard logger by default. Thus, in the logging configuration file, you have to configure the handler too. For example, you can turn on the java.util.logging.ConsoleHandler to write the logs to the console by adding the following lines to the logging configuration file:

```
handlers = java.util.logging.ConsoleHandler

java.util.logging.ConsoleHandler.level = FINER
java.util.logging.ConsoleHandler.formatter =
java.util.logging.SimpleFormatter
```

Here is another example that configures the console and a file to be the target of the logs:

```
handlers = java.util.logging.FileHandler,
java.util.logging.ConsoleHandler

java.util.logging.ConsoleHandler.level = FINER
java.util.logging.ConsoleHandler.formatter =
java.util.logging.SimpleFormatter

java.util.logging.FileHandler.formatter =
java.util.logging.SimpleFormatter
java.util.logging.FileHandler.pattern = /var/log/jetty6/solr-%u.log
java.util.logging.FileHandler.level = FINER

org.zkoss.zk.ui.impl.UiEngineImpl=FINER
org.zkoss.bind=FINE
```

Specify the configuration file in a library property

To let ZK load the logging configuration file, you have to specify in a library property called org.zkoss.util.logging.config.file. For example,

```
<library-property>
      <name>org.zkoss.util.logging.config.file</name>
      <value>conf/zk-log.properties</value>
</library-property>
```

If a relative path is specified, it will look for the class path first. If not found, it will assume it is related to the current directory, i.e., the directory specified in the system property called `user.dir`.

You could specify an absolute path, such as /usr/jetty/conf/zk-log.properties, if you are not sure what the current directory is.

Disable All Logs

If you want to disable all loggers completely or change the level for all loggers, you don't need to prepare a logging configuration file. Rather, you can configure DHtmlLayoutServlet [2] in `WEB-INF/web.xml` as follows.

```
<servlet>
    <servlet-name>zkLoader</servlet-name>
    <servlet-class>org.zkoss.zk.ui.http.DHtmlLayoutServlet</servlet-class>
    <init-param>
        <param-name>log-level</param-name>
        <param-value>OFF</param-value>
    </init-param>
</servlet>
```

For more information, please refer to ZK Configuration Reference.

How to Log

Class: Log [2]

The logger used by ZK is based on the standard logger, `java.util.logging.Logger`. However, we wrap it as Log [2] to make it more efficient.

> To log the message to the client rather than the console at the server, you could use Clients.log(java.lang.String) [3]

The typical use is as follows.

```
import org.zkoss.util.logging.Log;
class MyClass {
    private static final Log log = Log.lookup(MyClass.class);
    public void f(Object v) {
        if (log.debugable()) log.debug("Value is "+v);
    }
}
```

Version History

Version	Date	Content
6.0.0	February 2012	LogService was deprecated.

References

[1] http://docs.oracle.com/javase/1.4.2/docs/guide/util/logging/overview.html
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/util/logging/Log.html#
[3] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Clients.html#log(java.lang.String)

DSP

Package: org.zkoss.web.servlet.dsp [1]

A JSP-like template technology. It takes the same syntax as that of JSP. Unlike JSP, DSP is interpreted at the run time, so it is easy to deploy DSP pages. No Java compiler is required in your run-time environment. In addition, you could distribute DSP pages in jar files.

However, you cannot embed Java codes in DSP pages. Actions of DSP, though extensible through TLD files, are different from JSP tags.

If you want to use DSP in your Web applications, you have to set up `WEB-INF/web.xml` to add the following lines.

```
    <servlet>
        <description><![CDATA[
The servlet loads the DSP pages.
        ]]></description>
        <servlet-name>dspLoader</servlet-name>
        <servlet-class>org.zkoss.web.servlet.dsp.InterpreterServlet</servlet-class>

        <!-- Specify class-resource, if you want to access TLD defined in jar files -->
        <init-param>
            <param-name>class-resource</param-name>
            <param-value>true</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>dspLoader</servlet-name>
        <url-pattern>*.dsp</url-pattern>
    </servlet-mapping>
```

The mapping of the DSP loader is optional. Specify it only if you want to write Web pages in DSP syntax.

Though standard components of ZK use DSP as a template technology, they are handled directly by ZK loader.

A Sample of DSP

```
<%@ page contentType="text/css;charset=UTF-8" %>
<%@ taglib uri="http://www.zkoss.org/dsp/web/core" prefix="c" %>

<%-- header.jsp --%>
<style>
<!--Include-->
<c:include page="/css/header.css.dsp" />


<!--Test-->
<c:if test="${c:isSafari() || c:browser('chrome')}">
.search-input-outer input {
      padding: 0 2px;
}
</c:if>

</style>
```

For more details, please check the javadoc of org.zkoss.web.servlet.dep.action [2] package.

Version History

Version	Date	Content

References

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/web/servlet/dsp/package-summary.html
[2] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/web/servlet/dsp/action/package-summary.html

iDOM

Package: org.zkoss.idom [1]

An implementation of W3C DOM. It is inspired by JDOM[2] to have concrete classes for all XML objects, such as Element and Attribute. However, iDOM implements the W3C API, such as org.w3c.dom.Element. Thus, you could use iDOM seamlessly with XML utilities that only accept the W3C DOM.

