# ZK Essentials

An example-based guide for beginners

zk

# Table of Contents

current:

Improve this Doc

current:

Improve this Doc

# ZK's Value and Strength

ZK is a component-based UI framework that enables you to build Rich Internet Application (*RIA*) and mobile applications without having to learn JavaScript or AJAX. You can build highly-interactive and responsive AJAX web applications in pure **Java**. ZK provides hundreds of components which are designed for various purposes, some for displaying large amount of data and some for user input. We can easily create components in an XML-formatted language, **ZUL**.

All user actions on a page such as clicking and typing can be easily handled in a Controller. You can manipulate components to respond to users action in a Controller and the changes you made will reflect to browsers automatically. You don't need to care about communication details between browsers and servers, ZK will handle AJAX details for you. In addition to manipulating components directly i.e. MVC (Model-View-Controller) approach, ZK also supports MVVM (Model-View-ViewModel) approach which decouples the Controller and View. These two approaches are mutually interchangeable, and you can choose one of them upon your architectural consideration.

# Architecture of ZK
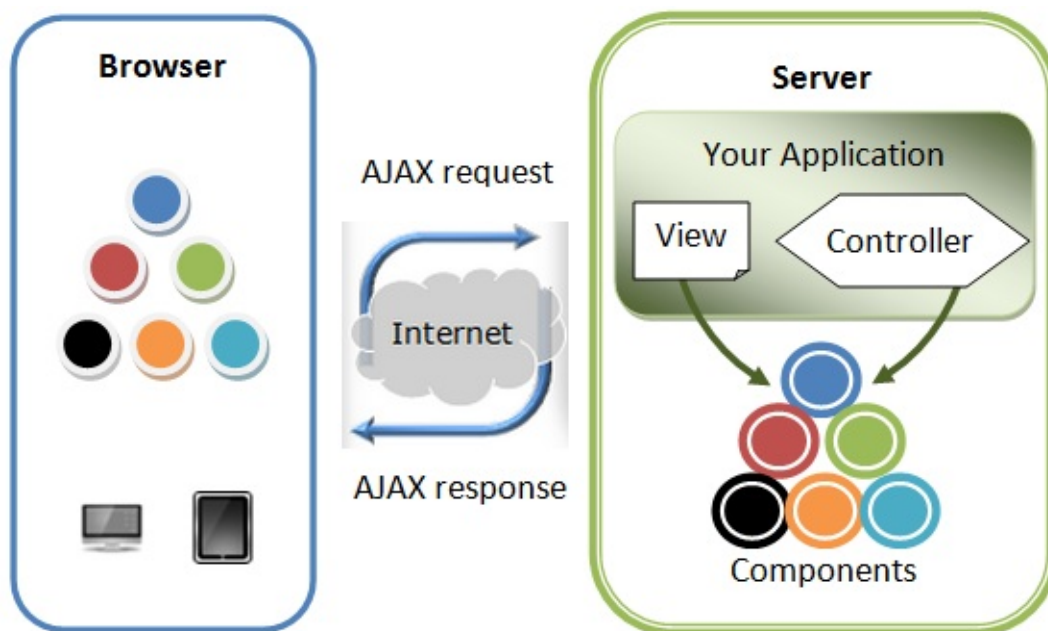


Above image is a simplified ZK architecture. When a browser visits a page of a ZK application, ZK creates components written in ZUL and renders them on the browser. You can manipulate components by your application's Controller to implement UI presentation logic. All changes you made on components will automatically reflect on users' browser and ZK handles underlying communication for you.

ZK application developed in a server-centric way can easily access Java EE technology stack and integrate many great third party Java frameworks like Spring or Hibernate. Moreover, ZK also supports client-centric development that allows you to customize visual effect or handle user actions at the client side.

# About This Book

This book presents you ZK key concepts and suggested usage from the perspective of building a web application. Each chapter has a main topic, and we introduce each chapter's topic with one or more example applications. Each chapter's applications are built upon previous chapter's application and add more features. In the last chapter, the example application becomes close to a real application. The source code of the example applications can be downloaded through github, please refer to Project Structure.

Chapter 1, we introduce how to build a common layout which contains a header, a footer, and a sidebar.

Chapter 2, we tell you how to control components programmatically.

Chapter 3, it describes how to collect, validate user input and response.

Chapter 4, we demonstrate how to implement common CRUD operations with a To-Do list application.

Chapter 5, it mentions about shadow components, a new component set introduced since ZK 8.

Chapter 6, we introduce 2 navigation ways in ZK, page-based and AJAX-based.

Chapter 7, it demonstrates a simple implementation to authenticate users.

Chapter 8, we describe how to integrate Spring framework into a ZK application.

Chapter 9, we demonstrate how to use JPA in a ZK application.

# Example Application

This book will guide you to build a small and rich application that has common features such as authentication, navigation, form input, and personal to-do list management as a final result. It manages its infrastructure with Spring and persists data into a database with JPA.

This application has a common layout. The header above has application's icon and its title, and the footer at the bottom contains general information. The central area displays the current main function. You must login before you can access other functions.
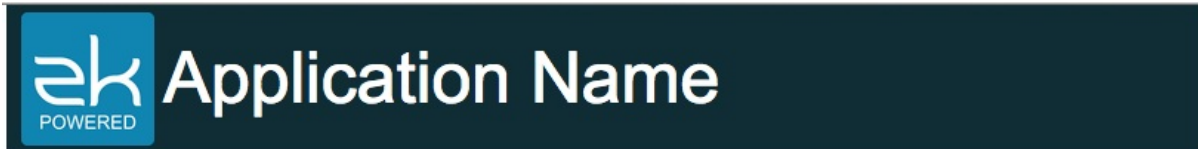
(use account='zkoss' and password='1234' to login)



**Example application - login**

After login, you can see the main page. The sidebar on the left is a navigation bar that allows you to switch between different functions. The upper three items lead you to external sites. There are 2 main functions, profile and todo list management, which are implemented by both the MVC and MVVM approach.

**Example application - profile form**

The image below shows the Todo list management function, you can create, delete, and update a todo item.

**Example application - todo list**

current:

Improve this Doc

# Source Code

All source codes used in this book are available on github. As our example application has 3 different configurations, our source code is divided into 3 branches:

- **master**: contains examples from chapter 1 to chapter 7.
- **zk-spring**: has examples integrated with Spring
- **zk-jpa**: contains examples which integrate with Spring and persiste data into a database with JPA.
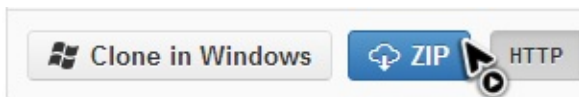


You can click the "ZIP" icon to download the current selected branch as a zip file.

# Run Example Application

After you download the source code, you will find it is a Apache Maven project with jetty plugin configured. Therefore, you can start the example application on Jetty without deploying.

# No Maven Installed

Even you don't install Maven, you can start the project with maven wrapper by the command below (it will automatically download required maven):

## Linux/Mac

```
./mvnw jetty:run
```

## Windows

```
mvnw jetty:run
```

# With Maven

Navigate to the root folder of the example project, e.g. it's "zkessentials" and type the command:

```
mvn jetty:run
```

# With Gradle

You have 2 options:

- Start with the gradle plugin gretty

```
gradle appRun
```

- Start with embedded jetty-runner that we configured in build.gradle

```
gradle startJettyRunner
```

After starting up, visit the URL http://localhost:8080/zkessentials/, and you should see the page below:
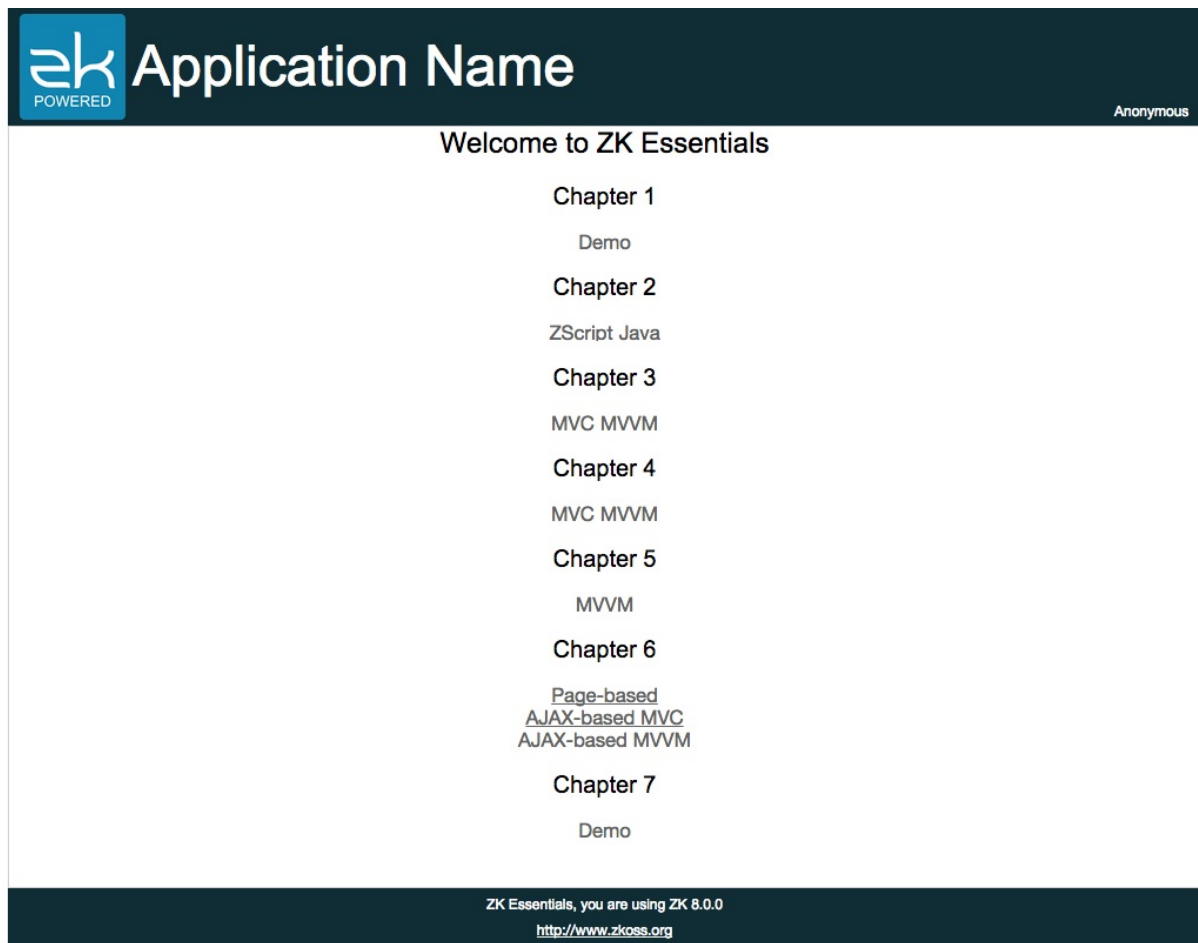


# Project Structure

The example project's folder structure follows Maven's default convention. We name Java packages according to each chapter, and each package contains the classes of that chapter. Some common classes are separated to an independent package as they are used in multiple chapters, e.g. the classes under `org.zkoss.essentials.entity.*` are entity class. We also define some service layer interfaces under `org.zkoss.essentials.service.*` because different chapters have different implementations.

For ZUL pages, we put them in separate folders for each chapter under `src/main/webapp/` . Under "WEB-INF" folder, **web.xml** contains minimal configuration to run ZK and for its detail please refer to ZK Installation Guide \ Create and Run Your First ZK Application Manually. The "zk.xml" is optional configuration descriptor of ZK. Provide this file if you need to configure ZK differently from the default behavior. Refer to ZK Configuration Reference/zk.xml for more detail.

current:

Improve this Doc

# Design the Layout

ZK provides various layout components for different layout requirements, and you can even configure a component's attribute to adjust layout details.

## Layout Requirement
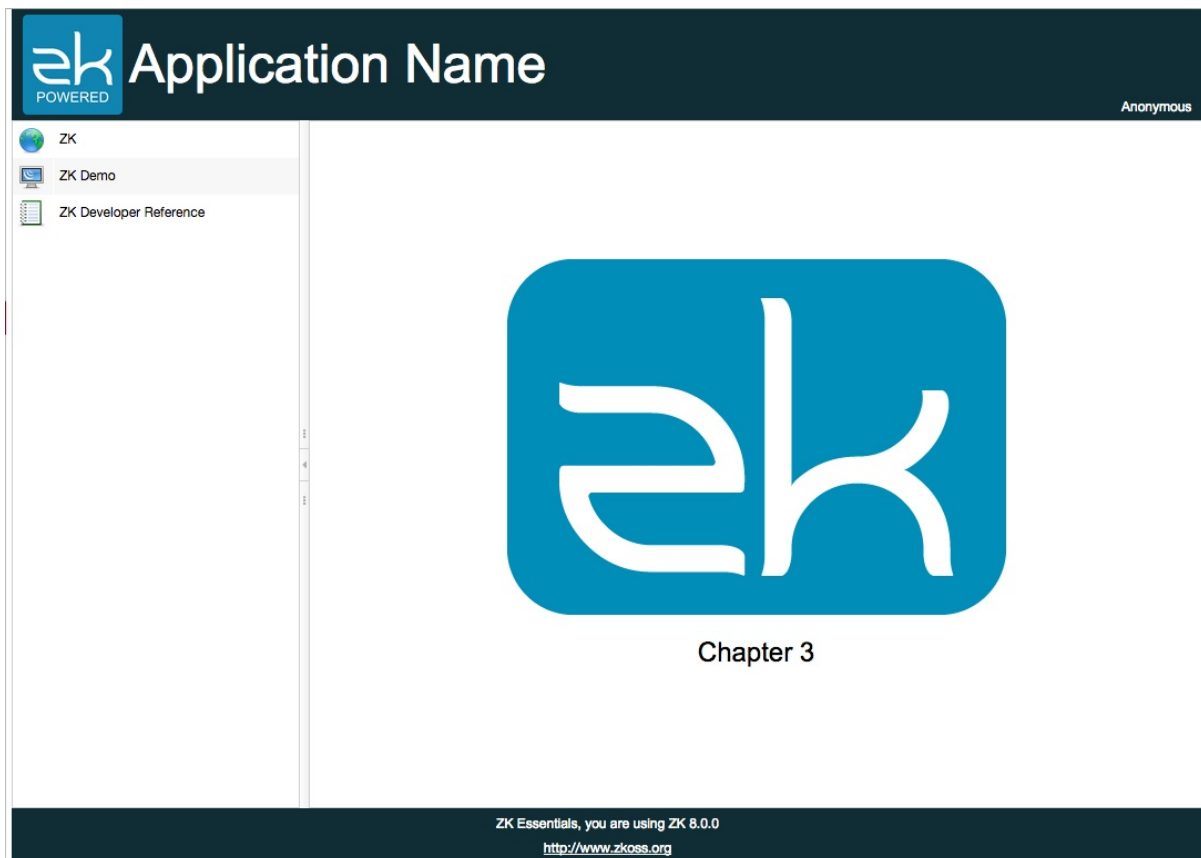


The image above is the target layout we are going to create in this chapter, and this kind of design is very common in web applications. It divides a page into several section:

- The **banner** at the top contains application icon, title, and user's name at the right most corner.
- The **footer** at the bottom contains general information.
- The **sidebar** on the left contains 3 links that direct you to 3 different URLs.
- The **central area** displays the current function.

ZK provides various layout components, and each of them has different styles. You can use a layout component alone or combine them to build a more complex layout. According to our requirement, *Border Layout* fits the requirement most since it has 5 areas: **north, west, center, east, and south**. We can use the north as a banner, west as a sidebar, south as a footer, and the center as the main content.
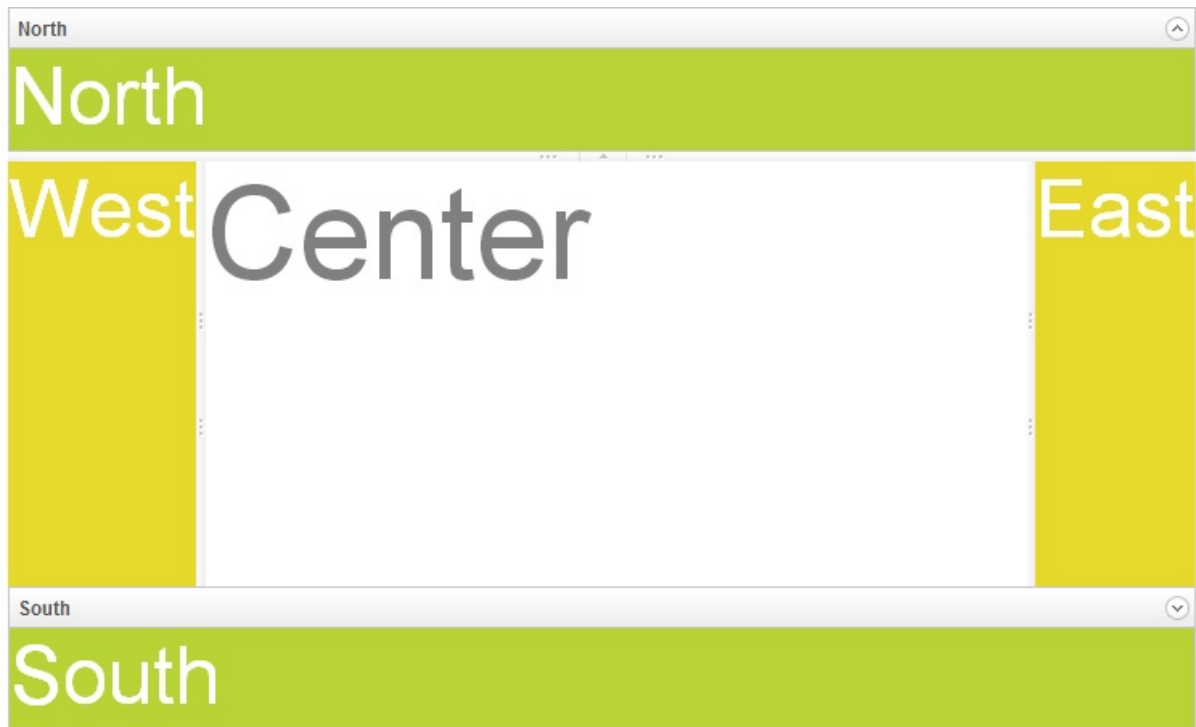
**Border Layout**

# Build the View

Building the view in ZK is basically creating components, and there are two ways to do it:

- **ZUML** (XML format, declarative) approach.
- **Java** (programmatic)

We encourage you to build UI with the XML-formatted language called **ZK User-Interface Markup Language (ZUML)**. Each XML element instructs ZK to create a component, and each XML attribute controls the component's function and looking. We will demonstrate this way mainly in our example.

You can even mix these two ways. For example, create a simple page with zul, then add components dynamically in Java within a ZK controller. You can see such usage in the subsequent chapters.

In addition to above ways, richlet allows you to compose the whole page in Java programmatically.

# Write a ZUL File

To create components in ZUML, you need to create a file with the file extension "**.zul**". In zul files, an XML element(tag) represent one component, and you can configure each component's style, behavior, and function by setting the element's attributes. Please refer to ZK Component Reference and Javadoc for complete component attributes.

A very simple, classic zul example is like:

```
<zk>
    <label value="hello &#xA;world"/>
</zk>
```

The tag `<label>` will create a `Label` component at the server and display "hello world" in one line in your browser. It's one line because it ignores special characters like a new line ( `&#xA;` ) by default.

But we can specify `pre` attribute to preserve new line character, so that it will display "hello world" in 2 lines.

```
<zk>
    <label pre="true" value="hello &#xA;world"/>
</zk>
```

Therefore, one ZK component can provide various functions with different attributes.

# Tools for IDE

### Eclipse

If you use Eclipse, we encourage you to install our Eclipse plugin, ZK Studio. Its "content assist" can save you from memorizing component name and attributes. The "word completion" can avoid typos.

### IntelliJ IDEA

If you prefer IntelliJ IDEA as I do, we also develop a ZK plugin, it supports content assist, word completion, and data binding syntax assist, too.

# Build Application Layout

Now, let's built our example application's main layout. create a new text file with name `index.zul` , and type the following content:

**Extracted from chapter1/index.zul**

```
<zk>
    <borderlayout hflex="1" vflex="1">
        <north height="100px" border="none" >
            <label style="font-size:50px">North for header</label>
        </north>
        <west width="260px" border="none" collapsible="true"
             splittable="true" minsize="300">
            <label style="font-size:50px">West for sidebar</label>
        </west>
        <center id="mainContent" autoscroll="true">
            <label style="font-size:50px">Center for content</label>
        </center>
        <south height="50px" border="none">
            <label style="font-size:50px">South for footer</label>
        </south>
    </borderlayout>
</zk>
```

- Line 2: Each XML tag represents one component, and the tag name is equal to the component name. The attribute "hflex" and "vflex" controls the horizontal and vertical size flexibility of a component. We set them to "1" which means *Fit-the-Rest* flexibility. Hence, the `<borderlayout>` will stretch itself to fill all available space of whole page in width and height because it is a root component. Only one component is allowed inside `<north>` in addition to a `<caption>` .
- Line 3: `<north>` * is a child component that can only be put inside a `<borderlayout>` . You can also fix a component's height by specifying a pixel value to avoid its height changing due to browser sizes.
- Line 6, 7: Setting `collapsible` to true allows you to collapse the `<west>` area by clicking an arrow button. Setting `splittable` to true allows you to adjust the width of `<west>` * and `minsize` limits the minimal size of width you can adjust.
- Line 10: Setting `autoscroll` to true will decorate the `<center>` *with a scroll bar when* `<center>` contains lots of information that exceed the its height.
- Line 4,8,11,14: These `<label>` s are just for identifying `borderlayout` 's areas and we will remove them in the final result.