A typical example is XSLT and XPath. You could use any of favorite XSL processor and XPath utilities with iDOM.

[1] http://www.zkoss.org/javadoc/latest/zk/org/zkoss/idom/package-summary.html
[2] http://www.jdom.org

Version History

Version	Date	Content

Article Sources and Contributors

ZK Developer's Reference *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference *Contributors*: Alicelin, Southerncrossie, Sphota, Tmillsclare, Tomyeh

Overture *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Overture *Contributors*: Alicelin, Tomyeh

Architecture Overview *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Overture/Architecture_Overview *Contributors*: Alicelin, Char, Flyworld, Tomyeh

Technology Guidelines *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Overture/Technology_Guidelines *Contributors*: Alicelin, Iantsai, SimonPai, Tmillsclare, Tomyeh

Extensions *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Overture/Extensions *Contributors*: Alicelin, Chanwit, SimonPai, Tomyeh

UI Composing *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing *Contributors*: Alicelin, Tomyeh

Component-based UI *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/Component-based_UI *Contributors*: Alicelin, Gap77, Henrichen, Tomyeh

ID Space *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/ID_Space *Contributors*: Alicelin, Henrichen, M17, Tomyeh

ZUML *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/ZUML *Contributors*: Alicelin, Char, SimonPai, Tomyeh

XML Background *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/ZUML/XML_Background *Contributors*: Alicelin, Tomyeh

Basic Rules *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/ZUML/Basic_Rules *Contributors*: Alicelin, Tomyeh

EL Expressions *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/ZUML/EL_Expressions *Contributors*: Alicelin, Dark1sun, Matthewcheng, Tomyeh

Scripts in ZUML *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/ZUML/Scripts_in_ZUML *Contributors*: Alicelin, Tomyeh

Conditional Evaluation *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/ZUML/Conditional_Evaluation *Contributors*: Alicelin, Tomyeh

Iterative Evaluation *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/ZUML/Iterative_Evaluation *Contributors*: Alicelin, Tomyeh

On-demand Evaluation *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/ZUML/On-demand_Evaluation *Contributors*: Alicelin, Jumperchen, Tomyeh

Include *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/ZUML/Include *Contributors*: Benbai, Char, Jeanher, Tomyeh

Load ZUML in Java *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/ZUML/Load_ZUML_in_Java *Contributors*: Alicelin, Char, Henrichen, Tomyeh

XML Namespaces *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/ZUML/XML_Namespaces *Contributors*: Alicelin, Tomyeh

Richlet *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/Richlet *Contributors*: Alicelin, Char, M17, Sphota, Tomyeh

Macro Component *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/Macro_Component *Contributors*: Alicelin, Char, Tomyeh

Inline Macros *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/Macro_Component/Inline_Macros *Contributors*: Alicelin, Tomyeh

Implement Custom Java Class *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/Macro_Component/Implement_Custom_Java_Class *Contributors*: Alicelin, Char, Tomyeh

Composite Component *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/Composite_Component *Contributors*: Alicelin, Char, Peterkuo, Tomyeh

Client-side UI Composing *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Composing/Client-side_UI_Composing *Contributors*: Alicelin, Tomyeh

Event Handling *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Event_Handling *Contributors*: Tomyeh

Event Listening *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Event_Handling/Event_Listening *Contributors*: Alicelin, Ashishd, Henrichen, Tomyeh

Event Firing *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Event_Handling/Event_Firing *Contributors*: Alicelin, Peterkuo, Tomyeh

Event Forwarding *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Event_Handling/Event_Forwarding *Contributors*: Alicelin, Ashishd, Flyworld, Henrichen, Tomyeh

Event Queues *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Event_Handling/Event_Queues *Contributors*: Alicelin, Henrichen, Jeanher, Tmillsclare, Tomyeh, Vincent

Client-side Event Listening *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Event_Handling/Client-side_Event_Listening *Contributors*: Alicelin, Tomyeh

MVC *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVC *Contributors*: Henrichen, Tomyeh

Controller *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVC/Controller *Contributors*: Alicelin, Char, Henrichen, Tomyeh

Composer *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVC/Controller/Composer *Contributors*: Alicelin, Henrichen, M17, SimonPai, Tomyeh

Wire Components *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVC/Controller/Wire_Components *Contributors*: Tomyeh

Wire Variables *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVC/Controller/Wire_Variables *Contributors*: Alicelin, Char, Tomyeh

Wire Event Listeners *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVC/Controller/Wire_Event_Listeners *Contributors*: Alicelin, Char, Tomyeh

Model *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVC/Model *Contributors*: Alicelin, Tomyeh

List Model *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVC/Model/List_Model *Contributors*: Alicelin, SimonPai, Tomyeh

Groups Model *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVC/Model/Groups_Model *Contributors*: Alicelin, Jimmyshiau, Jumperchen, Tomyeh

Tree Model *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVC/Model/Tree_Model *Contributors*: Alicelin, Jimmyshiau, Jumperchen, SimonPai, Southerncrossie, Tomyeh, Vincent

Chart Model *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVC/Model/Chart_Model *Contributors*: Alicelin, Tomyeh

View *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVC/View *Contributors*: Alicelin, Tomyeh

Template *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVC/View/Template *Contributors*: Tomyeh

@HeaderParam *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/ViewModel/Parameters/@HeaderParam *Contributors*: Hawk, Tmillsclare

@CookieParam *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/ViewModel/Parameters/@CookieParam *Contributors*: Hawk, Tmillsclare

@ExecutionParam *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/ViewModel/Parameters/@ExecutionParam *Contributors*: Hawk, Tmillsclare

@ExecutionArgParam *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/ViewModel/Parameters/@ExecutionArgParam *Contributors*: Hawk, Tmillsclare

@ScopeParam *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/ViewModel/Parameters/@ScopeParam *Contributors*: Hawk, Tmillsclare

@SelectorParam *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/ViewModel/Parameters/@SelectorParam *Contributors*: Hawk, Tmillsclare

@ContextParam *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/ViewModel/Parameters/@ContextParam *Contributors*: Hawk, Tmillsclare

@Default *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/ViewModel/Parameters/@Default *Contributors*: Hawk, Tmillsclare

Data Binding *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/Data_Binding *Contributors*: Hawk, Tmillsclare

@id *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/Data_Binding/@id *Contributors*: Hawk

@init *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/Data_Binding/@init *Contributors*: Hawk

@load *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/Data_Binding/@load *Contributors*: Hawk, Tmillsclare

@save *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/Data_Binding/@save *Contributors*: Hawk, Tmillsclare

@bind *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/Data_Binding/@bind *Contributors*: Hawk

@command *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/Data_Binding/@command *Contributors*: Hawk

@global-command *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/Data_Binding/@global-command *Contributors*: Hawk

@converter *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/Data_Binding/@converter *Contributors*: Hawk

@validator *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/Data_Binding/@validator *Contributors*: Hawk

@template *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/MVVM/Syntax/Data_Binding/@template *Contributors*: Hawk

UI Patterns *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns *Contributors*: Alicelin, Tomyeh

Message Box *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Message_Box *Contributors*: Tomyeh

Layouts and Containers *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Layouts_and_Containers *Contributors*: Alicelin, Jimmyshiau, Southerncrossie, Tomyeh

Hflex and Vflex *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Hflex_and_Vflex *Contributors*: Alicelin, Jimmyshiau, Peterkuo, SimonPai, Tomyeh

Grid's Columns and Hflex *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Grid%27s_Columns_and_Hflex *Contributors*: Alicelin, Flyworld, Henrichen, Jimmyshiau, SimonPai, Tmillsclare, Tomyeh

Tooltips, Context Menus and Popups *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Tooltips%2C_Context_Menus_and_Popups *Contributors*: Alicelin, Jimmyshiau, Tomyeh

Keystroke Handling *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Keystroke_Handling *Contributors*: Alicelin, Peterkuo, Tmillsclare, Tomyeh

Drag and Drop *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Drag_and_Drop *Contributors*: Alicelin, Tomyeh

Page Initialization *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Page_Initialization *Contributors*: Alicelin, Flyworld, Sphota, Tomyeh

Forward and Redirect *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Forward_and_Redirect *Contributors*: Alicelin, Tomyeh, V0v87

File Upload and Download *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/File_Upload_and_Download *Contributors*: Tomyeh

Browser Information and Control *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Browser_Information_and_Control *Contributors*: Alicelin, Tomyeh

Browser History Management *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Browser_History_Management *Contributors*: Alicelin, Flyworld, Tomyeh

Session Timeout Management *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Session_Timeout_Management *Contributors*: Alicelin, Ashishd, Matthewcheng, Tomyeh

Error Handling *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Error_Handling *Contributors*: Alicelin, Tomyeh

Actions and Effects *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Actions_and_Effects *Contributors*: Alicelin, Matthewcheng, Tomyeh

HTML Tags *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/HTML_Tags *Contributors*: Alicelin, Tomyeh

The html Component *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/HTML_Tags/The_html_Component *Contributors*: Char, Tomyeh

The native Namespace *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/HTML_Tags/The_native_Namespace *Contributors*: Alicelin, Char, Tomyeh

The XHTML Component Set *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/HTML_Tags/The_XHTML_Component_Set *Contributors*: Alicelin, Char, Tomyeh

Long Operations *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Long_Operations *Contributors*: Tomyeh

Use Echo Events *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Long_Operations/Use_Echo_Events *Contributors*: Alicelin, Tomyeh

Use Event Queues *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Long_Operations/Use_Event_Queues *Contributors*: Alicelin, Tomyeh

Use Piggyback *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Long_Operations/Use_Piggyback *Contributors*: Alicelin, Tomyeh

Communication *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Communication *Contributors*: Tomyeh

Inter-Page Communication *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Communication/Inter-Page_Communication *Contributors*: Alicelin, Tomyeh