Then, you can view the result from your browser as below:
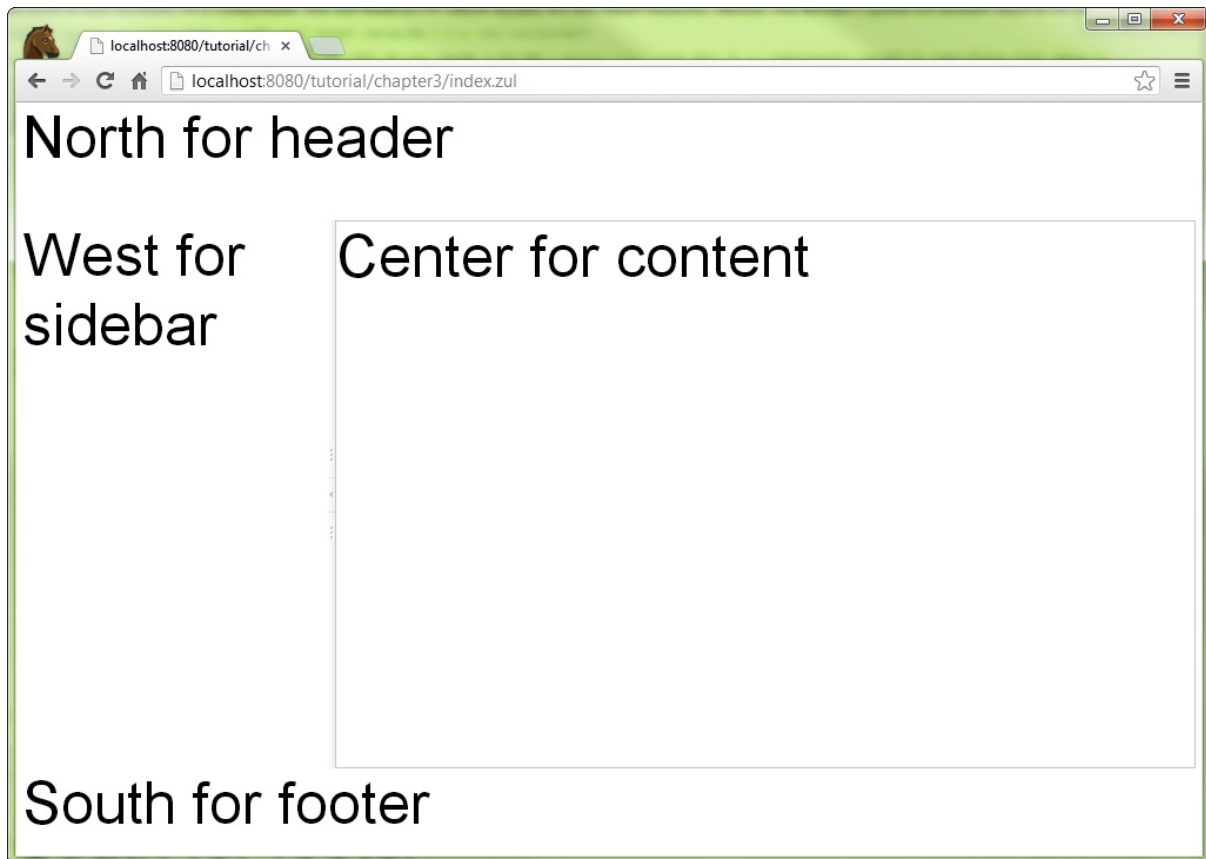
## Construct User Interface with Components

Now we have a skeleton of the application, the next we need to do is to fill each area with components. We will create a separate zul file for each area and then combine them together.

**chapter1/main.zul**

```
<vbox vflex="1" hflex="1" align="center"
        pack="center" spacing="20px" >
    <image src="/imgs/zklogo2.png" />
    <label value="Chapter 3" sclass="head1"/>
</vbox>
```

- Line 1: The `spacing` controls the space between child components it contains.

In the banner, there's an image with a hyperlink, title, and user name. Let's see how to construct these elements with existing ZK components:

**chapter1/banner.zul**

```
<div hflex="1" sclass="banner">
    <hbox hflex="1" align="center">
        <a href="http://www.zkoss.org/">
            <image src="/imgs/zklogo.png" width="90px" />
        </a>
        <div width="400px">
            <label value="Application Name" sclass="banner-head" />
        </div>
        <hbox hflex="1" vflex="1" pack="end" align="end">
            Anonymous
        </hbox>
```

```
        </hbox>
    </div>
```

- Line 1: The `sclass` , we can specify CSS class selector, and we will talk about it later.
- Line 2: The *Hbox* which is a layout component can arrange its child components in a row horizontally. Its `align` attribute controls the vertical alignment.
- Line 3: The `A` creates a hyperlink the same as an HTML `<a>` element.
- Line 4: The *image* is similar to HTML `<img>` which can display an image.
- Line 9: The `pack` controls the horizontal alignment. We specify `end` on both `pack` and `align` to make the text "Anonymous" display at the bottom right corner.
- Line 10: Here we still don't implement authentication yet, so we use static user name "Anonymous" here.

For the sidebar, we want to arrange navigation items one by one vertically. There are more than one way to achieve this. Here, we use a *Grid* which is suitable for arranging child components in a matrix layout.

**chapter1/sidebar.zul**

```
<grid hflex="1" vflex="1" sclass="sidebar">
    <columns>
        <column width="36px"/>
        <column/>
    </columns>
    <rows>
        <row>
            <image src="/imgs/site.png"/>
            <a href="http://www.zkoss.org/">ZK</a>
        </row>
        <row>
            <image src="/imgs/demo.png"/>
            <a href="http://www.zkoss.org/zkdemo">ZK Demo</a>
        </row>
        <row>
            <image src="/imgs/doc.png"/>
            <a href="http://books.zkoss.org/wiki/ZK_Developer's_Reference">
            ZK Developer Reference
            </a>
        </row>
    </rows>
</grid>
```

- Line 1: Some components like *Grid* supports limited child components and you should also notice hierarchical relations between child components, e.g. *Rows* can only contain *Row*. Please refer to ZK Component Reference/Data for detail.
- Line 3: You can only put *Column* inside *Columns*.
- Line 8,9: Since we define two *Column*s, each *Row* can have two components, and each one belongs to a column.

We usually put some contact information in the footer and make it aligned to the center.

**chapter1/footer.zul**

```
<div hflex="1" vflex="1" sclass="footer">
    <vbox hflex="1" vflex="1" align="center">
        ZK Essentials, you are using ZK ${desktop.webApp.version}
        <a href="http://www.zkoss.org">http://www.zkoss.org</a>
    </vbox>
</div>
```

- Line 2: The *Vbox*, like *Hbox,* arranges child components vertically. We specify "center" at `align` to align those texts horizontally in the center.
- Line 3: You can use EL expressions in the tag element's body or an attribute. There are also many implicit objects), and `desktop` is one of them. Refer to `org.zkoss.zk.ui.Desktop` 's Javadoc to find out available properties.

Next, we will combine these separated zul pages into `chapter1/index.zul` .

# Include a Separate Page

To complete the page, we need to put those individual pages into corresponding area of the *BorderLayout*.

For all areas, we use `<apply>` , a shadow component, to combine separated pages, and it doesn't consume extra server memory. This component can inject a separated zul for you when the parent zul is loaded. This usage is presented below:

**chapter1/index.zul**

```
<?link rel="stylesheet" type="text/css" href="/style.css"?>
<zk>
    <custom-attributes org.zkoss.zul.image.preload="true"/>
    <borderlayout hflex="1" vflex="1">
        <north vflex="min" border="none" >
            <apply templateURI="/chapter1/banner.zul"/>
        </north>
        <west width="260px" border="none" collapsible="true" splittable="true" minsize="300">
            <apply templateURI="/chapter1/sidebar.zul"/>
        </west>
        <center id="mainContent" autoscroll="true">
            <apply templateURI="/chapter1/main.zul"/>
        </center>
        <south border="none">
            <apply templateURI="/chapter1/footer.zul"/>
        </south>
    </borderlayout>
</zk>
```

- Line 1: This directive links a external style sheet under root folder.
- Line 6, 9, 12, 15: Specify a separated zul path at `templateURI` attribute to inject a page into current page.

For CE users, you still can use `<include>` as an alternative to `<apply>` .

# Apply CSS

In addition to setting a component's attribute, we can also change a component's style by *CSS (Cascading Style Sheet)*. There are two component attributes to apply CSS:

1.  `style` attribute. Like the "style" attribute on an HTML element, you can directly write CSS rules as the attribute's value.

    ```
    <label value="Chapter 1" style="font-weight: bold;"/>
    ```

2.  `sclass` attribute. You should specify a CSS class name as the attribute value.

    ```
    <div sclass="banner">
    ```

To apply a CSS class, you should define it first in a ZUL. There are 2 ways to declare a CSS class.

1.  Inside `<style>` tag.

    ```
    <zk>
     <style>
         .banner {
             background-color:#102d35;
             color: white;
             padding: 5px 10px;
         }
     </style>
     ...
    </zk>
    ```

2.  In an external CSS file and included by `<?link ?>` directive.

It can link to a external style sheet which can be applied to many pages. We use this way in the example to declare CSS.

```
<?link rel="stylesheet" type="text/css" href="/style.css"?>
<zk>
    ...
</zk>
```

# Controlling Components

We can construct the user interface with ZK components but also control them. In this chapter, we continue to use the previous chapter's example, but we will remove the 3 items with hyperlinks in the sidebar and replace them with a redirecting action. To achieve this, we will write code in Java for each item to respond to a user's clicking and redirect the user to an external site.

current:

Improve this Doc

# In Zscript

The simplest way to respond to a user's clicking is to define an event listener method and register it in the `onClick` attribute. We can define an event listener in Java inside a `<zscript>` and those codes will be interpreted when a zul file is loaded. This element also allows other scripting languages such as JavaScript, Ruby, or Groovy. `<zscript>` is very suitable for **fast prototyping**, and it's interpreted when a zul page is created. So after you modify the code inside, you can reload your browser to see the changed result without re-deployment. But it has issues in performance and clustering environment, **we don't recommend to use it in production environment**.

We can declare variables and write statements in `<zscript>` . ZK will run it when loading a zul.

**sidebar-zscript.zul**

```
<grid hflex="1" vflex="1" sclass="sidebar">
    <zscript><![CDATA[
        //zscript code, it runs on server side, use it for fast prototyping
        java.util.Map sites = new java.util.HashMap();

        sites.put("zk","http://www.zkoss.org/");
        sites.put("demo","http://www.zkoss.org/zkdemo");
        sites.put("devref","http://books.zkoss.org/wiki/ZK_Developer's_Reference");
    ]]></zscript>
...
```

- Line 2: It's better to enclose your code with `<![CDATA[ ]]>` .

## Implement an event listener

**sidebar-zscript.zul**

```
<grid hflex="1" vflex="1" sclass="sidebar">
    <zscript><![CDATA[
        ...

        void redirect(String name){
            String loc = sites.get(name);
            if(loc!=null){
                execution.sendRedirect(loc);
            }
        }
    ]]></zscript>
...
```

- Line 5: Define an event listener method like a normal Java method, and it redirects a browser according to the passed variable.
- Line 8: The execution is a implicit variable which you can use it directly without declaration. It's a wrapper object of `HttpServletRequest` .

## Register event listeners at "onClick"

After defining the event listener, we should specify it in a `<row>` 's event attribute `onClick` because we want to invoke the event listener when clicking a `<row>` .

**sidebar-zscript.zul**

```
<grid>
    ...
    <rows>
        <row sclass="sidebar-fn" onClick='redirect("zk")'>
            <image src="/imgs/site.png"/> ZK
        </row>
        <row sclass="sidebar-fn" onClick='redirect("demo")'>
            <image src="/imgs/demo.png"/> ZK Demo
        </row>
        <row sclass="sidebar-fn" onClick='redirect("devref")'>
            <image src="/imgs/doc.png"/> ZK Developer Reference
        </row>
    </rows>
</grid>
```

Now when you click a `<row>` in the sidebar, your browser will be redirected to the corresponding site.

This approach is very simple and fast, so it is especially suitable for building a prototype. However, if you need a better architecture for your application, you had better not use `<zscript>` .

current:

Improve this Doc

# In Controller

In this section, we will demonstrate how to redirect users to an external site with **event listeners** in a **Controller** when they click an item in the sidebar.

# MVC Pattern

A well-known design pattern is **MVC (Model-View-Controller)** which separates an application into 3 roles:

- **Model** is responsible for exposing data while performing business logic which is usually implemented by users.
- **View** is responsible for displaying data which is what ZUL does.
- **Controller** can change the View's presentation and handle events sent from the View.

Such architecture follows SRP (Single Responsibility Principle), and its benefit is that your application is more modularized. Therefore, modifying one part doesn't have to modify another part.

# ZK MVC Approach

By following this pattern, ZK traditionally supports MVC approach which controls components by calling their API. Under ZK context, the relationship of 3 roles looks like:
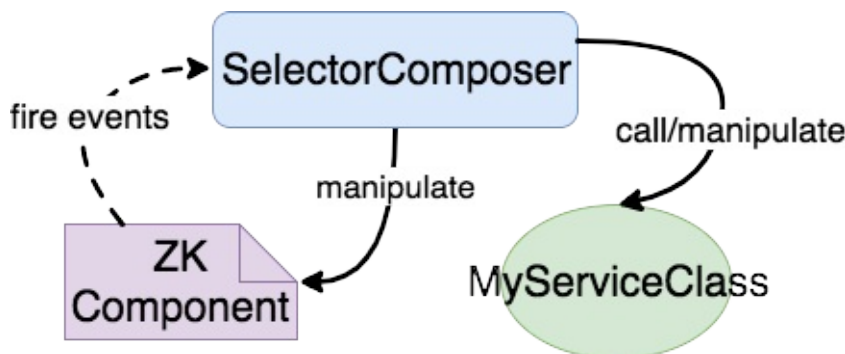


- ZK `SelectorComposer` , that implements `Composer` , plays Controller.
- ZK UI components plays View.
- MyServiceClass is not a real class name. It represents any class which is usually implemented by you performs business logic like searching or authentication.

# Sidebar

Let's start from making the sidebar work.

I build it with `<grid>` and 2 columns, a table-looking component.

```
<grid hflex="1" vflex="1" sclass="sidebar">
    <columns>
        <column width="36px"/>
        <column/>
    </columns>
    <rows>
    </rows>
</grid>
```

In real application, the items in the sidebar are usually built dynamically from a database or upon a user's permission. Hence, I also don't put static `<row>` in the zul, and I will create them dynamically based on data at the server.

When users click a row, they will be brought to the corresponding site.

## Sidebar Page Config

We implement our sidebar configuration including its name, image icon path, and corresponding URL, etc. We will create each `<row>` based on `SidePage`.

```
public class SidebarPageConfigChapter2Impl implements SidebarPageConfig{

    HashMap<String,SidebarPage> pageMap = new LinkedHashMap<String,SidebarPage>();
    public SidebarPageConfigChapter2Impl(){
        pageMap.put("fn1",new SidebarPage("zk","ZK","/imgs/site.png","http://www.zkoss.org/"));
        pageMap.put("fn2",new SidebarPage("demo","ZK Demo","/imgs/demo.png","http://www.zkoss.org/zkdemo"));
        pageMap.put("fn3",new SidebarPage("devref","ZK Developer Reference","/imgs/doc.png","http://books.zkoss
.org/wiki/ZK_Developer's_Reference"));
    }

    ...
}
```

## Create a Controller

In ZK world, a `org.zkoss.zk.ui.util.Composer` plays the same role as the **Controller**, and you can apply it to a target component. Through the composer, you can handle events of the target component (and its children) to change View's presentation according to your requirement. To create a Controller in ZK you simply create a class that inherits `org.zkoss.zk.ui.select.SelectorComposer`.

```
public class SidebarChapter2Controller extends SelectorComposer<Component>{
    //other codes...
}
```

Then "associate" the controller with a component in the zul by specifying fully qualified class name in `apply` attribute. After that the component and all its child components are under the controller's control.

**chapter2/sidebar.zul**

```
<grid apply="org.zkoss.essentials.chapter2.SidebarChapter2Controller"
    hflex="1" vflex="1" sclass="sidebar">
    ...
</grid>
```

# Wire Components

To control a component, we must get its object reference. In `org.zkoss.zk.ui.select.SelectorComposer` , 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. By default `SelectorComposer` locates the component whose ID and type both match the variable name and type respectively in the zul, and `@Wire` also supports selector syntax to wire.

I can specify a component's ID to get wired by default selector in a controller. **chapter2/sidebar.zul**

```
<grid id="sidebar"
    apply="org.zkoss.essentials.chapter2.SidebarChapter2Controller">
```

```
public class SidebarChapter2Controller extends SelectorComposer<Component>{

    //wire components
    @Wire
    private Grid sidebar;

    ...
}
```

`SelectorComposer` looks for a `Grid` whose ID is "sidebar" and assign it to the variable `sidebar` .

# Initialize the View

It is very common that we need to initialize components when a zul file is loaded. In our example, we need to create `<row>` in `<grid>` for the sidebar, therefore we should override a composer life-cycle method `doAfterCompose(Component)` . The passed argument, `comp` , is the component that the controller applies to, which in our example is the `Grid` . ZK will call this method after the applied component, `<grid>` , and its all child components are created, so we can change components' attributes or even create other components in it.

```
public class SidebarChapter2Controller extends SelectorComposer<Component>{

    //wire components
    @Wire
    private Grid sidebar;

    //services
    private SidebarPageConfig pageConfig = new SidebarPageConfigChapter2Impl();

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

        //initialize view after view construction.
        Rows rows = sidebar.getRows();
```

```
        for(SidebarPage page:pageConfig.getPages()){
            Row row = constructSidebarRow(page.getLabel(),page.getIconUri(),page.getUri());
            rows.appendChild(row);
        }
    }
}
```

- Line 8: Here we demonstrate a configurable architecture, the `SidebarPageConfig` stores hyperlink's configuration such as URL, and label and we use this configuration to create and setup components in the sidebar.
- Line 12: You have to call super class `doAfterCompose()` method , because it performs initialization like wiring components for you.
- Line 15 - 20: These codes involve the concept that we have not talked about yet. All you have to know for now is these codes create `<row>` s with event listeners and put them into *Grid*. We will show you the source in the next section.

# Events & Event Listeners

A ZK event ( `org.zkoss.zk.event.Event` ) is an abstraction of an activity made by user, a notification made by an application, and an invocation of server push. For example, a user clicks a button on the browser, this will trigger `onClick` sent to the server. If there is an event listener registered for the button's `onClick` event, ZK will pass the event to the listener to handle it. The event-listener mechanism allows us to handle all user interaction at server side.

# Create Components & Event Listeners Programmatically

Manipulating components is the most powerful feature of ZK. You can change the user interface by creating, removing, or changing components and all changes you made will reflect to browsers.

Now we are going to explain how to create components and add an event listener to respond to users' clicking. Basically, there are 3 steps to create a component:

1. Create a component object.
2. Setup the component's attributes.
3. Append to the target parent component.

In `constructSidebarRow()` method, we create `Row` s and add an event listener to each of them.

```
public class SidebarChapter2Controller extends SelectorComposer<Component>{

    //...

    private Row constructSidebarRow(String name,String label, String imageSrc, final String locationUri) {

        //construct component and hierarchy
        Row row = new Row();
        Image image = new Image(imageSrc);
        Label lab = new Label(label);

        row.appendChild(image);
        row.appendChild(lab);

        //set style attribute
        row.setSclass("sidebar-fn");

        //create and register event listener
        EventListener<Event> actionListener = new SerializableEventListener<Event>() {
            private static final long serialVersionUID = 1L;

            public void onEvent(Event event) throws Exception {
```

```
            //redirect current url to new location
            Executions.getCurrent().sendRedirect(locationUri);
        }
    };

    row.addEventListener(Events.ON_CLICK, actionListener);

    return row;
  }

}
```

- Line 8: The first step to create a component is instantiating its class.
- Line 12: Append a component to establish the parent-child relationship.
- Line 16: You can change a component's attributes by various setter methods and their method names correspond to tag's attribute name.
- Line 19: We create an `EventListener` anonymous class for convenience. Under a clustering environment, your event listener class should implement `org.zkoss.zk.ui.event.SerializableEventListener` .
- Line 24: Implement the business logic in `onEvent()` method, and ZK will call this method when the listened event is sent to the server. Here we get current execution by `org.zkoss.zk.ui.Executions` and redirect a client to a new URL.
- Line 28: Add the event listener to a `<row>` for listening `Events.ON_CLICK` event triggered by a mouse clicking action.

In Line 8 ~ 16, those codes work equally to writing a zul as follows:

```
<row sclass="sidebar-fn">
    <image/><label/>
</row>
```

After completing above steps, when a user clicks a `<row>` on the sidebar, ZK will call a corresponding `actionListener` then the browser will be redirected to a specified URL. You can see the result via /chapter2/index.zul.

current:

Improve this Doc

# MVVM Pattern

In addition to the MVC approach, ZK also allows you to design your application in another architecture: *MVVM (Model-View-ViewModel)*. This architecture also divides an application into 3 roles: **View**, **Model**, and **ViewModel**.

- **Model** is responsible for exposing data while performing business logic which is usually implemented by users.
- **View** is responsible for displaying data which is what ZUL does. (The View and Model plays the same roles as they do in MVC)
- **ViewModel** acts like a special Controller which is responsible for exposing data from the Model to the View and providing required actions and logic for user requests from the View.

The ViewModel is a View abstraction, which contains a View's state and behavior. The biggest difference from the Controller in the MVC is that **ViewModel does not contain any reference to UI components** and knows nothing about the View's visual elements. There should be a role that synchronizes data and handle events and response between View and ViewModel. This role eliminates the dependency between View and ViewModel. Hence, this clear separation between View and ViewModel decouples ViewModel from View and makes ViewModel more reusable and more abstract.