Inter-Desktop Communication *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Communication/Inter-Desktop_Communication *Contributors*: Alicelin, Tomyeh

Inter-Application Communication *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Communication/Inter-Application_Communication *Contributors*: Alicelin, Tomyeh

Templating *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Templating *Contributors*: Alicelin, Tomyeh

Composition *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Templating/Composition *Contributors*: Alicelin, Jumperchen, Tomyeh

Templates *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Templating/Templates *Contributors*: Tomyeh

XML Output *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/XML_Output *Contributors*: Alicelin, Tomyeh

Event Threads *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Event_Threads *Contributors*: Alicelin, Maya001122, Tomyeh

Modal Windows *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Event_Threads/Modal_Windows *Contributors*: Alicelin, Char, Jimmyshiau, Tomyeh

Message Box *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Event_Threads/Message_Box *Contributors*: Char, Tomyeh

File Upload *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/UI_Patterns/Event_Threads/File_Upload *Contributors*: Char, Dennischen, Tomyeh, Tonyq

Theming and Styling *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Theming_and_Styling *Contributors*: Alicelin, Tomyeh

Molds *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Theming_and_Styling/Molds *Contributors*: Alicelin, Tomyeh

CSS Classes and Styles *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Theming_and_Styling/CSS_Classes_and_Styles *Contributors*: Alicelin, Tomyeh

Theme Customization *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Theming_and_Styling/Theme_Customization *Contributors*: Alicelin, SimonPai, Tomyeh

Theme Providers *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Theming_and_Styling/Theme_Providers *Contributors*: Alicelin, Char, Tomyeh

Internationalization *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Internationalization *Contributors*: Tmillsclare, Tomyeh

Locale *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Internationalization/Locale *Contributors*: Alicelin, Char, Maya001122, Tomyeh

Time Zone *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Internationalization/Time_Zone *Contributors*: Alicelin, Maya001122, Tomyeh

Labels *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Internationalization/Labels *Contributors*: Alicelin, Char, Jimmyshiau, Maya001122, SimonPai, Tomyeh, Tonyq

The Format of Properties Files *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Internationalization/Labels/The_Format_of_Properties_Files *Contributors*: Alicelin, Ashishd, Tomyeh

Date and Time Formatting *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Internationalization/Date_and_Time_Formatting *Contributors*: Alicelin, Jumperchen, Tomyeh

The First Day of the Week *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Internationalization/The_First_Day_of_the_Week *Contributors*: Alicelin, Char, Maya001122, Tomyeh

Locale-Dependent Resources *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Internationalization/Locale-Dependent_Resources *Contributors*: Alicelin, Char, Maya001122, Tomyeh

Warning and Error Messages *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Internationalization/Warning_and_Error_Messages *Contributors*: Char, Jimmyshiau, Maya001122, Tomyeh

Server Push *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Server_Push *Contributors*: Alicelin, Sphota, Tomyeh

Event Queues *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Server_Push/Event_Queues *Contributors*: Alicelin, Tomyeh

Synchronous Tasks *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Server_Push/Synchronous_Tasks *Contributors*: Tomyeh

Asynchronous Tasks *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Server_Push/Asynchronous_Tasks *Contributors*: Alicelin, Tomyeh

Configuration *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Server_Push/Configuration *Contributors*: Alicelin, Tomyeh

Clustering *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Clustering *Contributors*: Tomyeh

ZK Configuration *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Clustering/ZK_Configuration *Contributors*: Alicelin, Jimmyshiau, Tomyeh

Server Configuration *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Clustering/Server_Configuration *Contributors*: Tomyeh

Programming Tips *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Clustering/Programming_Tips *Contributors*: Alicelin, Peterkuo, Tomyeh

Integration *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Integration *Contributors*: Tomyeh

Use ZK in JSP *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Integration/Use_ZK_in_JSP *Contributors*: Alicelin, Tmillsclare, Tomyeh

Spring *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Integration/Spring *Contributors*: Alicelin, Flyworld, Henrichen, Tomyeh

JDBC *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Integration/JDBC *Contributors*: Alicelin, Matthewcheng, Tomyeh

Hibernate *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Integration/Hibernate *Contributors*: Alicelin, Henrichen, Matthewcheng, Tomyeh

Struts *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Integration/Struts *Contributors*: Alicelin, Tomyeh

Portal *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Integration/Portal *Contributors*: Alicelin, Tomyeh

ZK Filter *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Integration/ZK_Filter *Contributors*: Tomyeh

CDI *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Integration/CDI *Contributors*: Alicelin, Paowang, Tomyeh

EJB and JNDI *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Integration/EJB_and_JNDI *Contributors*: Alicelin, Tomyeh

Google Analytics *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Integration/Google_Analytics *Contributors*: Tomyeh

Embed ZK Component in Foreign Framework *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Integration/Embed_ZK_Component_in_Foreign_Framework *Contributors*: Alicelin, Ashishd, Tomyeh

Start Execution in Foreign Ajax Channel *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Integration/Start_Execution_in_Foreign_Ajax_Channel *Contributors*: Alicelin, Ashishd, Henrichen, Tomyeh

Use ZK as Fragment in Foreign Templating Framework *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Integration/Use_ZK_as_Fragment_in_Foreign_Templating_Framework *Contributors*: Alicelin, Tomyeh

Performance Tips *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips *Contributors*: Char, Tmillsclare, Tomyeh

Use Compiled Java Codes *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips/Use_Compiled_Java_Codes *Contributors*: Alicelin, Char, Jumperchen, Maya001122, Tomyeh

Use Native Namespace instead of XHTML Namespace *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips/Use_Native_Namespace_instead_of_XHTML_Namespace *Contributors*: Alicelin, Jumperchen, Tomyeh

Use ZK JSP Tags instead of ZK Filter *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips/Use_ZK_JSP_Tags_instead_of_ZK_Filter *Contributors*: Maya001122, Tomyeh

Defer the Creation of Child Components *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips/Defer_the_Creation_of_Child_Components *Contributors*: Alicelin, Char, Maya001122, Tomyeh

Defer the Rendering of Client Widgets *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips/Defer_the_Rendering_of_Client_Widgets *Contributors*: Alicelin, Jumperchen, Maya001122, Tomyeh

Client Render on Demand *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips/Client_Render_on_Demand *Contributors*: Char, Maya001122, Tomyeh

Listbox, Grid and Tree for Huge Data *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips/Listbox%2C_Grid_and_Tree_for_Huge_Data *Contributors*: Alicelin, Tomyeh

Use Live Data and Paging *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips/Listbox%2C_Grid_and_Tree_for_Huge_Data/Use_Live_Data_and_Paging *Contributors*: Alicelin, Char, Maya001122, Sphota, Tomyeh

Turn on Render on Demand *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips/Listbox%2C_Grid_and_Tree_for_Huge_Data/Turn_on_Render_on_Demand *Contributors*: Alicelin, Ashishd, Char, Jimmyshiau, Maya001122, SimonPai, Tomyeh

Implement ListModel and TreeModel *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips/Listbox%2C_Grid_and_Tree_for_Huge_Data/Implement_ListModel_and_TreeModel *Contributors*: Alicelin, Jumperchen, Tomyeh

Minimize Number of JavaScript Files to Load *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips/Minimize_Number_of_JavaScript_Files_to_Load *Contributors*: Alicelin, Jimmyshiau, Jumperchen, Tomyeh, Tonyq

Load JavaScript and CSS from Server Nearby *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips/Load_JavaScript_and_CSS_from_Server_Nearby *Contributors*: Alicelin, Char, Flyworld, Tomyeh

Specify Stubonly for Client-only Components *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips/Specify_Stubonly_for_Client-only_Components *Contributors*: Alicelin, Char, Tomyeh

Reuse Desktops *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips/Reuse_Desktops *Contributors*: Char, Maya001122, Tomyeh

Miscellaneous *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Tips/Miscellaneous *Contributors*: Maya001122, Tomyeh

Security Tips *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Security_Tips *Contributors*: Jeanher, Tomyeh

Cross-site scripting *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Security_Tips/Cross-site_scripting *Contributors*: Alicelin, Tomyeh

Block Request for Inaccessible Widgets *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Security_Tips/Block_Request_for_Inaccessible_Widgets *Contributors*: Alicelin, Tomyeh

Performance Monitoring *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Monitoring *Contributors*: Alicelin, Tomyeh

Performance Meters *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Monitoring/Performance_Meters *Contributors*: Alicelin, Tomyeh

Event Interceptors *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Monitoring/Event_Interceptors *Contributors*: Tomyeh

Loading Monitors *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Performance_Monitoring/Loading_Monitors *Contributors*: Tomyeh

Testing *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Testing *Contributors*: Alicelin, Char, Tomyeh

Testing Tips *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Testing/Testing_Tips *Contributors*: Alicelin, Char, Tomyeh

ZTL *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Testing/ZTL *Contributors*: Jumperchen, Tomyeh

Customization *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Customization *Contributors*: Tomyeh

Packing Code *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Customization/Packing_Code *Contributors*: Alicelin, Tomyeh

Component Properties *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Customization/Component_Properties *Contributors*: Alicelin, Tomyeh

UI Factory *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Customization/UI_Factory *Contributors*: Alicelin, Tomyeh

Init and Cleanup *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Customization/Init_and_Cleanup *Contributors*: Tomyeh

AU Services *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Customization/AU_Services *Contributors*: Tomyeh

AU Extensions *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Customization/AU_Extensions *Contributors*: Tomyeh

How to Build ZK Source Code *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Customization/How_to_Build_ZK_Source_Code *Contributors*: Alicelin, Jumperchen, Tmillsclare, Tomyeh

Supporting Utilities *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Supporting_Utilities *Contributors*: Alicelin, Tomyeh

Logger *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Supporting_Utilities/Logger *Contributors*: Tomyeh