# ZK MVVM Approach

By following this pattern, ZK supports MVVM approach which controls components by the data binding. Under ZK context, the relationship of 3 roles looks like:

- ViewModel, it's just a POJO (plain ordinary Java object) that doesn't need to extend any parent class, neither implements any interface.
- ZK UI components plays View.
- MyServiceClass is not a real class name. It represents any class which is usually implemented by you performs business logic like searching or authentication.
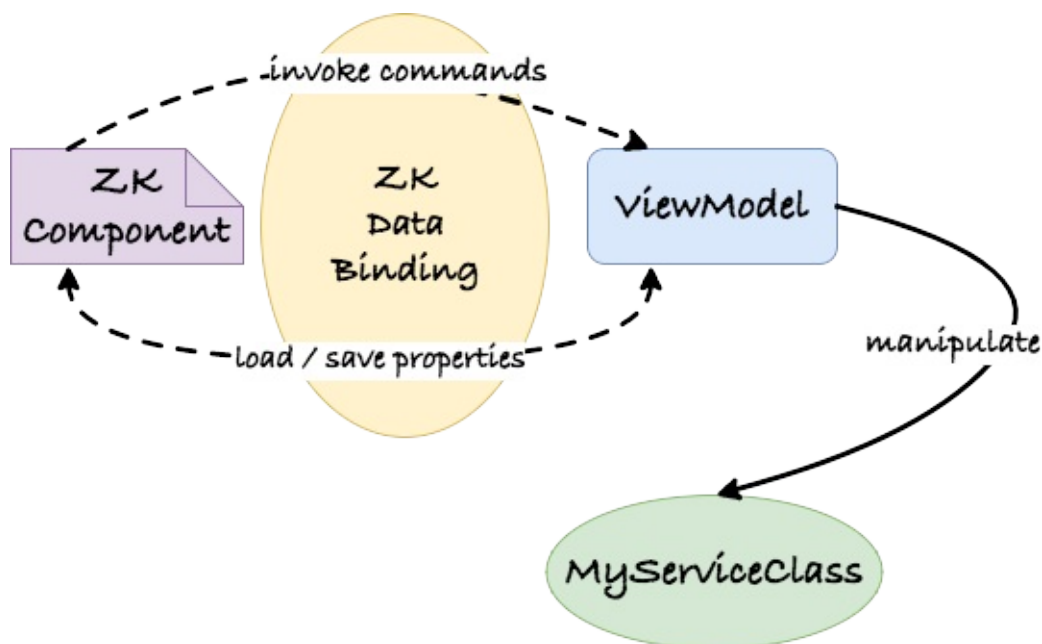
Since the ViewModel contains no reference to UI components, you cannot control components directly by their setter and getter. Therefore, ZK supports a data binding mechanism, ZK Bind, to synchronize data and handle events and respond between the View and ViewModel. Additionally, this mechanism also has to bridge events from the View to the action provided by the ViewModel. In this binding system, the **binder** plays the key role to operate the whole mechanism. The binder is like a broker and responsible for communication between View and ViewModel.

In brief, MVVM approach allows you to **control components with data binding** (not by API).

# Sidebar

Let's start from making the sidebar work.



I build it with `<grid>` and 2 columns, a table-looking component.

```
<grid hflex="1" vflex="1" sclass="sidebar">
    <columns>
        <column width="36px"/>
        <column/>
    </columns>
    <rows>
    </rows>
</grid>
```

In real application, the items in the sidebar are usually built dynamically from a database or upon a user's permission. Hence, I also don't put static `<row>` in the zul, and I will create them dynamically based on data at the server.

When users click a row, they will be brought to the corresponding site.

# Sidebar Page Config

We implement our sidebar configuration including its name, image icon path, and corresponding URL, etc. We will create each `<row>` based on `SidePage` .

```
public class SidebarPageConfigChapter2Impl implements SidebarPageConfig{

    HashMap<String,SidebarPage> pageMap = new LinkedHashMap<String,SidebarPage>();
    public SidebarPageConfigChapter2Impl(){
        pageMap.put("fn1",new SidebarPage("zk","ZK","/imgs/site.png","http://www.zkoss.org/"));
        pageMap.put("fn2",new SidebarPage("demo","ZK Demo","/imgs/demo.png","http://www.zkoss.org/zkdemo"));
        pageMap.put("fn3",new SidebarPage("devref","ZK Developer Reference","/imgs/doc.png","http://books.zkoss
.org/wiki/ZK_Developer's_Reference"));
    }

    ...
}
```

# Create a ViewModel

VieModel contains the state of the page, so we have to analysis what data to show in the page. Since I need to build sidebar rows, the only data we need is `SidebarPageConfig` . Consequently, we instantiate a `SidebarPageConfigChapter2Impl` and expose its `PageConfig` list by getter, `getSidebarPages()` ;

```java
public class SidebarViewModel {

    private SidebarPageConfig pageConfig = new SidebarPageConfigChapter2Impl();

    public List<SidebarPage> getSidebarPages() {
        return pageConfig.getPages();
    }

}
```

# Apply a ViewModel on a Component

To apply a ViewModel, we need to bind a ZK component to the ViewModel by setting its `viewModel` attribute [1] with the ViewModel's id in `@id` and the ViewModel's full-qualified class name in `@init` like:

```
<div viewModel="@id('vm')@init('foo.MyViewModel')">
```

The id is like a variable and we access the ViewModel's properties by the ID, e.g. `vm.name` . Whilst the full-qualified class name is used to instantiate the ViewModel object itself. So the component that a ViewModel is bound to becomes the **Root View Component** for the ViewModel. All child components of this Root View Component are bound to the same ViewModel and its properties, so we usually bind a page's root component to a ViewModel.

Apply the ViewModel like:

**chapter2/sidebar-mvvm.zul**

```
<grid viewModel="@id('vm') @init('org.zkoss.essentials.chapter2.SidebarViewModel')">
```

[1] Since ZK 8, you don't need to specify `apply="org.zkoss.bind.BindComposer"` explicitly. Because ZK implicitly applies `BindComposer` for you if you specify a ViewModel.

# Binding Data with ViewModel's Properties

Now that ViewModel is prepared and bound to a component, we can bind a component's attributes to the ViewModel's property. Once the binding is established, ZK will synchronize (load or save) data between components and the ViewModel for us automatically.

# Relationship among a Component, Model, and Template

In ZK, all data components are designed to accept a separate model object that contains data to be rendered, and the component renders the data model upon a template (what you specify inside `<template>` ).
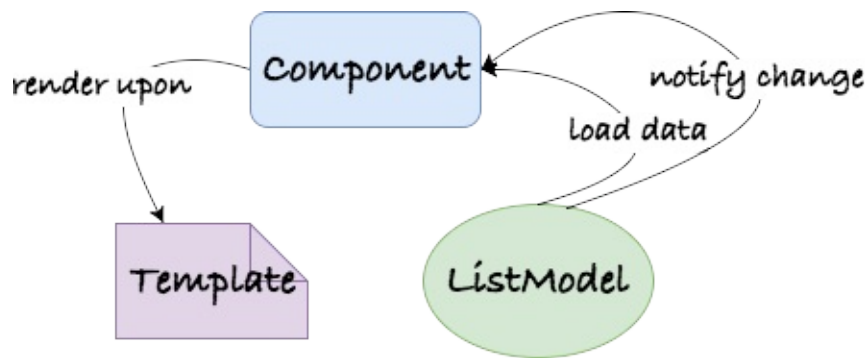
This design keeps each part in its single responsibility, so that increases their reusability and decouples the data from a component's implementation.

# Render a list of sidebar pages

`<grid>` is a component that can accept a `ListModel` object (or Java Collection) and renders its `<row>` based on it. Since `SidebarViewModel` has exposed `List<SidebarPage>` via getter method, it is the data model and we can bind it to `model` attribute like:

**chapter2/sidebar-mvvm.zul**

```
<grid viewModel="@id('vm') @init('org.zkoss.essentials.chapter2.SidebarViewModel')"
    ...
    model="@load(vm.sidebarPages)" >
```

## Define Listbox Template

The last part is to define a template, so that `<grid>` can know how to render its `List<SidebarPage>` with `<row>` . If you don't define it, `<grid>` renders the model with a default built-in template.

```
<grid viewModel="@id('vm') @init('org.zkoss.essentials.chapter2.SidebarViewModel')"
    ...
    model="@load(vm.sidebarPages)" >
    ...
    <rows>
        <template name="model">
            <row sclass="sidebar-fn" >
                <image src="@load(each.iconUri)"/>
                <label value="@load(each.label)"/>
            </row>
        </template>
    </rows>
</grid>
```

- Line 6: The `name` attribute has to be **model** which means it's [a template for `<listbox>` model](#).

## implicit `each`

The variable `each` is an implicit variable that you can use without declaration inside `<template>` , and it represents one object ( `SidebarPage` ) of the data model for each iteration when rendering. We use `each` with dot notation to reference a data object's property. In each `<row>` , we show an image icon by loading a URL and a label.

# Define Commands

ViewModel also contains View's application logic which are implemented by methods. We call such a method "Command" of the ViewModel. These methods usually manipulate data in the ViewModel, for example adding or deleting an item. The View's behaviors are usually triggered by events from the View. The data binding mechanism also supports binding an event to a ViewModel's command. Firing the component's event will trigger the execution of bound command that means invoking the corresponding command method.

In ZK, to declare a command method in a ViewModel, you should apply annotation `@Command` to a method. You could specify a command name which is the method's name by default if no specified.

Our sidebar just has one behavior: redirect a browser to the corresponding URL. In order to get the URL, we need to accept `SidebarPage` as a parameter with `@BindingParam` :

```java
public class SidebarViewModel {
    ...
    @Command
    public void navigate(@BindingParam("page") SidebarPage page) {
        Executions.getCurrent().sendRedirect(page.getUri());
    }
}
```

Line 5: `Executions.getCurrent()` ) returns the current `Execution` which is a wrapper of `HttpServletRequest` . `sendRedirect()` ) will redirect a browser to the specified URL.

I apply `@BindingParam("page")` in front of the Parameter which means I will pass `SidebarPage` object with a command binding in a zul with the key `page` .

# Handle User Interactions by Command Binding

After we finish binding attributes to the ViewModel's data, we still need to handle user actions, e.g. clicking. Under the MVVM approach, we handle events by binding an event attribute (e.g. `onClick` ) to a **Command** of a ViewModel like:

```
onClick="@command('mycmd')"
```

After we bind an event to a Command, each time the event is fired, ZK will invoke the corresponding command method.

**chapter2/sidebar-mvvm.zul**

```xml
<rows>
    <template name="model">
        <row sclass="sidebar-fn" onClick="@command('navigate', page=each)">
            ...
        </row>
    </template>
</rows>
```

Line 3: By default the command name is the method name `navigate` . We need to pass the `SidebarPage` with key `page` to the command method with the implicit variable `each` .

After above steps, the sidebar works as expected.

# Target Application

In this chapter, we will demonstrate a common scenario: collecting user input in form style page. The target application looks as follows:



It is a personal profile form with 5 different fields. Clicking the "Save" button saves the user's data and clicking the "Reload" button loads previous saved data back into the form.

Starting from this chapter, we will show how to implement an example application using both the MVC and MVVM approaches. If you are not familiar with these two approaches, we suggest that you read Get ZK Up and Running with MVC and Get ZK Up and Running with MVVM. These two approaches are mutually interchangeable. You can choose one of them depending on your situation. Please refer to Approach Comparison.

# MVC Approach

Under this approach, we implement the event handling and presentation logic in a controller with no code present in the ZUL file. This approach makes the responsibility of each role (Model, View, and Controller) more cohesive and allows you to control components directly. It is very intuitive and very flexible.

# Construct a Form Style Page

With the concept and technique we talked about in the previous chapter, it should be easy to construct a form style user interface as follows. We uses a two-column `<grid>` to build the form style layout and different input components to receive user's profile like name and birthday. The zul file below is included in the `<center>` of `<borderlayout>` .

We build the frame of this form first:

**chapter3/profile-mvc.zul**

```
<?link rel="stylesheet" type="text/css" href="/style.css"?>
<window
    border="normal" hflex="1" vflex="1" contentStyle="overflow:auto">
    <caption src="/imgs/profile.png" sclass="fn-caption"
        label="Profile (MVC)"/>
    <vlayout>
        <grid width="500px">
            ...
        </grid>
        <div>You are editing <label id="nameLabel"/>'s profile.</div>
        <hlayout>
            <button id="saveProfile" label="Save"/>
            <button id="reloadProfile" label="Reload"/>
        </hlayout>
    </vlayout>
</window>
```

- Line 4, 5: `<caption>` can be used to build compound header with an image for a `<window>` .
- Line 6: `<vlayout>` is a light-weight layout component which arranges its child components vertically without splitter, align, and pack support.
- Line 11: `<hlayout>` , like `<vlayout>` , but arranges its child components horizontally.

## A Grid Makes a Form

Then let's put components in a Grid to arrange them as a form style. A `<grid>` is basically composed by `<columns>` and `<rows>` :

```
<grid>
    <columns>

    </columns>
    <rows>

    </rows>
</grid>
```

- `<columns` > can have `<column>` (no 's'), and `<rows>` can have `<row>` (no 's').

**chapter3/profile-mvc.zul**

```
        <grid width="500px">
            <columns>
                <column align="right" hflex="min"/>
                <column/>
```

```
            </columns>
            <rows>
                <row>
                    <cell sclass="row-title">Account :</cell>
                    <cell><label id="account"/></cell>
                </row>
                <row>
                    <cell sclass="row-title">Full Name :</cell>
                    <cell>
                    <textbox id="fullName" width="200px"/>
                    </cell>
                </row>
                <row>
                    <cell sclass="row-title">Email :</cell>
                    <cell>
                    <textbox id="email" width="200px"/>
                    </cell>
                </row>
                <row>
                    <cell sclass="row-title">Birthday :</cell>
                    <cell>
                        <datebox id="birthday" width="200px"/>
                    </cell>
                </row>
                <row>
                    <cell sclass="row-title">Country :</cell>
                    <cell>
                        ...
                    </cell>
                </row>
                <row>
                    <cell sclass="row-title">Bio :</cell>
                    <cell><textbox id="bio" multiline="true"
                        hflex="1" height="200px" />
                    </cell>
                </row>
            </rows>
        </grid>
```

- Line 3: `hflex="min"` can limit the column's width just wider enough to hold each row's content without a line break.
- Line 8: `<cell>` is used inside `<row>` , `<hbox>` , or `<vbox>` to fully control a column's align, row/column span, and width in an individual row.

# User Input Validation

Each ZK input component provides a built-in input validation by `constraint` attribute. You can specify a pre-defined constraint rule to activate it, then the validation works without writing any code in a controller. For example:

```
<textbox id="fullName" constraint="no empty: Please enter your full name"
        width="200px"/>
```

- The constraint rule means "no empty value allowed" for the `<textbox>` . If the user input violates this rule, ZK will show the message after a colon.

```
<textbox id="email"
        constraint="/.+@.+\.[a-z]+/: Please enter an e-mail address"
        width="200px"/>
```

- We can also define a constraint rule using a regular expression that describes the email format to limit the value in correct format.

```
<datebox id="birthday" constraint="no future" width="200px"/>
```

- The constraint rule means "date in the future is not allowed" and it also restricts the available date to choose.

Then, the input component will show the specified error message when an input value violates a specified constraint rule.

# Initialize Input Components

When a user visits this page, we want profile data to be loaded in the form and ready to be modified. Hence, we should initialize those input components in a controller by loading previously-saved data to input components.

To manipulate components, we need to get their object reference by `@Wire` .

```java
public class ProfileViewController extends SelectorComposer<Component>{

    //wire components
    @Wire
    Label account;
    @Wire
    Textbox fullName;
    @Wire
    Textbox email;
    @Wire
    Datebox birthday;
    @Wire
    Listbox country;
    @Wire
    Textbox bio;

    //services
    AuthenticationService authService = new AuthenticationServiceChapter3Impl();
    UserInfoService userInfoService = new UserInfoServiceChapter3Impl();
    ...
}
```

- Line 19: A controller usually calls service classes to perform business operations or get necessary data.

To make sure all components objects are wired, we initialize them in `doAfterCompose()` since ZK will call this method after it creates all its child components, so that you can manipulate those children safely.

```java
public class ProfileViewController extends SelectorComposer<Component>{

    ...

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

        ListModelList<String> countryModel = new ListModelList<String>(CommonInfoService.getCountryList());
        country.setModel(countryModel);

        refreshProfileView();
    }

    ...

    private void refreshProfileView() {
        UserCredential userCredential = authService.getUserCredential();
        User user = userInfoService.findUser(userCredential.getAccount());
        if(user==null){
            //TODO handle un-authenticated access
            return;
        }

        //apply bean value to UI components
        account.setValue(user.getAccount());
```

```
        fullName.setValue(user.getFullName());
        email.setValue(user.getEmail());
        birthday.setValue(user.getBirthday());
        bio.setValue(user.getBio());

        ((ListModelList)country.getModel()).addToSelection(user.getCountry());
        ...
    }
}
```

- Line 12: Load previously-saved data to input components to initialize UI, so we should call it after initializing country list.
- Line 26-30: Populate saved user data to components by `setValue()`.
- Line 32: set the `Listbox`'s selected item with `ListModelList.addToSelection()`.

# Populate Country Drop-down List

This form needs a drop-down list that contains a list of countries. When a user visits the page, the data in drop-down list should be ready. To achieve this, we have to initialize a drop-down list in the controller.
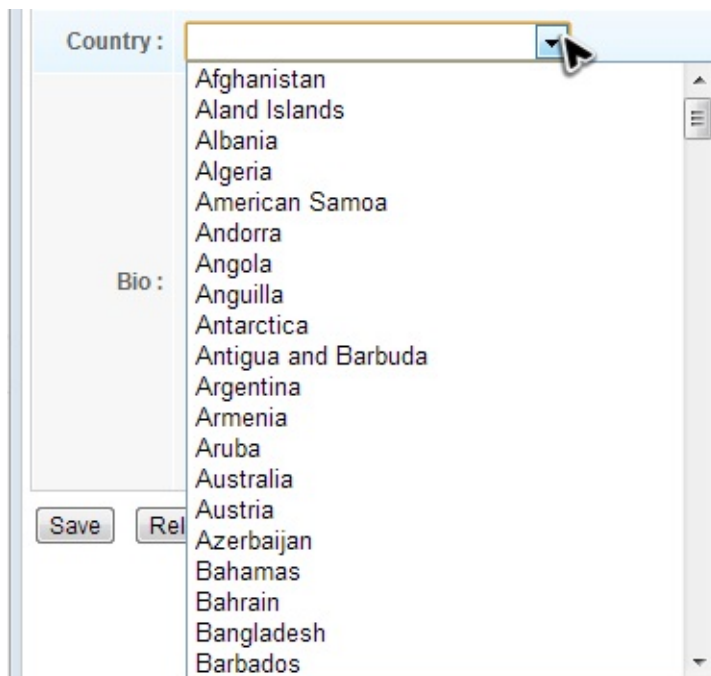


By setting `<listbox>` in "select" mold, We will have a drop-down list instead of a table-like component on the page.

```
<listbox id="country" mold="select" width="200px">
```

A component could have multiple different visual appearances. Each appearance is called a "mold". Therefore, you can choose a proper mold according to your visual requirement/page design.

# Relationship among a Component, Model, and Template

In ZK, all data components are designed to accept a separate model object that contains data to be rendered, and the component renders the data model upon a template (what you specify inside `<template>`).
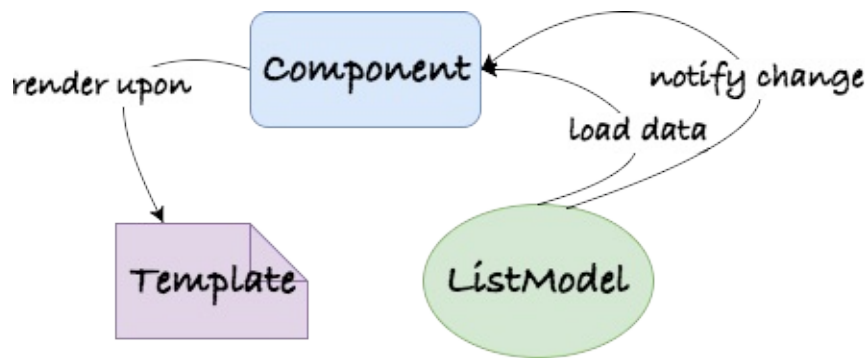
This design keeps each part in its single responsibility, so that increases their reusability and decouples the data from a component's implementation.

# Create a Data Model

Just a `<listbox>` in a zul doesn't provide any country to select. We need create a data model object.

```java
public class ProfileViewController extends SelectorComposer<Component>{
    ...
    @Wire
    Listbox country;

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

        ListModelList<String> countryModel = new ListModelList<String>(CommonInfoService.getCountryList());
        country.setModel(countryModel);

        ...
    }

    ...

}
```

- Line 10: Create a `ListModelList` object with a list of `String`
- Line 11: Provide prepared data model object to the component by `setModel()` .

# Define Listbox Template

The last part is to define a template, so that `<listbox>` can know how to render its data model. If you don't define it, `<listbox>` renders the model with a default built-in template.

```xml
<listbox id="country" mold="select" width="200px">
    <template name="model">
        <listitem label="${each}" />
    </template>
</listbox>
```

- Line 2: The `name` attribute has to be **model** which means it's a template for `<listbox>` model.
- Line 3: The `${each}` is an implicit variable that you can use without declaration inside `<template>` , and it represents one object of the data model for each iteration when rendering. We use this variable with dot notation at component attributes to reference a data object's property . In our example, we just set it at `<listitem>` 's label.

# Populate Input Components

When a user visits this page, we want profile data to be loaded in the form and ready to be modified. Hence, we should initialize those input components in a controller by loading previously-saved data to input components.

To manipulate components, we need to get their object reference by `@Wire`.

```java
public class ProfileViewController extends SelectorComposer<Component>{

    //wire components
    @Wire
    Label account;
    @Wire
    Textbox fullName;
    @Wire
    Textbox email;
    @Wire
    Datebox birthday;
    @Wire
    Listbox country;
    @Wire
    Textbox bio;

    //services
    AuthenticationService authService = new AuthenticationServiceChapter3Impl();
    UserInfoService userInfoService = new UserInfoServiceChapter3Impl();
    ...
```

- Line 19: A controller usually calls service classes to perform business operations or get necessary data.

To make sure all components objects are wired, we initialize them in `doAfterCompose()`.

```java
public class ProfileViewController extends SelectorComposer<Component>{

    ...

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

        ListModelList<String> countryModel = new ListModelList<String>(CommonInfoService.getCountryList());
        country.setModel(countryModel);

        refreshProfileView();
    }

    ...

    private void refreshProfileView() {
        UserCredential userCredential = authService.getUserCredential();
        User user = userInfoService.findUser(userCredential.getAccount());
        if(user==null){
            //TODO handle un-authenticated access
            return;
        }

        //apply bean value to UI components
        account.setValue(user.getAccount());
        fullName.setValue(user.getFullName());
        email.setValue(user.getEmail());
        birthday.setValue(user.getBirthday());
        bio.setValue(user.getBio());

        ((ListModelList)country.getModel()).addToSelection(user.getCountry());
```

```
        ...
    }
}
```

- Line 13: Load previously-saved data to input components to initialize UI, so we should call it after initializing country list.
- Line 27-31: Push saved user data to components by `setValue()` .
- Line 33: Use `ListModelList.addToSelection()` to set the `Listbox` 's selected item.