DSP *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Supporting_Utilities/DSP *Contributors*: Flyworld, Tomyeh

iDOM *Source*: http://new.zkoss.org/index.php?title=ZK_Developer%27s_Reference/Supporting_Utilities/iDOM *Contributors*: 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_MultiPage.png *Source*: http://new.zkoss.org/index.php?title=File:ZKEssentials_Intro_MultiPage.png *License*: unknown *Contributors*: Sphota
Image:zk the id space.jpg *Source*: http://new.zkoss.org/index.php?title=File:Zk_the_id_space.jpg *License*: unknown *Contributors*: Maya001122
File:Eventqueue-concept.jpg *Source*: http://new.zkoss.org/index.php?title=File:Eventqueue-concept.jpg *License*: unknown *Contributors*: Tomyeh
File:MVC.png *Source*: http://new.zkoss.org/index.php?title=File:MVC.png *License*: unknown *Contributors*: Tomyeh
Image:Composer.PNG *Source*: http://new.zkoss.org/index.php?title=File:Composer.PNG *License*: unknown *Contributors*: Tomyeh
Image:DrListModelRenderer.png *Source*: http://new.zkoss.org/index.php?title=File:DrListModelRenderer.png *License*: unknown *Contributors*: Tomyeh
Image:DrGroupsModel.png *Source*: http://new.zkoss.org/index.php?title=File:DrGroupsModel.png *License*: unknown *Contributors*: Tomyeh
Image:Grouping_model_explain.png *Source*: http://new.zkoss.org/index.php?title=File:Grouping_model_explain.png *License*: unknown *Contributors*: Tomyeh
Image:DrGroupsModelArray.png *Source*: http://new.zkoss.org/index.php?title=File:DrGroupsModelArray.png *License*: unknown *Contributors*: Tomyeh
Image:DrTreeModel1.png *Source*: http://new.zkoss.org/index.php?title=File:DrTreeModel1.png *License*: unknown *Contributors*: Tomyeh
Image:DrTreeModel2.png *Source*: http://new.zkoss.org/index.php?title=File:DrTreeModel2.png *License*: unknown *Contributors*: Tomyeh
File:St201107-listbox.png *Source*: http://new.zkoss.org/index.php?title=File:St201107-listbox.png *License*: unknown *Contributors*: Tomyeh
File:St201107-listbox-in-listbox.png *Source*: http://new.zkoss.org/index.php?title=File:St201107-listbox-in-listbox.png *License*: unknown *Contributors*: Tomyeh
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
File:Mvvm-databinding-role.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-databinding-role.png *License*: unknown *Contributors*: Hawk
File:Mvvm-event-command-reload.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-event-command-reload.png *License*: unknown *Contributors*: Hawk
File:Mvvm-viewmodel-command.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-viewmodel-command.png *License*: unknown *Contributors*: Hawk
File:Mvvm-binder.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-binder.png *License*: unknown *Contributors*: Hawk
File:Mvvm-command-execution.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-command-execution.png *License*: unknown *Contributors*: Hawk
File:Mvvm-children-binding.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-children-binding.png *License*: unknown *Contributors*: Hawk
File:Mvvm-dynamic-menu.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-dynamic-menu.png *License*: unknown *Contributors*: Hawk
File:Mvvm-form-binding.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-form-binding.png *License*: unknown *Contributors*: Hawk
File:smalltalks-mvvm-in-zk6-formbinding-form-dirty.png *Source*: http://new.zkoss.org/index.php?title=File:Smalltalks-mvvm-in-zk6-formbinding-form-dirty.png *License*: unknown *Contributors*: Hawk
File:Mvvm-global-command-overview.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-global-command-overview.png *License*: unknown *Contributors*: Hawk
File:Mvvm-global-command-simple.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-global-command-simple.png *License*: unknown *Contributors*: Hawk
File:Mvvm-global-command-simple-add.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-global-command-simple-add.png *License*: unknown *Contributors*: Hawk
File:Mvvm-global-command-simple-fail.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-global-command-simple-fail.png *License*: unknown *Contributors*: Hawk
File:Mvvm-global-command-menu.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-global-command-menu.png *License*: unknown *Contributors*: Hawk
File:Mvvm-global-command-execution.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-global-command-execution.png *License*: unknown *Contributors*: Hawk
File:Mvvm-binding-parameters.png *Source*: http://new.zkoss.org/index.php?title=File:Mvvm-binding-parameters.png *License*: unknown *Contributors*: Hawk
File:DrMessagebox.png *Source*: http://new.zkoss.org/index.php?title=File:DrMessagebox.png *License*: unknown *Contributors*: Tomyeh
File:DrMessagebox-error.png *Source*: http://new.zkoss.org/index.php?title=File:DrMessagebox-error.png *License*: unknown *Contributors*: Tomyeh
Image:DrHlayout.png *Source*: http://new.zkoss.org/index.php?title=File:DrHlayout.png *License*: unknown *Contributors*: Tomyeh
Image:DrVlayout.png *Source*: http://new.zkoss.org/index.php?title=File:DrVlayout.png *License*: unknown *Contributors*: Tomyeh
Image:DrHlayout_scrolling.png *Source*: http://new.zkoss.org/index.php?title=File:DrHlayout_scrolling.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrVlayout_scrolling.png *Source*: http://new.zkoss.org/index.php?title=File:DrVlayout_scrolling.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrHlayout_alignment.png *Source*: http://new.zkoss.org/index.php?title=File:DrHlayout_alignment.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrHbox.png *Source*: http://new.zkoss.org/index.php?title=File:DrHbox.png *License*: unknown *Contributors*: Tomyeh
Image:DrVbox.png *Source*: http://new.zkoss.org/index.php?title=File:DrVbox.png *License*: unknown *Contributors*: Tomyeh
Image:DrHbox_align.png *Source*: http://new.zkoss.org/index.php?title=File:DrHbox_align.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrHbox_pack.png *Source*: http://new.zkoss.org/index.php?title=File:DrHbox_pack.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrVbox_align.png *Source*: http://new.zkoss.org/index.php?title=File:DrVbox_align.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrVbox_pack.png *Source*: http://new.zkoss.org/index.php?title=File:DrVbox_pack.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrHbox_Cell.png *Source*: http://new.zkoss.org/index.php?title=File:DrHbox_Cell.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrVbox_Cell.png *Source*: http://new.zkoss.org/index.php?title=File:DrVbox_Cell.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrBorderlayout.png *Source*: http://new.zkoss.org/index.php?title=File:DrBorderlayout.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau, Tomyeh
Image:DrBorderlayout_flex.png *Source*: http://new.zkoss.org/index.php?title=File:DrBorderlayout_flex.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrBorderlayout_Center_scrolling.png *Source*: http://new.zkoss.org/index.php?title=File:DrBorderlayout_Center_scrolling.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrBorderlayout_grow.png *Source*: http://new.zkoss.org/index.php?title=File:DrBorderlayout_grow.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrColumnlayout.png *Source*: http://new.zkoss.org/index.php?title=File:DrColumnlayout.png *License*: unknown *Contributors*: Tomyeh
Image:DrPortallayout.png *Source*: http://new.zkoss.org/index.php?title=File:DrPortallayout.png *License*: unknown *Contributors*: Tomyeh
Image:DrTablelayout.png *Source*: http://new.zkoss.org/index.php?title=File:DrTablelayout.png *License*: unknown *Contributors*: Tomyeh
Image:DrDivSpan.png *Source*: http://new.zkoss.org/index.php?title=File:DrDivSpan.png *License*: unknown *Contributors*: Tomyeh
Image:DrDiv_scrolling.png *Source*: http://new.zkoss.org/index.php?title=File:DrDiv_scrolling.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrWindow.png *Source*: http://new.zkoss.org/index.php?title=File:DrWindow.png *License*: unknown *Contributors*: Tomyeh
Image:DrWindow_scrolling.png *Source*: http://new.zkoss.org/index.php?title=File:DrWindow_scrolling.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrPanel.png *Source*: http://new.zkoss.org/index.php?title=File:DrPanel.png *License*: unknown *Contributors*: Tomyeh
Image:DrPanel_scrolling.png *Source*: http://new.zkoss.org/index.php?title=File:DrPanel_scrolling.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrGroupbox3d.png *Source*: http://new.zkoss.org/index.php?title=File:DrGroupbox3d.png *License*: unknown *Contributors*: Tomyeh
Image:DrGroupbox3d_scrolling.png *Source*: http://new.zkoss.org/index.php?title=File:DrGroupbox3d_scrolling.png *License*: unknown *Contributors*: Jimmyshiau
Image:DrTabbox.png *Source*: http://new.zkoss.org/index.php?title=File:DrTabbox.png *License*: unknown *Contributors*: Tomyeh
Image:DrTabbox_scrolling.png *Source*: http://new.zkoss.org/index.php?title=File:DrTabbox_scrolling.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau
Image:DrFlex1.png *Source*: http://new.zkoss.org/index.php?title=File:DrFlex1.png *License*: unknown *Contributors*: Tomyeh
Image:DrFlexTabbox.png *Source*: http://new.zkoss.org/index.php?title=File:DrFlexTabbox.png *License*: unknown *Contributors*: Tomyeh
Image:DrFlex2.png *Source*: http://new.zkoss.org/index.php?title=File:DrFlex2.png *License*: unknown *Contributors*: Tomyeh