Improve this Doc

# Save & Reload Data

The example application has 2 functions, save and reload, which are both triggered by clicking a button. If you click the "Save" button, the application will save your input and show a notification box.



**Click "Save" button**

In this section, we will demonstrate a more flexible way to define an event listener in a controller with `@Listen` other than calling `addEventListener()` .

An event listener method should be public, have a void return type, and have either no parameter or one parameter of the specific event type (corresponding to the event listened) with `@Listen` in a controller. You should specify event listening rule in the annotation's element value. Then ZK will "wire" the method to the specified components for specified events. ZK provides various wiring selectors to specify in the annotation.

## Listen "Save" button Clicking

```
public class ProfileViewController extends SelectorComposer<Component>{

    @Listen("onClick=#saveProfile")
    public void doSaveProfile(){
```

```
        ...
    }
    ...
}
```

- Line 3: The `@Listen` will make `doSaveProfile()` be invoked when a user clicks a component ( `onClick` ) whose id is "saveProfile" ( `#saveProfile` ).

We can manipulate components to change the UI in the event listener. In `doSaveProfile()` , we get a user's input from input components and save the data to a `User` object. Then show a notification to the client.

## Handle "Save" Button Clicking

```
public class ProfileViewController extends SelectorComposer<Component>{
....

    @Listen("onClick=#saveProfile")
    public void doSaveProfile(){
        UserCredential userCredential = authService.getUserCredential();
        User user = userInfoService.findUser(userCredential.getAccount());
        if(user==null){
            //TODO handle un-authenticated access
            return;
        }

        //apply component value to bean
        user.setFullName(fullName.getValue());
        user.setEmail(email.getValue());
        user.setBirthday(birthday.getValue());
        user.setBio(bio.getValue());

        Set<String> selection = ((ListModelList)country.getModel()).getSelection();
        if(!selection.isEmpty()){
            user.setCountry(selection.iterator().next());
        }else{
            user.setCountry(null);
        }

        userInfoService.updateUser(user);

        Clients.showNotification("Your profile is updated");
    }
    ...
}
```

- Line 7: In this chapter's example, `UserCredential` is initialized with "Anonymous". We will put a user data in chapter 8.
- Line 14: Get users input by calling `getValue()` .
- Line 19: Get a user's selection for a `Listbox` from its model object.
- Line 28: Show a notification box which is the most easy way to show a message to users.

## Handle "Reload" Button Clicking

To wire the event listener for "Reload" button's is similar as the previous one, but we use a different selector this time. And the method pushes saved user data to components using `setValue()` , so we can just call previously-implemented `refreshProfileView()` .

```
public class ProfileViewController extends SelectorComposer<Component>{

    ...
```

```
    @Listen("onClick = button[label = 'Reload']")
    public void doReloadProfile(){
        refreshProfileView();
    }

    ...
}
```

- Line 6: This method is listed in the previous section.

After the above steps, we have finished all functions of the target application. Quite simple, right? Please run /chapter3/index.zul to see the result.

```
    @Listen("onClick = button[label = 'Reload']")
    public void doReloadProfile(){
        refreshProfileView();
    }
```

current:

Improve this Doc

# MVVM Approach

This section we will demonstrate how to implement a form style page and handle user input under MVVM approach.

Improve this Doc

# Construct a Form Style Page

Building a user interface using the MVVM approach is not different from the MVC approach.

**Extracted from chapter3/profile-mvvm-property.zul**

```
<?link rel="stylesheet" type="text/css" href="/style.css"?>
<window border="normal" hflex="1" vflex="1" contentStyle="overflow:auto">
    <caption src="/imgs/profile.png" sclass="fn-caption"
             label="Profile (MVVM)"/>
    <vlayout>
        <grid width="500px" >
            <columns>
                <column align="right" hflex="min"/>
                <column/>
            </columns>
            <rows>
                <row>
                    <cell sclass="row-title">Account :</cell>
                    <cell><label/></cell>
                </row>
                <row>
                    <cell sclass="row-title">Full Name :</cell>
                    <cell>
                    <textbox
                             width="200px"/>
                    </cell>
                </row>
                <row>
                    <cell sclass="row-title">Email :</cell>
                    <cell>
                    <textbox
                    width="200px"/>
                    </cell>
                </row>
                <row>
                    <cell sclass="row-title">Birthday :</cell>
                    <cell><datebox  width="200px"/>
                    </cell>
                </row>
                <row>
                    <cell sclass="row-title">Country :</cell>
                    <cell>
                        <listbox  mold="select" width="200px">
                        </listbox>
                    </cell>
                </row>
                <row>
                    <cell sclass="row-title">Bio :</cell>
                    <cell>
                    <textbox
                    multiline="true" hflex="1" height="200px" />
                    </cell>
                </row>
            </rows>
        </grid>
        <div>You are editing <label />'s profile.</div>
        <hlayout>
            <button  label="Save"/>
            <button  label="Reload"/>
        </hlayout>
    </vlayout>
```

```
</window>
```

# User Input Validation

Each ZK input component provides a built-in input validation by `constraint` attribute. You can specify a pre-defined constraint rule to activate it, then the validation works without writing any code in a controller. For example:

```
<textbox id="fullName" constraint="no empty: Please enter your full name"
        width="200px"/>
```

- The constraint rule means "no empty value allowed" for the `<textbox>` . If the user input violates this rule, ZK will show the message after a colon.

```
<textbox id="email"
        constraint="/.+@.+\.[a-z]+/: Please enter an e-mail address"
        width="200px"/>
```

- We can also define a constraint rule using a regular expression that describes the email format to limit the value in correct format.

```
<datebox id="birthday" constraint="no future" width="200px"/>
```

- The constraint rule means "date in the future is not allowed" and it also restricts the available date to choose.

Then, the input component will show the specified error message when an input value violates a specified constraint rule.

current:

Improve this Doc

# Create a ViewModel

ViewModel is an abstraction of View which contains the View's data, state and behavior. 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. So that you can think ViewModel is a "Model of the View". It contains the View's state (e.g. user's selection, whether a component is enabled or disabled) that might change during user interaction.

In ZK, a ViewModel can simply be a POJO which contains data to display on the ZUL and doesn't have any components. The example application displays 2 kinds of data:

- the user's profile
- country list

The ViewModel should look like the following:

```java
public class ProfileViewModel implements Serializable{

    //services
    private AuthenticationService authService = new AuthenticationServiceChapter3Impl();
    private UserInfoService userInfoService = new UserInfoServiceChapter3Impl();

    //data for the view
    private User currentUser;

    public User getCurrentUser(){
        return currentUser;
    }

    public List<String> getCountryList(){
        return CommonInfoService.getCountryList();
    }
}
```

- Line 4,5: The ViewModel usually contains service classes that are used to get data from them or perform business logic.
- Line 8, 10: We should declare current user profile and its getter method to be displayed in the zul.
- Line 14: ViewModel exposes its data by getter methods, it doesn't have to define a corresponding member variable. Hence we can expose country list by getting from the service class.

## Initialize a ViewModel

Since the ViewModel is just a POJO, we can initialize its member fields in a constructor.

```java
public class ProfileViewModel implements Serializable{
...
    public ProfileViewModel(){
        UserCredential userCredential = authService.getUserCredential();
        currentUser = userInfoService.findUser(userCredential.getAccount());
        if(currentUser==null){
            //TODO handle un-authenticated access
            return;
        }
    }
}
```

If you need ZK context object like `Desktop` at initialization, please refer to `@Init` .

# Apply a ViewModel on a Component

To apply a ViewModel, we need to bind a ZK component to the ViewModel by setting its `viewModel` attribute [1] with the ViewModel's id in `@id` and the ViewModel's full-qualified class name in `@init` like:

```
<div viewModel="@id('vm')@init('foo.MyViewModel')">
```

The id is like a variable and we access the ViewModel's properties by the ID, e.g. `vm.name` . Whilst the full-qualified class name is used to instantiate the ViewModel object itself. So the component that a ViewModel is bound to becomes the **Root View Component** for the ViewModel. All child components of this Root View Component are bound to the same ViewModel and its properties, so we usually bind a page's root component to a ViewModel.

```
<window viewModel="@id('vm') @init('org.zkoss.essentials.chapter3.mvvm.ProfileViewModel')"
    border="normal" hflex="1" vflex="1" contentStyle="overflow:auto">
...
</window>
```

- Line 1: Specify ViewModel's id with `@id` and the its full-qualified class name in `@init` .

[1] Since ZK 8, you don't need to specify `apply="org.zkoss.bind.BindComposer"` explicitly. Because ZK implicitly applies `BindComposer` for you if you specify a ViewModel.

# Binding Data with ViewModel's Properties

Now that ViewModel is prepared and bound to a component, we can bind a component's attributes to the ViewModel's property. The binding between an attribute and a ViewModel's property is called **property binding**. Once the binding is established, ZK will synchronize (load or save) data between components and the ViewModel for us automatically.



## Load a User

For the first row of this form, we want to show the user name, then we can load `User` 's `account` property to a `<label>` `value` attribute with data binding syntax `@Load` :

```
...
    <rows>
        <row>
            <cell sclass="row-title">Account :</cell>
            <cell>
```

```
                <label value="@load(vm.currentUser.account)"/>
            </cell>
        </row>
        ...
    </rows>
```

- Line 5: `vm` is the ViewModel's id. We use "dot notation" to access an object's properties. Then ZK actually calls getter for you, hence, `vm.currentUser.account` will invoke `getCurrentUser().getAccount()`.

# Relationship among a Component, Model, and Template

In ZK, all data components are designed to accept a separate model object that contains data to be rendered, and the component renders the data model upon a template (what you specify inside `<template>` ).
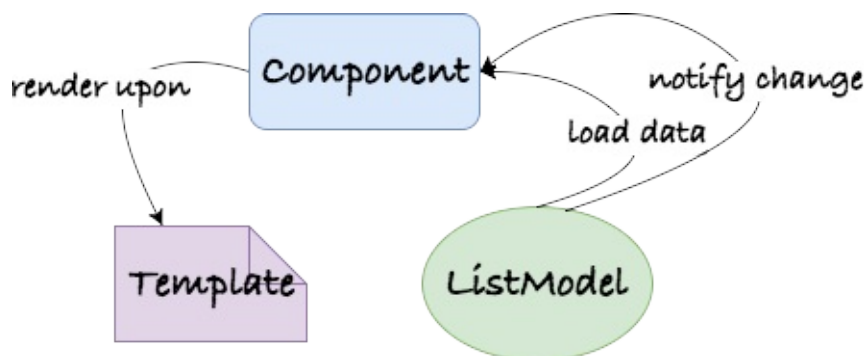


This design keeps each part in its single responsibility, so that increases their reusability and decouples the data from a component's implementation.

# Load a Collection Object, Country List

This form needs a drop-down list that contains a list of countries. When a user visits the page, the data in drop-down list should be ready. To achieve this, we have to load the country list from the ViewModel.
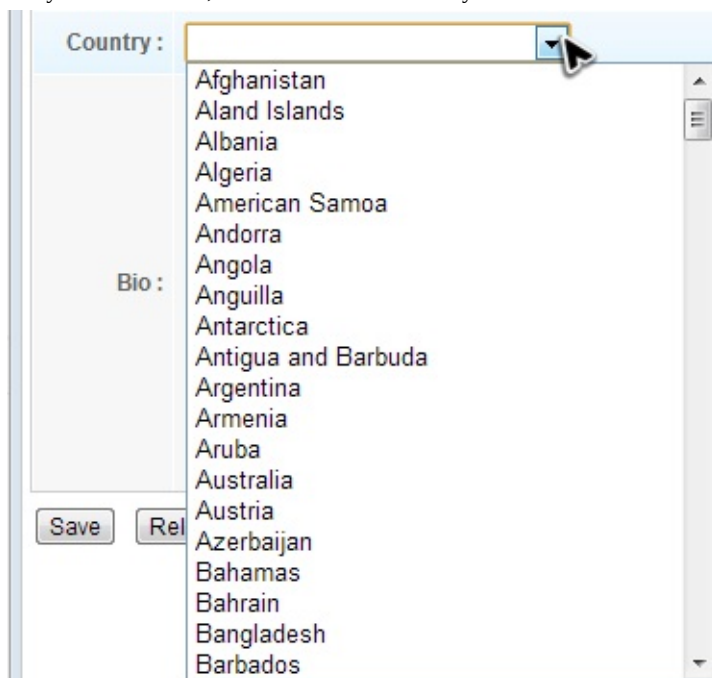


In order to provide a dropdown list, we put a `<listbox>` in a `select` mold.

```
    <cell>
        <listbox mold="select" width="200px">
        </listbox>
    </cell>
```

## Load a Data Model

Our ViewModel already returns a countryList. You might find `getCountryList()` return a `List` instead of a `ListModelList` , but don't worry. ZK will convert it automatically. To make the countryList as a data model of `<listbox>` , we have to bind it at `model` attribute:

```
    <cell>
        <listbox model="@load(vm.countryList)" mold="select" width="200px">
        </listbox>
    </cell>
```

## Define Listbox Template

The last part is to define a template, so that `<listbox>` can know how to render its data model with `<listitem>` . If you don't define it, `<listbox>` renders the model with a default built-in template.

```
    <cell>
        <listbox model="@load(vm.countryList)" mold="select" width="200px">
            <template name="model">
                <listitem label="@load(each)" />
            </template>
        </listbox>
    </cell>
 ...
```

- Line 3: The `name` attribute has to be **model** which means it's a template for `<listbox>` model.
- Line 4: The `each` is an implicit variable that you can use without declaration inside `<template>` , and it represents one object of the data model for each iteration when rendering. We use this variable with dot notation at component attributes to reference a data object's property . In our example, we just set it at `<listitem>` 's label.

After above steps, you can see a list of country in the form.

## Load User's Country as an Selected Item

To load a user's country as an selected item of Listbox, you need to bind the property to `selectedItem` :

```
 <listbox model="@load(vm.countryList)" selectedItem="@load(vm.currentUser.country)" mold="select" width="200px">
```
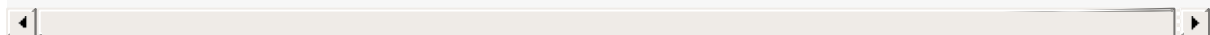
# Get User Input

In MVC pattern, we have to call an input component's getter (e.g. `getValue()`) to collect user input. But in MVVM pattern, after you specify `@save`, ZK can save user input back to a ViewModel automatically. For example in the below zul, user input is saved automatically when you move the focus out of the *Textbox*.

```
<textbox value="@save(vm.currentUser.fullName)" .../>
```

For the property `currentUser`, we want to both save user input back to the ViewModel and load value from the ViewModel, so the code becomes:

```
<textbox value="@load(vm.currentUser.fullName)@save(vm.currentUser.fullName)" .../>
```

The syntax is quite long. ZK knows your pain and provides a shortcut syntax: `@bind`:

```
<textbox value="@bind(vm.currentUser.fullName)" .../>
```

Therefore, we can save and load other `currentUser` property with `@bind` like:

```
...
    <rows>
        ...
        <row>
            ...
            <cell>
                <textbox value="@bind(vm.currentUser.fullName)"
                    constraint="no empty: Plean enter your full name"
                        width="200px"/>
            </cell>
        </row>
        <row>
            ...
            <cell>
                <textbox value="@bind(vm.currentUser.email)"
                constraint="/.+@.+\.[a-z]+/: Please enter an e-mail address"
                    width="200px"/>
            </cell>
        </row>
        <row>
            ...
            <cell>
                <datebox value="@bind(vm.currentUser.birthday)"
                    constraint="no future" width="200px"/>
            </cell>
        </row>
    ...
        <row>
            ...
            <cell>
                <textbox value="@bind(vm.currentUser.bio)"
                    multiline="true" hflex="1" height="200px" />
            </cell>
        </row>
    </rows>

    ...
```

# Save User Selection

Bind `selectedItem` to `vm.currentUser.country` and the selected country will be saved to `currentUser` when users select an item from the drop-down list.

```
<listbox model="@load(vm.countryList)" selectedItem="@bind(vm.currentUser.country)" ...>
```

# Define Commands

ViewModel also contains View's behaviors which are implemented by methods. We call such a method "Command" of the ViewModel. These methods usually manipulate data in the ViewModel, for example deleting an item. The View's behaviors are usually triggered by events from the View. The Data binding mechanism also supports binding an event to a ViewModel's command. Firing the component's event will trigger the execution of bound command that means invoking the corresponding command method.

For ZK to recognize a command method in a ViewModel, you should apply annotation `@Command` to a command method. You could specify a command name which is the method's name by default if no specified. Our example has two behavior: "save" and "reload", so we define two command methods for each of them:

**Define commands in a ViewModel**

```java
public class ProfileViewModel implements Serializable{
    ...

    @Command //@Command annotates a command method
    public void save(){
        currentUser = userInfoService.updateUser(currentUser);
        Clients.showNotification("Your profile is updated");
    }

    @Command
    public void reload(){
        UserCredential cre = authService.getUserCredential();
        currentUser = userInfoService.findUser(cre.getAccount());
    }
}
```

- Line 4, 10: Annotate a method with `@Command` to make it become a command method, and it can be bound with data binding in a zul.
- Line 5: Method name is the default command name if you don't specify in `@Command`. This method save the `currentUser` with a service class and show a notification.

During execution of a command, one or more properties may be changed due to performing business or presentation logic. Developers have to specify which property (or properties) is changed, then the data binding mechanism can reload them to synchronize the View to the latest state.

The syntax to notify property change:

One property:

```
@NotifyChange("oneProperty")
```

Multiple properties:

```
@NotifyChange({"property01","property02"})
```

All properties in a ViewModel:

```
@NotifyChange("*")
```

**Define notification & commands in a ViewModel**

```java
public class ProfileViewModel implements Serializable{
    ...

    @Command //@Command annotates a command method
    @NotifyChange("currentUser") //@NotifyChange annotates data changed notification after calling this method
    public void save(){
        currentUser = userInfoService.updateUser(currentUser);
        Clients.showNotification("Your profile is updated");
    }

    @Command
    @NotifyChange("currentUser")
    public void reload(){
        UserCredential cre = authService.getUserCredential();
        currentUser = userInfoService.findUser(cre.getAccount());
    }
}
```

- Line 5, 12: Notify which property change with `@NotifyChange` and zK will reload those attributes that are bound to `currentUser`.

# Handle User Interactions by Command Binding

After we finish binding attributes to the ViewModel's data, we still need to handle user actions, button clicking. Under the MVVM approach, we handle events by binding an event attribute (e.g. `onClick`) to a **Command** of a ViewModel. After we bind an event to a Command, each time the event is sent, ZK will invoke the corresponding command method. Hence, we should write our business logic in a command method. After executing the command method, some properties might be changed. We should tell ZK which properties are changed by us, then the binder will reload them to components.

When creating the `ProfileViewModel` in the previous section, we have defined two commands: `save` and `reload`.

Then, we can bind `onClick` event to above commands with command binding `@command('commandName')` as follows:

```xml
...
    <hlayout>
        <button onClick="@command('save')" label="Save"/>
        <button onClick="@command('reload')" label="Reload"/>
    </hlayout>
...
```

Done with this binding, clicking each button will invoke corresponding command methods to save (or reload) the user profile to the ViewModel.

## Keep Unsaved Input Away

Once you create a property binding with `@bind` for an input component, ZK will save user input back to a ViewModel automatically. But sometimes this automation is not what users want. In our example, most people usually expect `currentUser` to change after their confirmation, e.g. clicking a button.

There is a line of text "You are editing an Anonymous's profile" at the bottom of the form. If you change the full name to "Anonymous Somebody" and move to next field, the line of text is changed even you don't press the "Save" button. This could be a problem because it would mislead users, making them think they have changed their profile, so we want to remove such

behavior.



**Unsaved Input Changes Data**

We are going to improve this part with **form binding** feature in this section.

Form binding automatically creates a middle object as a buffer. 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 the user confirms.

Steps to use a form binding:

1. Give an id to middle object in `'''form'''` attribute with `@id` .

   Then you can reference the middle object in ZK bind expression with its id, e.g. `@id('fx')` .

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 the middle object's properties to ViewModel before a command execution.

4. Bind component's attribute to the middle object's properties like you do in property binding.

   You should use middle object's id specified in `@id` to reference its property, e.g. `@load(fx.account)` .

**extracted from chapter3/profile-mvvm.zul**

```
...
    <grid width="500px"
    form="@id('fx')@load(vm.currentUser)@save(vm.currentUser, before='save')">
        ...
        <rows>
```

```xml
        <row>
            <cell sclass="row-title">Account :</cell>
            <cell><label value="@load(fx.account)"/></cell>
        </row>
        <row>
            <cell sclass="row-title">Full Name :</cell>
            <cell>
                <textbox value="@bind(fx.fullName)" width="200px"
                constraint="no empty: Plean enter your full name"/>
            </cell>
        </row>
        <row>
            <cell sclass="row-title">Email :</cell>
            <cell>
                <textbox value="@bind(fx.email)" width="200px"
            constraint="/.+@.+\.[a-z]+/: Please enter an e-mail address"/>
            </cell>
        </row>
        <row>
            <cell sclass="row-title">Birthday :</cell>
            <cell>
                <datebox value="@bind(fx.birthday)" width="200px"
                constraint="no future" />
            </cell>
        </row>
        <row>
            <cell sclass="row-title">Country :</cell>
            <cell>
                <listbox model="@load(vm.countryList)"
                    mold="select" width="200px"
                    selectedItem="@bind(fx.country)">
                    <template name="model">
                        <listitem label="@load(each)"/>
                    </template>
                </listbox>
            </cell>
        </row>
        <row>
            <cell sclass="row-title">Bio :</cell>
            <cell><textbox value="@bind(fx.bio)" multiline="true"
                hflex="1" height="200px" />
            </cell>
        </row>
    </rows>
</grid>
<div>You are editing
    <label value="@load(vm.currentUser.fullName)"/>'s profile.
</div>
...
```

- Line 2, 3: Define a form binding at `form` attribute and give the middle object's id `fx` . Specify `@load(vm.currentUser)` makes the binder load `currentUser` 's properties to the middle object and `@save(vm.currentUser, before='save')` makes the binder save middle object's data back to `vm.currentUser` before executing the command `save` .
- Line 8, 13, 20, 27, 36, 45: We should bind attributes to middle object's properties to avoid altering ViewModel's properties.
- LIne 51, 52, 53: The label bound to `vm.currentUser.fullName` is not affected when `fx` is changed.