Image:DrFlexErr1Fix.png *Source*: http://new.zkoss.org/index.php?title=File:DrFlexErr1Fix.png *License*: unknown *Contributors*: Tomyeh

Image:vflexborderlayout.png *Source*: http://new.zkoss.org/index.php?title=File:Vflexborderlayout.png *License*: unknown *Contributors*: Elton776, Jimmyshiau

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

File:ZK5DevRef_GridColumn_FormHflex.png *Source*: http://new.zkoss.org/index.php?title=File:ZK5DevRef_GridColumn_FormHflex.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau

File:ZK5DevRef_GridColumn_FormHflex2.png *Source*: http://new.zkoss.org/index.php?title=File:ZK5DevRef_GridColumn_FormHflex2.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau

File:ZK5DevRef_GridColumn_FormHflex_colspan.png *Source*: http://new.zkoss.org/index.php?title=File:ZK5DevRef_GridColumn_FormHflex_colspan.png *License*: unknown *Contributors*: Alicelin, Jimmyshiau

Image:DrFlexErr1.png *Source*: http://new.zkoss.org/index.php?title=File:DrFlexErr1.png *License*: unknown *Contributors*: Tomyeh

Image:DrFlexErr2.png *Source*: http://new.zkoss.org/index.php?title=File:DrFlexErr2.png *License*: unknown *Contributors*: Tomyeh

File:ZK5DevRef_GridColumn_Default.png *Source*: http://new.zkoss.org/index.php?title=File:ZK5DevRef_GridColumn_Default.png *License*: unknown *Contributors*: Flyworld

File:ZK5DevRef_GridColumn_hflex.png *Source*: http://new.zkoss.org/index.php?title=File:ZK5DevRef_GridColumn_hflex.png *License*: unknown *Contributors*: Flyworld

File:ZK5DevRef_GridColumn_nospan.png *Source*: http://new.zkoss.org/index.php?title=File:ZK5DevRef_GridColumn_nospan.png *License*: unknown *Contributors*: Flyworld

File:ZK5DevRef_GridColumn_span.png *Source*: http://new.zkoss.org/index.php?title=File:ZK5DevRef_GridColumn_span.png *License*: unknown *Contributors*: Flyworld

File:ZK5DevRef_GridColumn_span0.png *Source*: http://new.zkoss.org/index.php?title=File:ZK5DevRef_GridColumn_span0.png *License*: unknown *Contributors*: SimonPai

File:ZK5DevRef_GridColumn_sizedByCnt.png *Source*: http://new.zkoss.org/index.php?title=File:ZK5DevRef_GridColumn_sizedByCnt.png *License*: unknown *Contributors*: SimonPai

File:ZK5DevRef_GridColumn_DefaultLong.png *Source*: http://new.zkoss.org/index.php?title=File:ZK5DevRef_GridColumn_DefaultLong.png *License*: unknown *Contributors*: Flyworld

File:ZK5DevRef_GridColumn_DefaultWidth.png *Source*: http://new.zkoss.org/index.php?title=File:ZK5DevRef_GridColumn_DefaultWidth.png *License*: unknown *Contributors*: Flyworld

File:ZK5DevRef_GridColumn_LongHflex.png *Source*: http://new.zkoss.org/index.php?title=File:ZK5DevRef_GridColumn_LongHflex.png *License*: unknown *Contributors*: Flyworld

File:ZK5DevRef_GridColumn_MixhflexnumWidth.png *Source*: http://new.zkoss.org/index.php?title=File:ZK5DevRef_GridColumn_MixhflexnumWidth.png *License*: unknown *Contributors*: Flyworld

File:ZK5DevRef_GridColumn_MixhflexMinWidth.png *Source*: http://new.zkoss.org/index.php?title=File:ZK5DevRef_GridColumn_MixhflexMinWidth.png *License*: unknown *Contributors*: Flyworld

File:ZK5DevRef_GridColumn_MixhflAllh.png *Source*: http://new.zkoss.org/index.php?title=File:ZK5DevRef_GridColumn_MixhflAllh.png *License*: unknown *Contributors*: Flyworld, Jimmyshiau

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

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

Image:10000000000001750000052E60F488A.png *Source*: http://new.zkoss.org/index.php?title=File:10000000000001750000052E60F488A.png *License*: unknown *Contributors*: Maya001122

File:events_1_finger.jpg *Source*: http://new.zkoss.org/index.php?title=File:Events_1_finger.jpg *License*: unknown *Contributors*: Jimmyshiau

Image:10000000000002AF000001BB582C2DD7.png *Source*: http://new.zkoss.org/index.php?title=File:10000000000002AF000001BB582C2DD7.png *License*: unknown *Contributors*: Char

Image:100000000000036D000000FE561CE3BC.png *Source*: http://new.zkoss.org/index.php?title=File:100000000000036D000000FE561CE3BC.png *License*: unknown *Contributors*: Maya001122

Image:10000000000002840000000226A7DEE65.png *Source*: http://new.zkoss.org/index.php?title=File:10000000000002840000000226A7DEE65.png *License*: unknown *Contributors*: Maya001122

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

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

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

Image:Exception-au2.png *Source*: http://new.zkoss.org/index.php?title=File:Exception-au2.png *License*: unknown *Contributors*: Tomyeh

Image:html_1.png *Source*: http://new.zkoss.org/index.php?title=File:Html_1.png *License*: unknown *Contributors*: Char

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

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

Image:html_5.png *Source*: http://new.zkoss.org/index.php?title=File:Html_5.png *License*: unknown *Contributors*: Char

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

Image:performancemeter.png *Source*: http://new.zkoss.org/index.php?title=File:Performancemeter.png *License*: unknown *Contributors*: Elton776