After applying form binding, any user's input will not actually change `currentUser` 's value and they are stored in the middle object until you click the "Save" button, ZK puts the middle object's data back to the ViewModel's properties ( `currentUser` ).

**Unsaved Input Doesn't Change Data**

</div> After completing above steps, visit http://localhost:8080/zkessentials/chapter3/index-mvvm.zul to see the result.

current:

Improve this Doc

# Target Application

In this chapter, we are going to build a "todo list" with 4 basic operations, create, read, update, and delete (CRUD). The application's user interface looks like the images below:
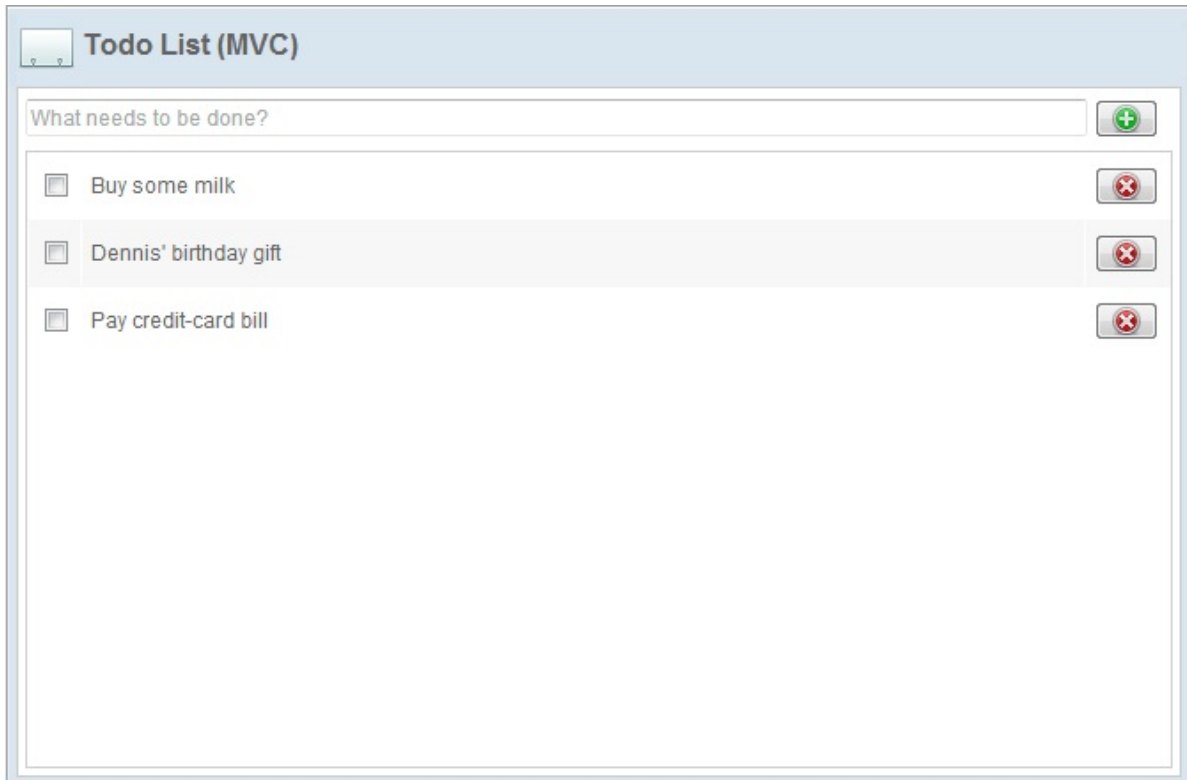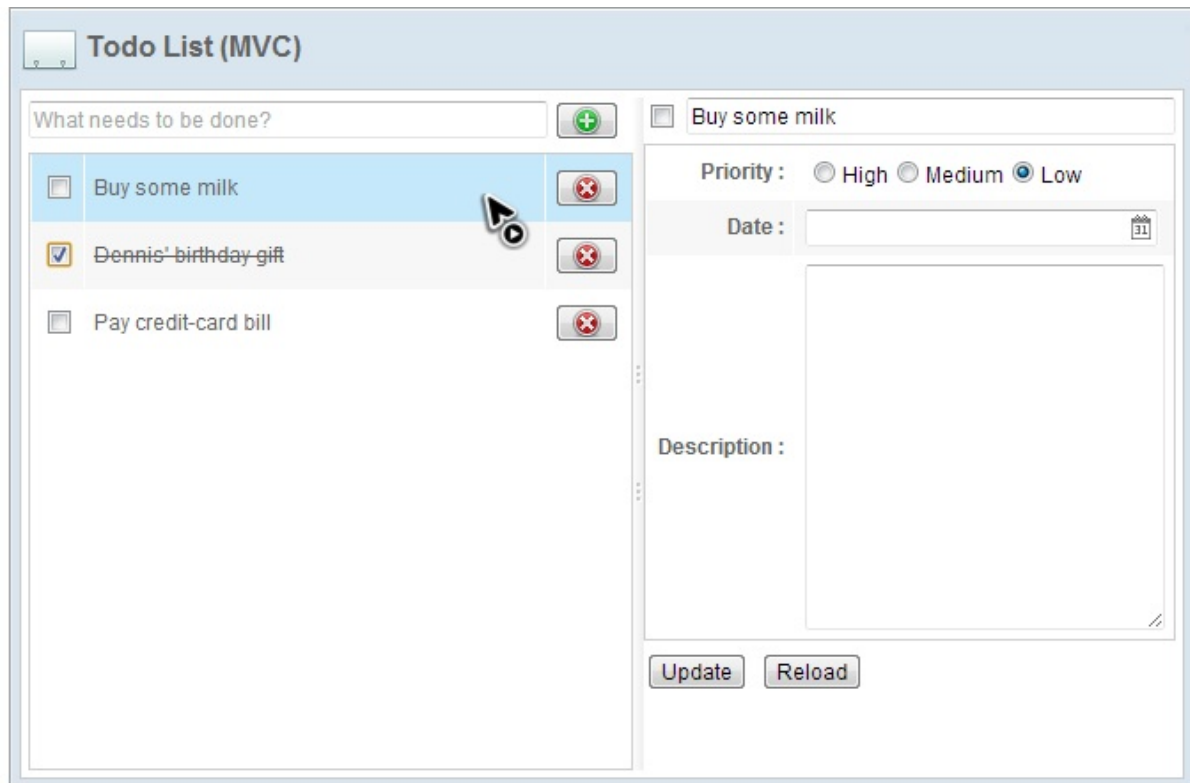
**Select a Todo Item**

</div> It is a personal todo list management system and it has following features:

1. List all todo items

2. Create a todo item.

   Type item name in upper-left textbox and click  or press "Enter" key to create a new todo item.

3. Finish a todo item.

   Click the checkbox in front of a todo item to mark it as finished and the item name will be decorated with line-through.

4. Modify a todo item.

   Click an existing item and the detail editor appears. Then you can edit the item's details.

5. Delete a todo item.

   Click  to delete an existing todo item.

In this chapter, we will show how to implement the target application using both the MVC and MVVM approaches. If you are not familiar with these two approaches, we suggest you to read Get ZK Up and Running with MVC and Get ZK Up and Running with MVVM. These two approaches are mutually interchangeable. You can choose one of them depending on your situation.

# MVC Approach

If you have read previous chapters, constructing the user interface for the example application should not be a big problem. Let's look at the layout first and ignore the details.

**Layout in chapter4/todolist-mvc.zul**

```
<?link rel="stylesheet" type="text/css" href="/style.css"?>
<window apply="org.zkoss.essentials.chapter4.mvc.TodoListController"
    border="normal" hflex="1" vflex="1" contentStyle="overflow:auto">
    <caption src="/imgs/todo.png" sclass="fn-caption" label="Todo List (MVC)"/>
    <borderlayout>
        <center autoscroll="true" border="none">
            <vlayout hflex="1" vflex="1">
                <!-- todo creation function-->
                <!-- todo list -->
            </vlayout>
        </center>
        <east id="selectedTodoBlock" visible="false"
        width="300px" border="none" collapsible="false"
        splittable="true" minsize="300" autoscroll="true">
            <vlayout >
                <!-- detail editor -->
            </vlayout>
        </east>
    </borderlayout>
</window>
```

- Line 5: We construct the user interface with a *Border Layout* to separate user interface into 2 areas.
- Line 6: The center area contains a todo creation function and a todo list.
- Line 12, 13: The east area is a todo item detail editor which is invisible if no item selected.

# Read

As we talked in previous chapters, we can use *Template* to define how to display a data model list with implicit variable `each` .

**Display a ToDo List**

```
...
    <listbox id="todoListbox" vflex="1">
        <listhead>
            <listheader width="30px" />
            <listheader/>
            <listheader hflex="min"/>
        </listhead>
        <template name="model">
            <listitem sclass="${each.complete?'complete-todo':''}"
                value="${each}">
                <listcell>
                    <checkbox forward="onCheck=todoListbox.onTodoCheck"
                        checked="${each.complete}"/>
                </listcell>
                <listcell>
                    <label value="${each.subject}"/>
                </listcell>
                <listcell>
```

```
                    <button forward="onClick=todoListbox.onTodoDelete"
                           image="/imgs/cross.png" width="36px"/>
                </listcell>
            </listitem>
        </template>
    </listbox>
...
```

- Line 8: The default value for the required attribute `name` is "model".
- Line 10: The `${each}` is an implicit variable that you can use without declaration inside *Template*, and it represents each object of the data model list. We can implement simple presentation logic with EL expressions. Here we apply different styles according to a flag `each.complete`. We also set a whole object in `value` attribute, and later we can get the object in the controller.
- Line 13: The `each.complete` is a boolean variable so that we can assign it to `checked`. By doing this, the *Checkbox* will be checked if the todo item's `complete` variable is true.
- Line 12, 19: The `forward` attribute is used to forward events to another component and we will talk about it in later sections.

In the controller, we should provide a data model for the *Listbox*.

```
public class TodoListController extends SelectorComposer<Component>{


    //wire components
    ...
    @Wire
    Listbox todoListbox;


    ...


    //services
    TodoListService todoListService = new TodoListServiceChapter4Impl();

    //data for the view
    ListModelList<Todo> todoListModel;
    ListModelList<Priority> priorityListModel;
    Todo selectedTodo;


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

        //get data from service and wrap it to list-model for the view
        List<Todo> todoList = todoListService.getTodoList();
        todoListModel = new ListModelList<Todo>(todoList);
        todoListbox.setModel(todoListModel);

        ...
    }
...
}
```

- Line 25 ~ 27: We initialize the data model in `doAfterCompose()`. Get data from the service class `todoListService` and create a `ListModelList` object. Then set it as the data model of `todoListbox`.

There is a priority radiogroup in todo item detail editor appeared on the right hand side when you select an item.



**Todo Item's Priority Radiogroup**

</div> In our application, its priority labels come from an enumerating `Priority` instead of a static text. We can still use *Template* to define how to create each *Radio* under a *Radiogroup*. The zul looks like as follows:

```
...
        <row>
            <cell sclass="row-title">Priority :</cell>
            <cell>
                <radiogroup id="selectedTodoPriority">
                    <template name="model">
                        <radio label="${each.label}"/>
                    </template>
                </radiogroup>
            </cell>
        </row>
...
```

- Line 6 ~8: Define how to create each *Radio* with *Template* and assign `each.label` to `label` attribute.

We also need to provide a data model for the *Radiogroup* in the controller:

```
public class TodoListController extends SelectorComposer<Component>{

    //wire components
    ...
    @Wire
    Listbox todoListbox;

    ...
    @Wire
    Radiogroup selectedTodoPriority;
    ...

    //services
    TodoListService todoListService = new TodoListServiceChapter4Impl();

    //data for the view
    ListModelList<Todo> todoListModel;
    ListModelList<Priority> priorityListModel;
    Todo selectedTodo;

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

        //get data from service and wrap it to list-model for the view
        List<Todo> todoList = todoListService.getTodoList();
        todoListModel = new ListModelList<Todo>(todoList);
        todoListbox.setModel(todoListModel);

        priorityListModel = new ListModelList<Priority>(Priority.values());
        selectedTodoPriority.setModel(priorityListModel);
    }
...
}
```

- Line 31, 32: Create a `ListModelList` with `Priority` and set it as a model of `selectedTodoPriority`.

## Create

After typing the todo item name, we can save the item by either clicking the button with the plus icon (  ) or pressing "Enter" key. Therefore, we have to listen to 2 events: `onClick` and `onOK` . For handling other key pressing events, please refer to ZK_Developer's_Reference/UI_Patterns/Keystroke_Handling.

```java
public class TodoListController extends SelectorComposer<Component>{

    //wire components
    @Wire
    Textbox todoSubject;

    //services
    TodoListService todoListService = new TodoListServiceChapter4Impl();

    //data for the view
    ListModelList<Todo> todoListModel;
    ListModelList<Priority> priorityListModel;
    Todo selectedTodo;

    ...

    //when user clicks on the button or enters on the textbox
    @Listen("onClick = #addTodo; onOK = #todoSubject")
    public void doTodoAdd(){
        //get user input from view
        String subject = todoSubject.getValue();
        if(Strings.isBlank(subject)){
            Clients.showNotification("Nothing to do ?",todoSubject);
        }else{
            //save data
            selectedTodo = todoListService.saveTodo(new Todo(subject));
            //update the model of listbox
            todoListModel.add(selectedTodo);
            //set the new selection
            todoListModel.addToSelection(selectedTodo);

            //refresh detail view
            refreshDetailView();

            //reset value for fast typing.
            todoSubject.setValue("");
        }
    }
...
}
```

- Line 18: Listen the button's `onClick` event and "Enter" key pressing event: `onOK` .
- Line 19: This method adds a todo item, update the data model of *Listbox*, change the selection to a newly created one, then reset the input field of the *Textbox*.
- Line 21: Get user input in the *Textbox* `todoSubject` by `getValue()` .
- Line 23: Show a notification at the right hand side of the *Textbox* `todoSubject` .
- Line 28: When you change (add or remove) items in a `ListModelList` object, it will automatically render in the *Listbox*'s.
- Line 30: Call `addToSelection()` to assign a component's selection and it will automatically reflect to the corresponding widget's selection.

## Update

To update a todo item, you should select an item first then detail editor will appear. The following codes demonstrate how to listen a "onSelect" event and display the item's detail.

```java
public class TodoListController extends SelectorComposer<Component>{
```

```
    //wire components
    @Wire
    Textbox todoSubject;
    @Wire
    Button addTodo;
    @Wire
    Listbox todoListbox;

    @Wire
    Component selectedTodoBlock;
    @Wire
    Checkbox selectedTodoCheck;
    @Wire
    Textbox selectedTodoSubject;
    @Wire
    Radiogroup selectedTodoPriority;
    @Wire
    Datebox selectedTodoDate;
    @Wire
    Textbox selectedTodoDescription;
    @Wire
    Button updateSelectedTodo;


    //when user selects a todo of the listbox
    @Listen("onSelect = #todoListbox")
    public void doTodoSelect() {
        if(todoListModel.isSelectionEmpty()){
            //just in case for the no selection
            selectedTodo = null;
        }else{
            selectedTodo = todoListModel.getSelection().iterator().next();
        }
        refreshDetailView();
    }

    private void refreshDetailView() {
        //refresh the detail view of selected todo
        if(selectedTodo==null){
            //clean
            selectedTodoBlock.setVisible(false);
            selectedTodoCheck.setChecked(false);
            selectedTodoSubject.setValue(null);
            selectedTodoDate.setValue(null);
            selectedTodoDescription.setValue(null);
            updateSelectedTodo.setDisabled(true);

            priorityListModel.clearSelection();
        }else{
            selectedTodoBlock.setVisible(true);
            selectedTodoCheck.setChecked(selectedTodo.isComplete());
            selectedTodoSubject.setValue(selectedTodo.getSubject());
            selectedTodoDate.setValue(selectedTodo.getDate());
            selectedTodoDescription.setValue(selectedTodo.getDescription());
            updateSelectedTodo.setDisabled(false);

            priorityListModel.addToSelection(selectedTodo.getPriority());
        }
    }
...
}
```

- Line 29: Use `@Listen` to listen `onSelect` event of the *Listbox* whose id is `todoListbox` .
- Line 30: This method checks `todoListModel` 's selection and refreshes the detail editor.
- Line 35: Get user selection from data model by `getSelection()` which returns a `Set` .

- Line 40: If an item is selected, it makes detail editor visible and pushes data into those input components of the editor by calling setter methods. If no item is selected, it makes detail editor invisible and clear all input components' value.
- Line 53: Make the detail editor visible when `selectedTodo` is not null.
- Line 60: Use `addToSelection()` to assign a component's selection and it will automatically reflect to the corresponding widget's selection.

After modifying the item's detail, you can click the "Update" button to save the modification or "Reload" to revert back original data. The following codes demonstrate how to implement these functions:

**Handle clicking "update" and "reload" button**

```java
//when user clicks the update button
@Listen("onClick = #updateSelectedTodo")
public void doUpdateClick(){
    if(Strings.isBlank(selectedTodoSubject.getValue())){
        Clients.showNotification("Nothing to do ?",selectedTodoSubject);
        return;
    }

    selectedTodo.setComplete(selectedTodoCheck.isChecked());
    selectedTodo.setSubject(selectedTodoSubject.getValue());
    selectedTodo.setDate(selectedTodoDate.getValue());
    selectedTodo.setDescription(selectedTodoDescription.getValue());
    selectedTodo.setPriority(priorityListModel.getSelection().iterator().next());

    //save data and get updated Todo object
    selectedTodo = todoListService.updateTodo(selectedTodo);

    //replace original Todo object in listmodel with updated one
    todoListModel.set(todoListModel.indexOf(selectedTodo), selectedTodo);

    //show message for user
    Clients.showNotification("Todo saved");
}

//when user clicks the update button
@Listen("onClick = #reloadSelectedTodo")
public void doReloadClick(){
    refreshDetailView();
}
```

- Line 4: Validate user input and show a notification.
- Line 9 ~ 13: Update selected `Todo` by getting user input from components.
- Line 16, 19: We save the selected `Todo` object and get an updated one. Then, we replace the old one in the list model with the updated one.

## Complete a Todo

Click a *Checkbox* in front of a todo item means to finish it. To implement this feature, the first problem is: how do we know which *Checkbox* is checked as there are many of them. We cannot listen to a *Checkbox* event as they are created in template using `@Listen("onCheck = #todoListbox checkbox")` ,thus are created dynamically. Therefore, we introduce the "Event Forwarding" feature to demonstrate ZK's flexibility. This feature can forward an event from a component to another component, so we can forward an `onCheck` event from each *Checkbox* to the *Listbox* that encloses it, then we can just listen to the *Listbox*'s events instead of all events of *Checkbox*.

**extracted from chapter4/todolist-mvc.zul**

```xml
...
    <listbox id="todoListbox" vflex="1">
...
        <template name="model">
```

```
            <listitem sclass="${each.complete?'complete-todo':''}" value="${each}">
                <listcell>
                    <checkbox forward="onCheck=todoListbox.onTodoCheck" checked="${each.complete}"/>
                </listcell>
                <listcell>
                    <label value="${each.subject}"/>
                </listcell>
                <listcell>
                    <button forward="onClick=todoListbox.onTodoDelete" image="/imgs/cross.png" width="36px"/>
                </listcell>
            </listitem>
        </template>
```

- Line 7: Forward the Checkbox's `onCheck` to an event `onTodoCheck` of a *Listbox* whose id is `todoListbox` . The `onTodoCheck` is a customized forward event name, and you can use whatever name you want. Then we can use `@Listen` to listen this special event name.

Next, we listen to the customized event `onTodoCheck` and mark the todo as finished.

```
public class TodoListController extends SelectorComposer<Component>{
...

    //when user checks on the checkbox of each todo on the list
    @Listen("onTodoCheck = #todoListbox")
    public void doTodoCheck(ForwardEvent evt){
        //get data from event
        Checkbox cbox = (Checkbox)evt.getOrigin().getTarget();
        Listitem litem = (Listitem)cbox.getParent().getParent();

        boolean checked = cbox.isChecked();
        Todo todo = (Todo)litem.getValue();
        todo.setComplete(checked);

        //save data
        todo = todoListService.updateTodo(todo);
        if(todo.equals(selectedTodo)){
            selectedTodo = todo;
            //refresh detail view
            refreshDetailView();
        }
        //update listitem style
        ((Listitem)cbox.getParent().getParent()).setSclass(checked?"complete-todo":"");
    }
...
}
```

- Line 5: Listen to the customized event name `onTodoCheck` of a *Listbox* `todoListbox` for we already forward `onCheck` to the *Listbox* in the zul.
- Line 6: An event listener method can have a argument, but argument's type depends on which event you listen. As the customized event is forwarded from another component, the argument should be org.zkoss.zk.ui.event.ForwardEvent. This method set the `Todo` object of the selected item as complete and decorate *Listitem* with line-through by changing its `sclass` .
- Line 8: You should call `getOrigin()` to get the original event that is forwarded. Every event object has a method `getTarget()` that allows you get the target component that receives the event.
- Line 9: Navigate the component tree by `getParent()` .
- Line 12: Here we get `Todo` object of the selected todo item from `value` attribute that we assigned in the zul by `<listitem ... value="${each}"/>`

# Delete

Implement deletion feature is similar to completing a todo item. We also forward each delete button's (  ) `onClick` event to the *Listbox* that encloses those buttons.

**Forward delete button's `onClick`**

```
    <listbox id="todoListbox" vflex="1">
 ...
        <template name="model">
            <listitem sclass="${each.complete?'complete-todo':''}"
                        value="${each}">
                <listcell>
                    <checkbox forward="onCheck=todoListbox.onTodoCheck"
                                checked="${each.complete}"/>
                </listcell>
                <listcell>
                    <label value="${each.subject}"/>
                </listcell>
                <listcell>
                    <button forward="onClick=todoListbox.onTodoDelete"
                        image="/imgs/cross.png" width="36px"/>
                </listcell>
            </listitem>
        </template>
 ...
```

- Line 14, 15: Forward delete button's `onClick` to the *Listbox*'s as a custom forward event named `onTodoDelete` .

Then we can listen to the forwarded event and perform deletion.

```
    //when user clicks the delete button of each todo on the list
    @Listen("onTodoDelete = #todoListbox")
    public void doTodoDelete(ForwardEvent evt){
        Button btn = (Button)evt.getOrigin().getTarget();
        Listitem litem = (Listitem)btn.getParent().getParent();

        Todo todo = (Todo)litem.getValue();

        //delete data
        todoListService.deleteTodo(todo);

        //update the model of listbox
        todoListModel.remove(todo);

        if(todo.equals(selectedTodo)){
            //refresh selected todo view
            selectedTodo = null;
            refreshDetailView();
        }
    }
```

- Line 2: Listen the customized event name `onTodoDelete` of a *Listbox* that we forward from delete button.
- Line 7: Since we have set each `Todo` object to each `Listitem` 's `value` in the zul, we can get it by `getValue()`

After completing the above steps, vist http://localhost:8080/zkessentials/chapter4/todolist-mvc.zul to see the result.

# MVVM Approach

Building a user interface for the example application using the MVVM approach is very similar to building it using the MVC approach, however, you don't need to give an `id` to components since we don't need to identify components for wiring. For defining the ViewModel's properties, we should analyse what data required to display on the user interface or to be kept as a View's state. There are 4 types of data, todo item's subject for creating a new todo item, todo item list for displaying all todos, selected todo item for keeping user selection, and todo's priority list for *Radiogroup* in detail editor.

```
public class TodoListViewModel implements Serializable{


    //data for the view
    String subject;
    ListModelList<Todo> todoListModel;
    Todo selectedTodo;

    public List<Priority> getPriorityList(){
        return Arrays.asList(Priority.values());
    }

    //omit property accessor methods and others
...
}
```

- A property is retrieved by a getter, so ViewModel doesn't have to declare a variable for a property.

# Read

As we discussed in previous chapter, displaying a collection of data requires to prepare a data model in the ViewModel.

```
public class TodoListViewModel implements Serializable{


    //services
    TodoListService todoListService = new TodoListServiceChapter4Impl();

    //data for the view
    String subject;
    ListModelList<Todo> todoListModel;
    Todo selectedTodo;

    @Init // @Init annotates a initial method
    public void init(){
        //get data from service and wrap it to model for the view
        List<Todo> todoList = todoListService.getTodoList();
        //you can use List directly, however use ListModelList provide efficient control in MVVM
        todoListModel = new ListModelList<Todo>(todoList);
    }

...
}
```

- Line 17: Initialize `ListModelList` with `todoList` retrieved with a service class.

Then we can bind Listbox's `model` to prepared data model of the ViewModel with data binding expression.

```
    <listbox model="@bind(vm.todoListModel)"
        selectedItem="@bind(vm.selectedTodo)" vflex="1" >
        <listhead>
            <listheader width="30px" />
            <listheader/>
            <listheader hflex="min"/>
        </listhead>
        <template name="model">
            <listitem sclass="@bind(each.complete?'complete-todo':'')">
                <listcell>
                    <checkbox checked="@bind(each.complete)"
                        onCheck="@command('completeTodo',todo=each)"/>
                </listcell>
                <listcell>
                    <label value="@bind(each.subject)"/>
                </listcell>
                <listcell>
                    <button onClick="@command('deleteTodo',todo=each)"
                        image="/imgs/cross.png" width="36px"/>
                </listcell>
            </listitem>
        </template>
    </listbox>
```

- Line 1, 2: Set *Listbox*'s data model by binding `model` attribute to a property of type `ListModelList` . Binding `selecteditem` to `vm.selectedTodo` to keep user selection in the ViewModel.
- Line 9: You can fill any valid EL expression In a data binding annotation, so that you can implement simple presentation logic with EL. Here we set `sclass` according to a `Todo` object's `complete` property.
- Line 11, 12, 15: Use implicit variable `each` to access each `Todo` object in the data model.

# Create

We can create a new todo item by either clicking the button with plus icon (⬜) or pressing the "Enter" key, therefore we can bind these two events to the same command method that adds a todo item.

**Command method** `addTodo`

```
    @Command //@Command annotates a command method
    @NotifyChange({"selectedTodo","subject"}) //@NotifyChange annotates data changed notification after calling
this method
    public void addTodo(){
        if(Strings.isBlank(subject)){
            Clients.showNotification("Subject is blank, nothing to do ?");
        }else{
            //save data
            selectedTodo = todoListService.saveTodo(new Todo(subject));
            //update the model, by using ListModelList, you don't need to notify todoListModel change
            //it is efficient that only update one item of the listbox
            todoListModel.add(selectedTodo);
            todoListModel.addToSelection(selectedTodo);

            //reset value for fast typing.
            subject = null;
        }
    }
```

- Line 2: You can notify multiple properties change by filling an array of String. Here we specify `{"selectedTodo","subject"}` , since we change them in the method.

We can see that the benefits of abstraction provided by command binding allows developers to bind different events to the same command without affecting the ViewModel.

**Binding to `addTodo`**

```
<hbox align="center" hflex="1" sclass="todo-box">
    <textbox value="@bind(vm.subject)"
        onOK="@command('addTodo')"
            hflex="1" placeholder="What needs to be done?"/>
    <button onClick="@command('addTodo')"
        image="/imgs/plus.png" width="36px"/>
</hbox>
```

- Line 2~6: The `onOK` and `onClick` can invoke the same command method.

# Update

How do we achieve the feature that selecting a todo item then detail editor becomes visible under MVVM? Simple, just determine editor's visibility upon selected todo item is null or not.

```
<east visible="@bind(not empty vm.selectedTodo)" width="300px"
border="none" collapsible="false" splittable="true"
minsize="300" autoscroll="true">
<!-- todo item detail editor-->
</east>
```

- Line 1: ZK bind monitors all binding properties. If one property changes, ZK bind re-evaluates those binding expressions that bind to the changed property.

In order to make selected todo item's properties display in the detail editor, we just bind input components in the detail editor to the corresponding `selectedTodo` 's properties.

**Binding input components to selected item's properties**

```
<vlayout
form="@id('fx') @load(vm.selectedTodo)
    @save(vm.selectedTodo, before='updateTodo')">
    <hbox align="center"  hflex="1">
        <checkbox checked="@bind(fx.complete)"/>
        <textbox value="@bind(fx.subject)" hflex="1" />
    </hbox>
    <grid hflex="1">
        <columns>
            <column align="right" hflex="min"/>
            <column/>
        </columns>
        <rows>
            <row>
                <cell sclass="row-title">Priority :</cell>
                <cell>
                    <radiogroup model="@bind(vm.priorityList)"
                        selectedItem="@bind(fx.priority)">
                        <template name="model">
                            <radio label="@bind(each.label)"/>
                        </template>
                    </radiogroup>
                </cell>
            </row>
            <row>
```

```
                    <cell sclass="row-title">Date :</cell>
                    <cell><datebox value="@bind(fx.date)" width="200px"/>
                    </cell>
                </row>
                <row>
                    <cell sclass="row-title">Description :</cell>
                    <cell>
                        <textbox value="@bind(fx.description)" multiline="true"
                            hflex="1" height="200px" />
                    </cell>
                </row>
            </rows>
        </grid>
        <hlayout>
            <button onClick="@command('updateTodo')" label="Update"/>
            <button onClick="@command('reloadTodo')" label="Reload"/>
        </hlayout>
    </vlayout>
```

- Line 2,3: Here we create a form binding at `form` attribute and give the middle object's id fx. Specify `@load(vm.selectedTodo)` makes the binder load selected todo's properties to the middle object and `@save(vm.selectedTodo, before='updateTodo')` makes the binder save middle object's data back to `vm.selectedTodo` before executing the command `updateTodo`, bound in line 36.
- Line 5,6,17,27,32,33: Binding each input field to each property of the middle object through `fx`.
- Line 17: Binding model of *Radiogroup* to `vm.priorityList` to display 3 priority levels.

After modifying item's detail, you can click the "Update" button to save the modification or "Reload" to revert back original data. These two functions are implemented in command methods:

```
    @Command
    @NotifyChange("selectedTodo")
    public void updateTodo(){
        //update data
        selectedTodo = todoListService.updateTodo(selectedTodo);

        //update the model, by using ListModelList, you don't need to notify todoListModel change
        //by reseting an item , it make listbox only refresh one item
        todoListModel.set(todoListModel.indexOf(selectedTodo), selectedTodo);
    }

    //when user clicks the update button
    @Command @NotifyChange("selectedTodo")
    public void reloadTodo(){
        //do nothing, the selectedTodo will reload by notify change
    }
```

- Line 9: `ListModelList` can update its change to the client automatically, you don't have to notify change of `todoListModel`.

then we can invoke them by command binding:

```
        <hlayout>
            <button onClick="@command('updateTodo')" label="Update"/>
            <button onClick="@command('reloadTodo')" label="Reload"/>
        </hlayout>
```

## Input Validation

Under MVVM approach, ZK provides a **validator** to help developers perform user input validation. Validator is a reusable element that performs validation. If 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. Here we implement a validator to avoid empty value of todo's subject.

**Define a validator in the ViewModel**

```java
//the validator is the class to validate data before set ui data back to todo
public Validator getTodoValidator(){
    return new AbstractValidator() {

        public void validate(ValidationContext ctx) {
            //get the form that will be applied to todo
            Form fx = (Form)ctx.getProperty().getValue();
            //get filed subject of the form
            String subject = (String)fx.getField("subject");

            if(Strings.isBlank(subject)){
                Clients.showNotification("Subject is blank, nothing to do ?");
                //mark the validation is invalid, so the data will not update to bean
                //and the further command will be skipped.
                ctx.setInvalid();
            }
        }
    };
}
```

- Line 2: Returning a validator object by a getter method makes it as a ViewModel's property, so we can bind it to an attribute.
- Line 3: In most case, we can create a validator by extending org.zkoss.bind.validator.AbstractValidator and override `validate()` instead of creating from scratch.
- Line 7: Get user input from `ValidationContext`. In our example, because we will apply this validator to form binding, we expect `ctx.getProperty().getValue()` returns a `Form` object.
- Line 9: You can get every field that middle object contains with a property name.
- Line 15: Call `set Invalid()` to fail the validation then further command execution will be skipped.

Then apply this validator with data binding expression.

```xml
<vlayout
form="@id('fx') @load(vm.selectedTodo)
@save(vm.selectedTodo, before='updateTodo')
@validator(vm.todoValidator)">
```

Hence, if `vm.todoValidator` fails validation, ZK won't execute `updateTodo` command. Then binder won't save value to `selectedTodo`.

## Complete a Todo

We want clicking a *Checkbox* in front of each todo item to complete a todo item. First, we implement business logic to complete a todo item.

```java
@Command
//@NotifyChange("selectedTodo") //use postNotifyChange() to notify dynamically
public void completeTodo(@BindingParam("todo") Todo todo){
    //save data
    todo = todoListService.updateTodo(todo);
    if(todo.equals(selectedTodo)){
        selectedTodo = todo;
        //for the case that notification is decided dynamically
```

```
            //you can use BindUtils.postNotifyChange to notify a value changed
            BindUtils.postNotifyChange(null, null, this, "selectedTodo");
        }
    }
```

- LIne 3: 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.
- Line 10: We demonstrate programmatic way to notify change by `BindUtils.postNotifyChange()` . We leave the first and second parameters to null as default. The third parameters is the target bean that is changed and the fourth parameter is the changed property name.

Then we bind `onCheck` to the command `completeTodo` .

```
<template name="model">
    <listitem sclass="@bind(each.complete?'complete-todo':'')">
        <listcell>
            <checkbox checked="@bind(each.complete)"
                onCheck="@command('completeTodo',todo=each)"/>
        </listcell>
        ...
    </listitem>
</template>
```

- Line 4,5: Command binding allows you to pass an arguments in key-value pairs. We pass `each` object with key `todo` .

## Delete

Implementing a delete function is very similar to "completing a todo", we perform business logic and notify change.

```
@Command
//@NotifyChange("selectedTodo") //use postNotifyChange() to notify dynamically
public void deleteTodo(@BindingParam("todo") Todo todo){
    //save data
    todoListService.deleteTodo(todo);

    //update the model, by using ListModelList, you don't need to notify todoListModel change
    todoListModel.remove(todo);

    if(todo.equals(selectedTodo)){
        //refresh selected todo view
        selectedTodo = null;
        //for the case that notification is decided dynamically
        BindUtils.postNotifyChange(null, null, this, "selectedTodo");
    }
}
```

- Line 8: When you change (add or remove) items in a `ListModelList` object, it will automatically reflect to *Listbox*'s rendering.

Next, bind `onClick` to the command `deleteTodo` then we are done editing this function.

```
<template name="model">
    <listitem sclass="@bind(each.complete?'complete-todo':'')">
        <listcell>
            <checkbox checked="@bind(each.complete)"
                onCheck="@command('completeTodo',todo=each)"/>
        </listcell>
```

```
            <listcell>
                <label value="@bind(each.subject)"/>
            </listcell>
            <listcell>
                <button onClick="@command('deleteTodo',todo=each)"
                    image="/imgs/cross.png" width="36px"/>
            </listcell>
        </listitem>
    </template>
```

- Line 11,12: In order to know which `Todo` object we should delete, we pass the deleting `Todo` by `todo=each` . The `todo` is key and `each` is value.

The key development activities under MVVM approach are designing a ViewModel, implementing command methods, and binding attributes to a ViewModel. You can see that the relationship between ZUL and ViewModel is relatively decoupling and only established by data binding expressions.

After completing above steps, please visit http://localhost:8080/zkessentials/chapter4/todolist-mvvm.zul to see the result.

# Shadow Components

ZK 8 introduces a new set of components: **shadow components**. Shadow components, which are similar to shadow DOM, neither create a corresponding component on the server side nor a widget on the client side. They simply inject their child components into the current page. If we use them with data binding, a page can dynamically change upon a ViewModel's data. Since shadow components have flow control and iteration function, if you identify a repeating view pattern appearing in your application, you can implement that view pattern with shadow components to modulize your view. Hence, their most powerful feature is **making the view reusable**.

Below is the list of shadow components:

- `<apply>` : allows you to choose which template to be applied. It will look up the template inside-out recursively.
- `<forEach>` : allows you to iterate over a collection of objects. Specify the collection by using the `items` attribute, and you can access the current item through a variable specified at the `var` attribute.
- `<if>` : allows the conditional execution of its body according to the value of the test attribute.
- `<choose>` / `<when>` / `<otherwise>` : they are used for logic and flow control like Java's `switch` / `case` / `default` statement.

We can use shadow components anywhere on a zul. For example, we can create a component based on a condition:

```
<if test="@load(vm.readonly)">
    <button label="Edit"/>
</if>
```

Or create a collection of components:

```
<forEach begin="0" end="3">
    <button label="${'Button'+=each}"/>
</forEach>
```

# Setup

Before using shadow elements, make sure you include the required jar - `zuti.jar` . With maven, you should add the dependency below:

```
<dependency>
    <groupId>org.zkoss.zk</groupId>
    <artifactId>zuti</artifactId>
    <version>${zk.version}</version>
</dependency>
```

# Navigation Menu Example

In this chapter, we will demonstrate the power of shadow components with a navigation menu example shown below:



The menu is made by templates containing shadow components, and it can render different menu hierarchies without any change. You can switch 2 sets of menu hierarchy via the radio group on the right hand side. This example demonstrates how to reuse a common view pattern.

## Menu Data Structure

We store a menu hierarchical structure in a linked list, and each node in the list may have sub-menus.

```
package org.zkoss.essentials.chapter5.template;

import java.util.*;
import org.zkoss.bind.annotation.*;
import org.zkoss.zk.ui.event.SelectEvent;
import org.zkoss.zkmax.zul.Navitem;

public class MenuViewModel {

    private List<MenuNode> menuHierarchy = null;
    ...
}
```

```
package org.zkoss.essentials.chapter5.template;

import java.util.List;

public class MenuNode {

    private String label;
    private String iconSclass;
    private List<MenuNode> subMenus;
    ...
}
```

current:

Improve this Doc

# Iterate a Collection

To start with the basics, we can just render the list of menu nodes with `<forEach>` (notice the captalized 'E') and `<navitem>` . The `<forEach>` can iterate over a collection of objects. Specify the collection by the `items` attribute and access the current item through a variable named `each` . Or you can define the current item variable by the `var` attribute e.g. if you write `<forEach items="@load(menuItems)" var="menu">` then you should use `menu` in a data binding expression like `<navitem label="@load(menu.label)">` .

**chapter5/index.zul**

```
<navbar id="navbar" orient="horizontal" collapsed="false">
    <forEach items="@load(menuItems)">
        <navitem label="@load(each.label)"
            iconSclass="@load(each.iconSclass)">
        </navitem>
    </forEach>
</navbar>
```

However, doing it this way will only render each menu node as a `<navitem>` . Thus, we need to further render these menu nodes with a sub-menu in order to create a `<nav>` .

# Flow Control

Next, we will render a menu node upon its sub-menu existence. If a menu node has a sub-menu, we can render it either by `<nav>` or `<navitem>` . To achieve this, we have to use shadow components for flow control: `<choose>` , `<when>` , and `<otherwise>` . (There is an `<if>` which we don't use in the example.)

- The `<choose>` is similar to Java `switch` statement.
- The `<when>` is similar to `case` statement, and you should specify a data binding expression on `test` attribute to be evaluated.
- The `<otherwise>` works like `default` case in a `switch` statement for specifying a default action.

Now, we can test whether a menu node has a subMenu or not by `test="@load(empty each.subMenus)"` . If it does, create a `<nav>` ; otherwise create a `<navitem>` .

```
<navbar id="navbar" orient="horizontal" collapsed="false">
    <forEach items="@load(menuItems)">
        <choose>
            <when test="@load(empty each.subMenus)">
                <navitem label="@load(each.label)" />
            </when>
            <otherwise>
                <nav label="@load(each.label)" iconSclass="@load(each.iconSclass)"/>
            </otherwise>
        </choose>
    </forEach>
</navbar>
```

But the above method only allows us to create a `<nav>` without its `<navitem>` . Thus, we need to traverse each node in its sub-menu and create corresponding components ( `<nav>` or `<navitem>` ) which is like what we are doing now for each menu-node. This means we need to reuse this `<forEach>` and its child elements.

Improve this Doc

# Reusing Components with a Template

In ZK 8, `<template>` is our recommended form for reusing a view pattern composed by a group of components. Putting components into `<template>` can make them reusable easily by `<apply>` . It usually involves 2 steps:

1. Define a template
2. Apply a template

## Defining a template

With ZK 8, you can put a `<template>` inside any component. Defining a template will not create any component until you apply it. You can define a template like this:

```
<div>
    <template name="layout">
    <!-- UI components or shadow components -->
    </template>
</div>
```

or a path of a zul

```
<div>
    <template name="layout" src="/mytemplate.zul"/>
</div>
```

The usage is the same as what we mentioned in previous chapters. But we can also use the following tags to describe a component creation logic based on certain conditions.

## Applying a Template

When we apply a template, ZK will create the components inside the template upon its logic and insert those components into the position of `<apply>` tag. Therefore, we also call it *Template Injection*.

We usually apply a template with its name like:

```
<apply template="layout"/>
```

Or apply with a path of a zul like:

```
<apply templateURI="/chapter1/banner.zul"/>
```

## Turning Components into Templates

To reuse the `<forEach>` , we turn it into a template named `iterate` first.

```
<navbar id="navbar" orient="horizontal" collapsed="false" onSelect="@command('navigate')" >
    <apply template="iterate" menuItems="@ref(vm.menuHierarchy)"/>
```

```
    </navbar>
<template name="iterate">
    <forEach items="@load(vm.menuHierarchy)">
        <choose>
            <when test="@load(empty each.subMenus)">
                <navitem label="@load(each.label)" />
            </when>
            <otherwise>
                <nav label="@load(each.label)" iconSclass="@load(each.iconSclass)"/>
            </otherwise>
        </choose>
    </forEach>
</template>
```

- Line 2: We pass a parameter by `menuItems="@ref(vm.menuHierarchy)"` . Therefore, we can access the menu list in `<forEach>` by `items="@load(menuItems)"` .

In this simple case (just 2 choices), we can re-write it in a simpler way by creating 2 templates for `menu` and `menuitem` respectively.

```
<template name="menu">
    <nav label="@load(menuItem.label)" iconSclass="@load(menuItem.iconSclass)"/>
</template>
```

```
<template name="menuitem" >
    <navitem label="@load(menuItem.label)" />
</template>
```

Then replace `<choose>` / `<when>` / `<otherwise>` with ternary operator `?` like:

```
<template name="iterate">
    <forEach items="@load(menuItems)">
        <apply template="@load(empty each.subMenus?'menuitem':'menu')" menuItem="@ref(each)"/>
    </forEach>
</template>
```

# Applying a Template Inside a Template

Everything is fine so far except for the fact those sub-menus are not rendered. That's because in template `menu` , we only render the menu node itself to a `<nav>` and don't render its sub-menu. A node in a sub-menu is also a menu node, and it can also have a sub-menu. We still need to render a sub-menu node like what we do for a menu node, by using a control structure. The best thing is: we don't need to repeat ourselves in template `menu` . We can just apply the template `iterate` to iterate a collection of menu nodes recursively.

**All 3 templates are used in this example**

```
<template name="menu">
    <nav label="@load(menuItem.label)" iconSclass="@load(menuItem.iconSclass)">
        <apply template="iterate" menuItems="@ref(menuItem.subMenus)"/>
    </nav>
</template>
<template name="iterate">
    <forEach items="@load(menuItems)">
        <apply template="@load(empty each.subMenus?'menuitem':'menu')" menuItem="@ref(each)"/>
    </forEach>
</template>
<template name="menuitem" >
    <navitem label="@load(menuItem.label)" />
</template>
```

- Line 3: Apply the previous template `iterate` here to traverse each menu node and render them.

current:

Improve this Doc

# Navigation and Template

In traditional navigation, a user usually switches to different functions by visiting different pages of an application, a.k.a page-based navigation. In ZK, we can have another choice to design the navigation in AJAX-based, and users don't need to visit different pages. In page-based navigation, users need to switch pages frequently, we should maintain a consistent page design throughout whole application to help users keep track of where they are. Luckily, since 8.0 ZK provides **Template Injection** to keep multiple pages in the same style easily.

In this chapter, the example application we are going to build looks as follows:
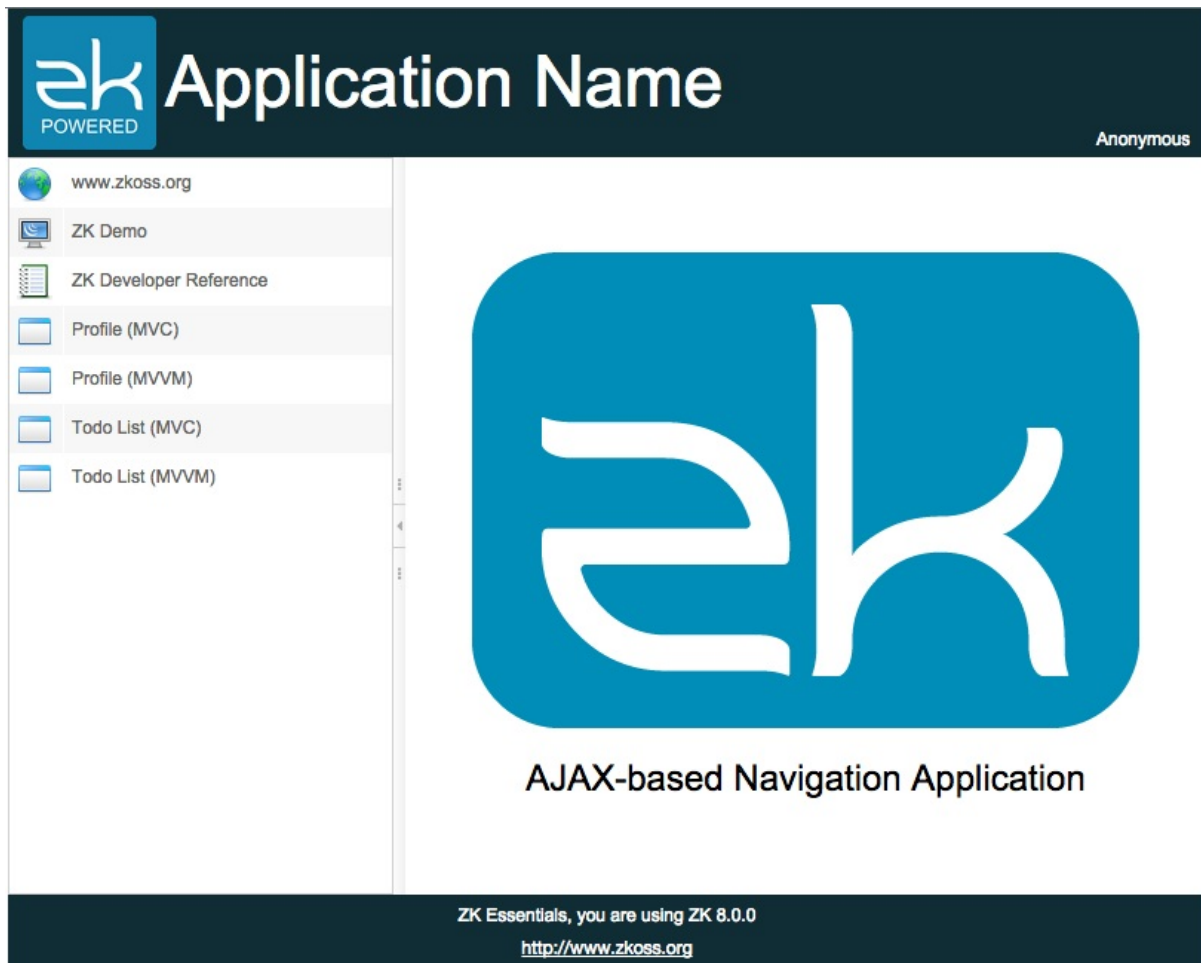


The sidebar is used for navigation control. There are 7 menu items on the sidebar, and the lower 4 items lead you to different functions. They only change the central area's content. All other areas are unchanged to maintain a consistent layout style during the navigation.

## Layout Template

In our example application, we want to keep a consistent layout for all functions. In this layout, the header, the sidebar, and the footer keep unchanged regardless of which function a user chooses. Only the central area changes its content according to the function chosen by users.

In page-based navigation, each function is put into a separated page, and we need to keep a consistent layout style. One way is to copy the unchanged part from one zul to another, but it is hard to maintain. Fortunately, ZK provides a *Template Injection* that lets you define a zul as a template and apply (inject) it to multiple zul pages afterwards. All zul pages that apply the same template have the same layout, so changing the template zul can change the layout of all pages once.

The steps to use a layout template are:

1. Create a zul as a template
2. Declare a template with a name in the target zul
3. Apply the template in the target zul with its name

Then when you visit the target zul page, ZK will insert the template to the position where you apply it upon template's name.

# 1. Create a Template ZUL

Creating a template zul is nothing different from creating a ordinary zul.

**chapter6/pagebased/layout/template.zul**

```
<zk>
    <borderlayout hflex="1" vflex="1">
        <north height="100px" border="none" >
            <include src="/chapter3/banner.zul"/>
        </north>
        <west width="260px" border="none" collapsible="true" splittable="true" minsize="300">
            <include src="/chapter6/pagebased/layout/sidebar.zul"/>
        </west>
        <center id="mainContent" autoscroll="true" border="none" >
            <apply template="center"/>
        </center>
        <south height="50px" border="none">
            <include src="/chapter3/footer.zul"/>
        </south>
    </borderlayout>
</zk>
```

- Line 10: Apply another template which is a changed part.

# 2. Declare the Template

After creating a template zul, we shoulde decalre a template by `<template>` before applying it like:

```
<template name="layout" src="/chapter6/pagebased/layout/template.zul"/>
```

This tag will decalre a template named `layout` with its source zul path.

# 3. Apply the Template

Then we can apply the template with `<apply>` , which is a *shadow component* introduced in ZK 8, and specify a template's name like:

```
<apply template="layout"/>
```

# Page-based Navigation

A traditional web application's navgation is usually designed as page-based. Each function corresponds to an independent page with independent URL. The navigation is very clear for users as they know where they are from the URL and they can press "go back" button on their browser to go back to previous pages in history. But the drawback is users have to wait whole page reloading every time they switch to a function. Additionally, developers also have to maintain multiple pages that have similar contents but applying a template zul can reduce this problem.



**Page-based Navigation**

To build a page-based navigation, first you should prepare corresponding pages for those items in the sidebar.

From below image, you can see there are 4 zul pages which correspond to items in the sidebar under "pagebased" folder (index-profile-mvc.zul, index-profile-mvvm.zul, index-todolist-mvc.zul, index-todolist-mvvm.zul). Then we can link four items of the sidebar to these zul pages by redirecting a browser.
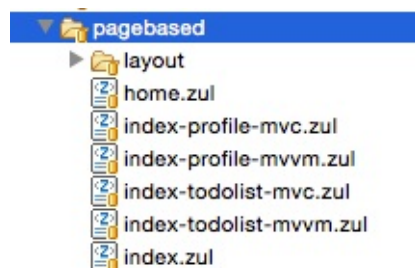


Next, we apply the template zul created before on those 4 pages. As you can see in previous `template.zul` , in order to inject different content in the `<center>` area, we apply a template named `center` that is not declared in the `template.zul` . Therefore, We can decalre `center` template with different path to inject different content with the same layout.

**/chapter6/pagebased/index-profile-mvc.zul**

```
<?link rel="stylesheet" type="text/css" href="/style.css"?>
```

```
<zk>
<div height="100%">
    <template name="layout" src="/chapter6/pagebased/layout/template.zul"/>
    <apply template="layout"/>
    <template name="center" src="/chapter5/profile-mvc.zul"/>
</div>
</zk>
```

- Line 6: Apply the template named 'layout'.
- Line 7: Declare a tempalte injected in `layout` template.

Our example application creates those menu items dynamically upon a configuration, so we should initialize configuration.

**Page-based navigation's sidebar configuration**

```
public class SidebarPageConfigPagebasedImpl implements SidebarPageConfig{

    HashMap<String,SidebarPage> pageMap = new LinkedHashMap<String,SidebarPage>();
    public SidebarPageConfigPagebasedImpl(){
        pageMap.put("zk",new SidebarPage("zk","www.zkoss.org","/imgs/site.png","http://www.zkoss.org/"));
        pageMap.put("demo",new SidebarPage("demo","ZK Demo","/imgs/demo.png","http://www.zkoss.org/zkdemo"));
        pageMap.put("devref",new SidebarPage("devref","ZK Developer Reference","/imgs/doc.png"
                ,"http://books.zkoss.org/wiki/ZK_Developer's_Reference"));

        pageMap.put("fn1",new SidebarPage("fn1","Profile (MVC)","/imgs/fn.png"
                ,"/chapter6/pagebased/index-profile-mvc.zul"));
        pageMap.put("fn2",new SidebarPage("fn2","Profile (MVVM)","/imgs/fn.png"
                ,"/chapter6/pagebased/index-profile-mvvm.zul"));
        pageMap.put("fn3",new SidebarPage("fn3","Todo List (MVC)","/imgs/fn.png"
                ,"/chapter6/pagebased/index-todolist-mvc.zul"));
        pageMap.put("fn4",new SidebarPage("fn4","Todo List (MVVM)","/imgs/fn.png"
                ,"/chapter6/pagebased/index-todolist-mvvm.zul"));
    }

    ...

}
```

- Line 10~17: Specify URL and related data for each menu item's configuration.

The following code shows how to redirect a user to an independent page when users click a menu item in the sidebar.

**Controller for page-based navigation**

```
public class SidebarPagebasedController extends SelectorComposer<Component>{

    ...

    //wire service
    SidebarPageConfig pageConfig = new SidebarPageConfigPagebasedImpl();

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

        //to initial view after view constructed.
        Rows rows = sidebar.getRows();

        for(SidebarPage page:pageConfig.getPages()){
            Row row = constructSidebarRow(page.getName(),page.getLabel(),page.getIconUri(),page.getUri());
            rows.appendChild(row);
        }
    }

    private Row constructSidebarRow(String name,String label, String imageSrc, final String locationUri) {
```

```
        //construct component and hierarchy
        Row row = new Row();
        Image image = new Image(imageSrc);
        Label lab = new Label(label);

        row.appendChild(image);
        row.appendChild(lab);

        //set style attribute
        row.setSclass("sidebar-fn");

        EventListener<Event> actionListener = new SerializableEventListener<Event>() {
            private static final long serialVersionUID = 1L;

            public void onEvent(Event event) throws Exception {
                //redirect current url to new location
                Executions.getCurrent().sendRedirect(locationUri);
            }
        };

        row.addEventListener(Events.ON_CLICK, actionListener);

        return row;
    }
}
```

- Line 15: Create menu items in the sidebar upon configurations with *Row*s.
- Line 39: Add an event listener to redirect a browser to the URL specified in the menu item a user clicks.

Visit http://localhost:8080/zkessentials/chapter6/pagebased/index.zul. You will see the URL changes and whole page reloads each time you click a different menu item.

# AJAX-based Navigation - MVC approach

When switching between different functions in paged-based navigation, you find that only central area's content is different among those pages and other three areas (header, sidebar, and footer) contain identical content. But in page-based navigation, a browser has to reload all contents no matter they are identical to previous page when switching to another function. With AJAX's help, ZK allows you to implement another navigation way that only updates necessary part of a page instead of reloading the whole page.
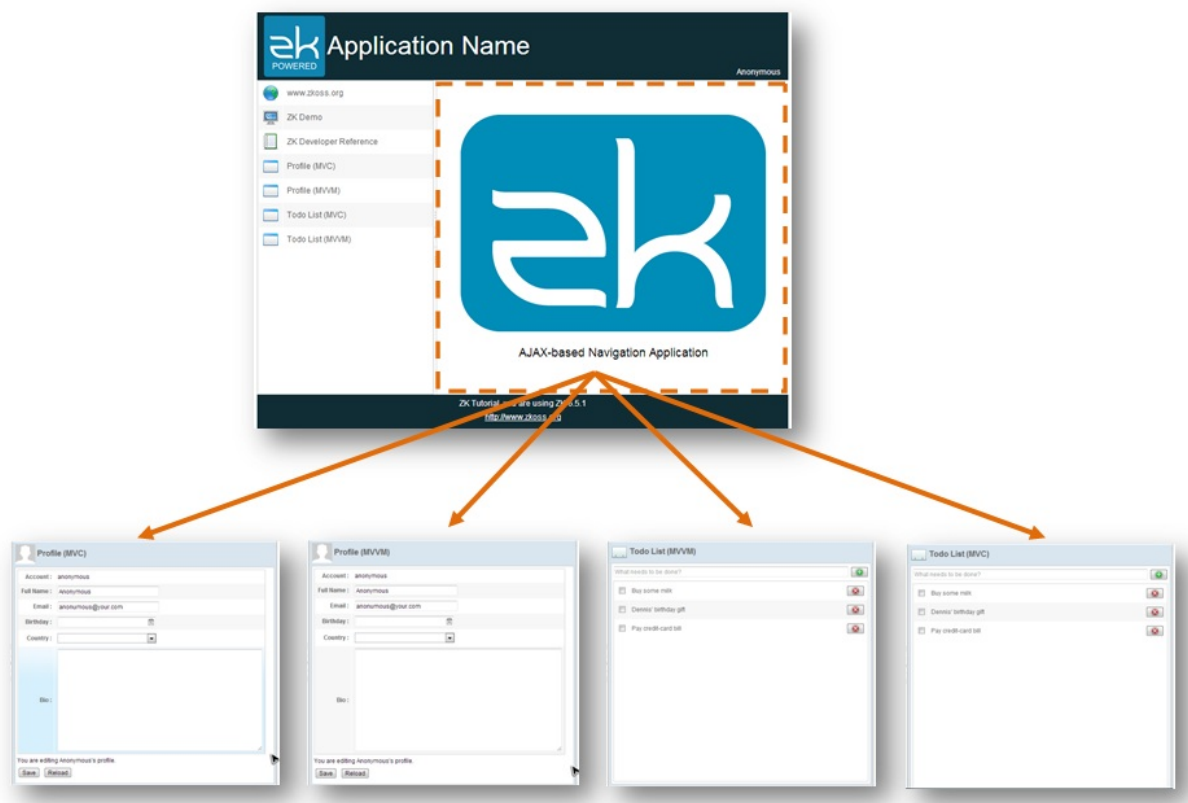


**AJAX-based Navigation**

The easiest way to implement AJAX-based navigation is to change *Include* component's `src` attribute. It can change only partial content (inside the *Include*) of a page instead of redirecting to another page to achieve the navigation purpose. This navigation way switches functions by only replacing a group of components instead of whole page and therefore has faster response than page-based one. But it doesn't change a browser's URL when each time switching to a different function. However, if you want users can keep track of different functions with bookmark, please refer to Browser History Management.

We will demonstrate AJAX-based navigation with the same layout example.

Below is the index page, its content is nearly the same as the index page of page based example except it replaces all `<include>` with `<apply>` .

**chapter6/ajaxbased/index.zul**

```
<?link rel="stylesheet" type="text/css" href="/style.css"?>
<zk>
    <borderlayout hflex="1" vflex="1" apply="org.zkoss.essentials.chapter6.ajaxbased.BookmarkChangeController">
        <north height="100px" border="none" >
            <apply templateURI="/chapter3/banner.zul"/>
```

```
        </north>
        <west width="260px" border="none" collapsible="true" splittable="true" minsize="300">
            <apply templateURI="/chapter6/ajaxbased/sidebar.zul"/>
        </west>
        <center id="mainContent" autoscroll="true" border="none">
            <apply templateURI="/chapter6/ajaxbased/mainContent.zul"/>
        </center>
        <south height="50px" border="none">
            <apply templateURI="/chapter3/footer.zul"/>
        </south>
    </borderlayout>
</zk>
```

**Replace `<include>` with `<apply>`**

As we don't need to dynamically change the path of those 3 areas (banner, side bar, footer), using `<apply>` is a better choice than `<include>` . Because `<apply>` create neither an extra `<div>` enclosin its content nor a id space. It's the main strength of using a shadow component that doesn't create a real component.

**chapter6/ajaxbased/mainContent.zul**

```
<zk>
    <include id="mainInclude"  src="/chapter6/ajaxbased/home.zul"/>
</zk>
```

- Line 2: We give the component id for we can find it later with ZK selector.

This navigation is mainly implemented by changing the `src` attribute of the *Include* component to switch between different zul pages so that it only reloads included components without affecting other areas . We still need to initialize sidebar configuration:

**AJAX-based navigation's sidebar configuration**

```
public class SidebarPageConfigAjaxbasedImpl implements SidebarPageConfig{

    HashMap<String,SidebarPage> pageMap = new LinkedHashMap<String,SidebarPage>();
    public SidebarPageConfigAjaxbasedImpl(){
        pageMap.put("zk",new SidebarPage("zk","www.zkoss.org","/imgs/site.png","http://www.zkoss.org/"));
        pageMap.put("demo",new SidebarPage("demo","ZK Demo","/imgs/demo.png","http://www.zkoss.org/zkdemo"));
        pageMap.put("devref",new SidebarPage("devref","ZK Developer Reference","/imgs/doc.png","http://books.zk
oss.org/wiki/ZK_Developer's_Reference"));

        pageMap.put("fn1",new SidebarPage("fn1","Profile (MVC)","/imgs/fn.png","/chapter3/profile-mvc.zul"));
        pageMap.put("fn2",new SidebarPage("fn2","Profile (MVVM)","/imgs/fn.png","/chapter3/profile-mvvm.zul"));
        pageMap.put("fn3",new SidebarPage("fn3","Todo List (MVC)","/imgs/fn.png","/chapter4/todolist-mvc.zul"))
;
        pageMap.put("fn4",new SidebarPage("fn4","Todo List (MVVM)","/imgs/fn.png","/chapter4/todolist-mvvm.zul"
));
    }
    ...
}
```

- Line 9 ~ 12: Because we only need those pages that doesn't have header, sidebar, and footer, we can re-use those pages written in previous chapters.

In the sidebar controller, we get the *Include* component and change its `src` according to the menu item's URL.

**Controller for AJAX-based navigation**

```
public class SidebarAjaxbasedController extends SelectorComposer<Component>{

    @Wire
    Grid sidebar;
```

```
    //wire service
    SidebarPageConfig pageConfig = new SidebarPageConfigAjaxbasedImpl();

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

        //to initial view after view constructed.
        Rows rows = sidebar.getRows();

        for(SidebarPage page:pageConfig.getPages()){
            Row row = constructSidebarRow(page.getName(),page.getLabel(),page.getIconUri(),page.getUri());
            rows.appendChild(row);
        }
    }


    private Row constructSidebarRow(final String name,String label, String imageSrc, final String locationUri)
 {

        //construct component and hierarchy
        Row row = new Row();
        Image image = new Image(imageSrc);
        Label lab = new Label(label);

        row.appendChild(image);
        row.appendChild(lab);

        //set style attribute
        row.setSclass("sidebar-fn");

        //new and register listener for events
        EventListener<Event> onActionListener = new SerializableEventListener<Event>(){
            private static final long serialVersionUID = 1L;

            public void onEvent(Event event) throws Exception {
                //redirect current url to new location
                if(locationUri.startsWith("http")){
                    //open a new browser tab
                    Executions.getCurrent().sendRedirect(locationUri);
                }else{
                    //use iterable to find the first include only
                    Include include = (Include)Selectors.iterable(sidebar.getPage(), "#mainInclude")
                            .iterator().next();
                    include.setSrc(locationUri);

                    ...
                }
            }
        };
        row.addEventListener(Events.ON_CLICK, onActionListener);

        return row;
    }

 }
```

- Line 46,47: Since *Include* is not a child component of the component that `SidebarAjaxbasedController` applies to, we cannot use `@Wire` to retrieve it. Therefore, we use `Selectors.iterable()` to get components with the id selector from the page. Because ZK combines included components and its parent into one ZK page.
- Line 48: Change the `src` to the corresponding URL that belongs to the clicked menu item.

Visit the http://localhost:8080/zkessentials/chapter6/ajaxbased/index.zul to see the result.

# AJAX-based Navigation - MVVM approach

Surely we can also implement AJAX-based navigation in MVVM approach. Every page are quite similar with previous section. Just switch different pages with data binding.

In this example, we modularize each page with an independent ViewModel. In order to communicate between sidebar and content area that are bound with different ViewModels, we need to use global command binding. You can treat it as a command binding mentioned in previous chapter, but it can invoke a command method declared in other ViewModels.

The basic idea is: sidebar sends a global command when a user clicks an item then content area change *Include* component's `src` attribute to navigate pages.

**chapter6/ajaxbased_mvvm/sidebar.zul**

```
<rows>
    <template name="model">
        <row sclass="sidebar-fn" onClick="@global-command('onNavigate', page=each)">
            <image src="@load(each.iconUri)"/>
            <label value="@load(each.label)"/>
        </row>
    </template>
</rows>
```

- Line 3: Specify a global command binding and pass `SidebarPage` as a parameter

**chapter6/ajaxbased_mvvm/mainContent.zul**

```
<zk>
    <include id="mainInclude"
        viewModel="@id('vm') @init('org.zkoss.essentials.chapter6.ajaxbased.mvvm.NavigationViewModel')"
        src="@load(vm.includeSrc)" />
</zk>
```

The code below demonstrate how to declare a global command method and receive a parameters.

```
package org.zkoss.essentials.chapter6.ajaxbased.mvvm;

import org.zkoss.bind.annotation.BindingParam;
import org.zkoss.bind.annotation.GlobalCommand;
import org.zkoss.bind.annotation.NotifyChange;
import org.zkoss.essentials.services.SidebarPage;
import org.zkoss.zk.ui.Executions;

public class NavigationViewModel {

    private String includeSrc = "/chapter6/ajaxbased/home.zul";

    @GlobalCommand("onNavigate")
    @NotifyChange("includeSrc")
    public void onNavigate(@BindingParam("page") SidebarPage page) {
        String locationUri = page.getUri();
        String name = page.getName();

        //redirect current url to new location
        if(locationUri.startsWith("http")){
            //open a new browser tab
            Executions.getCurrent().sendRedirect(locationUri);
```

```java
        } else {
            includeSrc = locationUri;

            //advance bookmark control,
            //bookmark with a prefix
            if(name!=null){
                Executions.getCurrent().getDesktop().setBookmark("p_"+name);
            }
        }
    }

    public String getIncludeSrc() {
        return includeSrc;
    }

}
```

# Authentication

In this chapter, we will demonstrate how to implement authentication and protect your pages from illegal access. We will create a login page without a sidebar as follows:


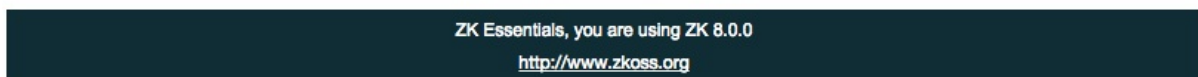
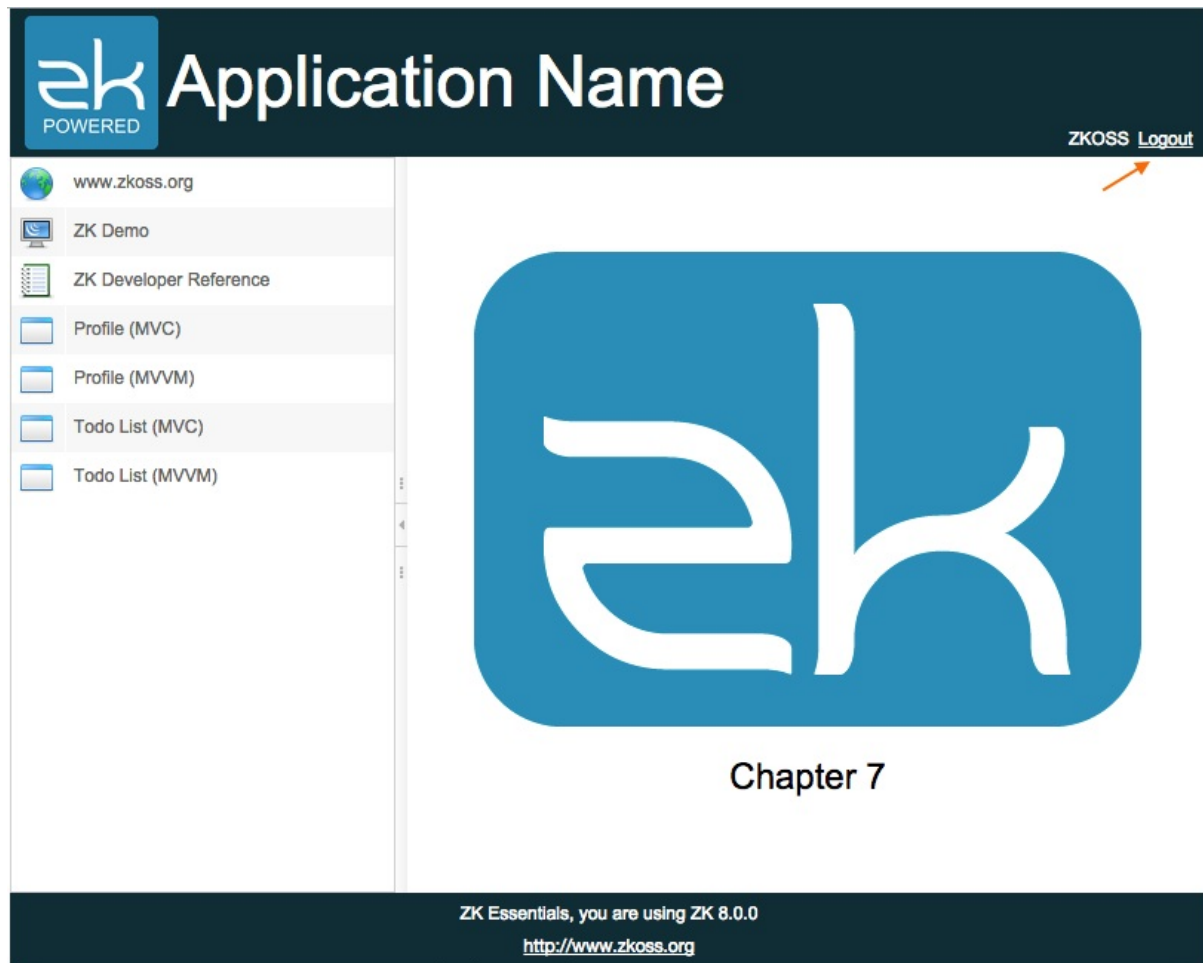After login, we redirect users to the index page and display the user name on the right side of the header.

For third party framework integration approach to secure your application, please refer to ZK Deverloper's Reference/Integration/Security

# Session

Before proceeding to implement the authentication function, we have to understand a "session" first. A web application operates over HTTP protocol which is stateless; each request and its corresponding response is handled independently. Hence, a HTTP server cannot know whether a series of requests is sent from the same client or from different clients. That means the server cannot maintain client's state between multiple requests.

Application servers maintain a *session* to keep a client's state. When the server receives the first request from a client, the server creates a session and give the session a unique identifier. The client should send a request with the session identifier. The server can determine which session the request belongs to.

In a Java EE environment, an application server creates a `javax.servlet.http.HttpSession` object to track client's session. ZK's `org.zkoss.zk.ui.Session` is a wrapper of `HttpSession`, you can use it to store user's data when you handle events. The usage:

- Get current session: `Sessions.getCurrent()`
- Store data into a session: `Session.setAttribute("key", data)`
- Retrieve data from a session: `Session.getAttribute("key")`

Almost all business applications require a security mechanism and authentication is the fundamental part of it. Some resources are only available to those authenticated users. Authentication is a process to verify that a user is who he claims to be. After we authenticate a user, the application needs to remember the user and identifies his subsequent requests so that the application doesn't need to authenticate him repeatedly for each further request. Hence, storing user credentials in a session is good practice and we can tell whom the request belongs to by the request session. Additionally, we can't tell whether a request comes from an authenticated user by checking a user's credentials in the request's session.

current:

Improve this Doc

# Secure Your Pages

Even if you have implemented an authentication mechanism, a user still can access a page if he knows the page's URL. Therefore, we should protect a page from illegal access by checking user's credentials in his session when a page is requested by a user.

## Authentication Service

We can implement a service class that performs the authentication operations.

**Get user credentials**

```java
public class AuthenticationServiceChapter3Impl implements AuthenticationService,Serializable{
    public UserCredential getUserCredential(){
        Session sess = Sessions.getCurrent();
        UserCredential cre = (UserCredential)sess.getAttribute("userCredential");
        if(cre==null){
            cre = new UserCredential();//new a anonymous user and set to session
            sess.setAttribute("userCredential",cre);
        }
        return cre;
    }
...
}
```

- Line 3, 4: As we mentioned, we can get current session and check user's credentials to verify its authentication status.

**Log in/ log out**

```java
public class AuthenticationServiceChapter7Impl extends AuthenticationServiceChapter5Impl{

    UserInfoService userInfoService = new UserInfoServiceChapter3Impl();

    @Override
    public boolean login(String nm, String pd) {
        User user = userInfoService.findUser(nm);
        //a simple plan text password verification
        if(user==null || !user.getPassword().equals(pd)){
            return false;
        }

        Session sess = Sessions.getCurrent();
        UserCredential cre = new UserCredential(user.getAccount(),user.getFullName());
        ...
        sess.setAttribute("userCredential",cre);


        return true;
    }

    @Override
    public void logout() {
        Session sess = Sessions.getCurrent();
        sess.removeAttribute("userCredential");
    }
    ...
}
```

- Line 14: Get the current session.
- Line 17: Store user credentials into the session with a key `userCredential` which is used to retrieve credential back in the future.
- Line 26: Remove the stored user credentials in the session.

# Page Initialization

ZK allows you to run some code before ZK Loader instantiates any component by implementing an `org.zkoss.zk.ui.util.Initiator` . When we apply an initiator to a zul, ZK will invoke our initiator before creating components.

We can create an initiator to check existence of a user's credentials in the session. If a user's credentials is absent, we determine it's an illegal request and redirect it back to the login page.

**Page initiator to check a user's credentials**

```java
public class AuthenticationInit implements Initiator {

    //services
    AuthenticationService authService = new AuthenticationServiceChapter7Impl();

    public void doInit(Page page, Map<String, Object> args) throws Exception {

        UserCredential cre = authService.getUserCredential();
        if(cre==null || cre.isAnonymous()){
            Executions.sendRedirect("/chapter7/login.zul");
            return;
        }
    }
}
```

- Line 1, 6: A page initiator class should implement `org.zkoss.zk.ui.util.Initiator` and override `doInit()` .
- Line 8: Get a user's credentials from current session.
- Line 10: Redirect users back to login page.

Then we can apply this page initiator to those pages we want to protect from unauthenticated access.

**chapter7/index.zul**

```xml
<?link rel="stylesheet" type="text/css" href="/style.css"?>
<!-- protect page by the authentication init  -->
<?init class="org.zkoss.essentials.chapter7.AuthenticationInit"?>
```

- Line 3: Because index.zul is the main page, we apply this page initiator on it.

After above steps are complete, if you directly visit http://localhost:8080/zkessentials/chapter7/index.zul without authentication, you will be redirected to the login page.

# Login

It is a common way to request an account and a password for authentication. We create a login page to collect user's account and password, and the login page also uses a template zul to keep a consistent style with the index page. However, there should be no sidebar because users without logging in should not be able to access main functions. So we need to change our template to create the sidebar according to user credential in the session with a shadow component, `<if>` .

**chapter7/layout/template.zul**

```xml
<zk>
    <borderlayout hflex="1" vflex="1" >
        <north height="100px" border="none" >
            <apply templateURI="/chapter7/layout/banner.zul"/>
        </north>
    <!-- create only when the currentUser is not an anonymous  -->
        <if test="${not sessionScope.userCredential.anonymous}">
        <west width="260px" border="none" collapsible="true" splittable="true" minsize="300">
            <apply templateURI="/chapter6/ajaxbased/sidebar.zul"/>
        </west>
        </if>
        <center id="mainContent" autoscroll="true" border="none">
            <!-- the main content will be insert to here -->
            <apply template="center"/>
        </center>
        <south height="50px" border="none">
            <apply templateURI="/chapter1/footer.zul"/>
        </south>
    </borderlayout>
</zk>
```

- Line 7: Determine if there is an anonymous user credential in the session by EL.

The login form is built with *Grid*. This page should be accessible for all users, so we don't have to apply `AuthenticationInit` .

**chapter7/login.zul**

```xml
<?link rel="stylesheet" type="text/css" href="/style.css"?>
<zk>
<div height="100%">
    <template name="layout" src="/chapter7/layout/template.zul"/>
    <apply template="layout"/>
    <template name="center">
        <hbox vflex="1" hflex="1" align="center"
            pack="center" spacing="20px">
            <vlayout>
                <window id="loginWin"
                    apply="org.zkoss.essentials.chapter7.LoginController"
                    title="Login with you name" border="normal" hflex="min">
                    <vbox hflex="min" align="center">
                        <grid hflex="min">
                            <columns>
                                <column hflex="min" align="right" />
                                <column />
                            </columns>
                            <rows>
                                <row>
                                    Account :
                                    <textbox id="account" width="200px" />
                                </row>
```

```
                                    <row>
                                        Password :
                                        <textbox id="password" type="password"
                                            width="200px" />
                                    </row>
                                </rows>
                            </grid>
                            <label id="message" sclass="warn" value=" " />
                            <button id="login" label="Login" />

                        </vbox>
                    </window>
                    (use account='zkoss' and password='1234' to login)
                </vlayout>
            </hbox>
        </template>
</div>
</zk>
```

- Line 6: Define a template named `cetner` which is used in `template.zul` .
- Line 26: Specify "password" at `type` , then user input will be masked.

This login controller collects the account and password and validates them with an authentication service class. If the password is correct, the authentication service class saves user's credentials into the session.

**Controller used in chapter7/login.zul**

```
public class LoginController extends SelectorComposer<Component> {

    //wire components
    @Wire
    Textbox account;
    @Wire
    Textbox password;
    @Wire
    Label message;

    //services
    AuthenticationService authService = new AuthenticationServiceChapter7Impl();


    @Listen("onClick=#login; onOK=#loginWin")
    public void doLogin(){
        String nm = account.getValue();
        String pd = password.getValue();

        if(!authService.login(nm,pd)){
            message.setValue("account or password are not correct.");
            return;
        }
        UserCredential cre= authService.getUserCredential();
        message.setValue("Welcome, "+cre.getName());
        message.setSclass("");

        Executions.sendRedirect("/chapter7/");

    }
}
```

- Line 20: Authenticate a user with account and password and save user's credential into the session if it passes.
- Line 28: Redirect to index page after successfully authenticated.

After login, we want to display a user's account in the banner. We can use EL to get a user's account from `UserCredential` in the session.

**chapter7/layout/banner.zul**

```
<div hflex="1" vflex="1" sclass="banner">
    <hbox hflex="1" vflex="1" align="center">
        <!-- other components -->

        <hbox apply="org.zkoss.essentials.chapter7.LogoutController"
            hflex="1" vflex="1" pack="end" align="end" >
            <label value="${sessionScope.userCredential.name}"
                if="${not sessionScope.userCredential.anonymous}"/>
            <label id="logout" value="Logout"
            if="${not sessionScope.userCredential.anonymous}" sclass="logout"/>
        </hbox>
    </hbox>
</div>
```

- Line 7: The sessionScope/sessionScope) is an implicit object that you can use within EL to access session's attribute. It works as the same as `getAttribute()` . You can use it to get session's attribute with dot notation, e.g. `${sessionScope.userCredential}` equals to calling `getAttribute("userCredential")` of a `Session` object.

# Logout

When a user logs out, we usually clear his credentials from the session and redirect him to the login page. In our example, clicking "Logout" label in the banner can log you out.

**chpater7/layout/banner.zul**

```xml
<div hflex="1" vflex="1" sclass="banner" >
    <hbox hflex="1" vflex="1" align="center">
        <!-- other components -->

        <hbox apply="org.zkoss.essentials.chapter7.LogoutController"
            hflex="1" vflex="1" pack="end" align="end" >
            <label value="${sessionScope.userCredential.name}"
                if="${not sessionScope.userCredential.anonymous}"/>
            <label id="logout" value="Logout"
            if="${not sessionScope.userCredential.anonymous}" sclass="logout"/>
        </hbox>
    </hbox>
</div>
```

- Line 9, 10: We listen `onClick` event on logout label to perform logout action.

**LogoutController.java**

```java
public class LogoutController extends SelectorComposer<Component> {

    //services
    AuthenticationService authService = new AuthenticationServiceChapter7Impl();

    @Listen("onClick=#logout")
    public void doLogout(){
        authService.logout();
        Executions.sendRedirect("/chapter7/");
    }
}
```

- Line 8: Call service class to perform logout.
- Line 9: Redirect users to login page.

After completing the above steps, you can visit http://localhost:8080/zkessentials/chapter7 to see the result.

# Spring Integration

The Spring Framework is a popular application development framework for enterprise Java. One key element is its infrastructural support: a light-weighted IoC (Inversion of Control) container that manages POJOs as Spring beans and their dependency relationship.

The most common integration way is to let Spring manage class dependencies of an application. When a class "A" references a class "B" and calls B's method, we say that A depends on B. In examples of previous chapters, we create this dependency by instantiating B class in A class as follows:

```
public class ProfileViewController extends SelectorComposer<Component>{

    AuthenticationService authService = new AuthenticationServiceChapter8Impl();
    ...
}
```

- `ProfileViewController` depends on `AuthenticationService` .

Spring can help us manage these dependencies without instantiating dependent classes manually. In this chapter, we won't create new example applications but will make previous examples integrate with Spring.

## Source Code

As we mentioned in the Introduction, our source code has 3 branches in github. The source code of this chapter's example belongs to the branch: **zk8-spring**. You can select the branch and click "zip" icon to download as a zip.

We don't create new examples in this chapter, but we re-organize some classes. You can see from the image below. We move all service class implementations to the package `org.zkoss.essentials.services.impl` .

current:

Improve this Doc

# Configuration

## Maven

In order to integrate our ZK application with Spring, we must add dependencies for Spring. The cglib is an optional dependency and we add it because our application uses Spring's scoped-proxy that requires it.

**Extracted from pom.xml**

```xml
<properties>
    <zk.version>8.0.0-Eval</zk.version>
    <maven.build.timestamp.format>yyyy-MM-dd</maven.build.timestamp.format>
    <packname>-${project.version}-FL-${maven.build.timestamp}</packname>
    <spring.version>3.1.2.RELEASE</spring.version>
</properties>
...
    <!-- Spring 3 dependencies -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>cglib</groupId>
        <artifactId>cglib</artifactId>
        <version>2.2.2</version>
    </dependency>
```

## Deployment Descriptor

The deployment descriptor (web.xml) also needs two more listeners from Spring.

**Extracted from web.xml**

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

<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <description><![CDATA[ZK Essentials]]></description>
    <display-name>ZK Essentials</display-name>

    <!-- ZK configuration-->
    ...

    <!-- Spring configuration -->
    <!-- Initialize spring context -->
    <listener>
        <listener-class>
        org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <!-- Enable webapp Scopes-->
     <listener>
        <listener-class>
```

```
            org.springframework.web.context.request.RequestContextListener
        </listener-class>
    </listener>

    <welcome-file-list>
        <welcome-file>index.zul</welcome-file>
    </welcome-file-list>
</web-app>
```

- Line 16,17: The `ContextLoaderListener` reads Spring configuration, and default location is `WEB-INF/applicationContext.xml`.
- Line 21,22: Use `RequestContextListener` to support web-scoped beans ( `request` , `session` , `global session` ).

# Spring Configuration File

Create Spring configuration file with default name ( `applicationContext.xml` ). We enable Spring's classpath scanning to detect and register those class with annotation as beans automatically.

**WEB-INF/applicationContext.xml**

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:component-scan base-package="org.zkoss.essentials" />

</beans>
```

- Line 10: This configuration enables classpath scanning. Spring will automatically detect those classes with Spring bean annotations and register them in bean definitions. You should specify a common parent package or a comma-separated package list that includes all candidate classes at `base-package` attribute.

current:

Improve this Doc

# Register Spring Beans

Starting from 2.0, Spring provides an option to detect beans by scanning the classpath. Developers can use annotations (e.g. `@Component` ) to register bean definitions in the Spring container and this removes the use of XML. We can use `@Component` which is a generic stereotype annotation or those specialized stereotype annotation: `@Controller` , `@Service` , or `@Repository` for presentation, service, persistence layer, respectively. These annotations work equally for registering beans but using specialized annotation makes your classes suited for processing by tools.

When you register a bean, its bean scope is a "singleton" by default if you don't specify it. Our service class is stateless so that it is suitable to be a singleton-scoped bean. For those beans used in composers, they should use scoped-proxy to ensure every time Spring will retrieve them when a composer uses them. (Please use scoped-proxy even for a singleton scoped bean, because scope of Spring beans doesn't match scope of composers. Scoped-proxy can ensure composers get the latest bean under their context. For furthermore explanation, please refer to Developer's Reference/Integration/Middleware Layer/Spring )

```
@Service("authService")
@Scope(value="singleton",proxyMode=ScopedProxyMode.TARGET_CLASS)
public class AuthenticationServiceImpl implements AuthenticationService,Serializable{
...
}
```

- Line 1: You could specify bean's name in `@Service` or its bean is derived from class name with first character in lower case (e.g. `authenticationServiceImpl` in this case).
- Line 2: If you want to specify a bean's scope, use `@Scope` . For those beans used in composers, you should use scoped-proxy to ensure every time you get the latest bean.

# Wire Spring Beans

After registering beans for service classes, we can "wire" them in our controllers with ZK's variable resolver. To wire a Spring bean in a composer, we need to apply a `org.zkoss.zkplus.spring.DelegatingVariableResolver`. Then, we can apply annotation `@WireVariable` on a variable which we want to wire a Spring bean with. ZK will then wire the corresponding Spring bean with **the variable whose name is the same as the bean's name**. Alternatively, you can specify the bean's name with `@WireVariable("beanName")`.

You might think why don't we just register our controllers(or ViewModels) as Spring beans, so that we can use Spring's `@Autowire`. We don't recommend to do so. The main reason is that none of Spring bean's scope matches ZK's composer's life cycle, for details please refer to Developer's Reference.

**Wire beans in a composer**

```
@VariableResolver(org.zkoss.zkplus.spring.DelegatingVariableResolver.class)
public class SidebarChapter4Controller extends SelectorComposer<Component>{

    private static final long serialVersionUID = 1L;

    //wire components
    @Wire
    Grid sidebar;

    //wire service
    @WireVariable("sidebarPageConfigPagebase")
    SidebarPageConfig pageConfig;

    ...
}
```

- Line 11: Specify bean's name `sidebarPageConfigPagebase`

**Wire beans in a ViewModel**

```
@VariableResolver(org.zkoss.zkplus.spring.DelegatingVariableResolver.class)
public class ProfileViewModel implements Serializable{
    private static final long serialVersionUID = 1L;

    //wire services
    @WireVariable
    AuthenticationService authService;
    @WireVariable
    UserInfoService userInfoService;
    ...
}
```

- Line 6: Wire a Spring bean whose bean name is `authService`.

## Wire Manually

When using `@WireVariable` out of a composer (or a ViewModel), ZK will not wire Spring beans for you automatically. If you need to get a Spring bean, you can wire them manually. The example below wires a Spring bean in a page initiator:

```
@VariableResolver(org.zkoss.zkplus.spring.DelegatingVariableResolver.class)
```

```
public class AuthenticationInit implements Initiator {

    @WireVariable
    AuthenticationService authService;

    public void doInit(Page page, Map<String, Object> args) throws Exception {
        //wire service manually by calling Selectors API
        Selectors.wireVariables(page, this, Selectors.newVariableResolvers(getClass(), null));

        UserCredential cre = authService.getUserCredential();
        if(cre==null || cre.isAnonymous()){
            Executions.sendRedirect("/chapter8/login.zul");
            return;
        }
    }
}
```

- Line 9: After applying `@VariableResolver` and `@WireVariable` , use org.zkoss.zk.ui.select.Selectors to wire Spring beans manually.

After completing above steps, integration of Spring is done. The application's appearance doesn't change, but its infrastructure is now managed by Spring. You can visit http://localhost:8080/zkessentials to see the result.

# JPA Integration

In previous chapters, we mimic a database with a static list as follows:

```java
public class TodoListServiceChapter6Impl implements TodoListService {

    static int todoId = 0;
    static List<Todo> todoList = new ArrayList<Todo>();
    static{
        todoList.add(new Todo(todoId++,"Buy some milk",Priority.LOW,null,null));
        todoList.add(new Todo(todoId++,"Dennis' birthday gift",Priority.MEDIUM,dayAfter(10),null));
        todoList.add(new Todo(todoId++,"Pay credit-card bill",Priority.HIGH,dayAfter(5),"$1,000"));
    }

    /** synchronized is just because we use static userList in this demo to prevent concurrent access **/
    public synchronized List<Todo>getTodoList() {
        List<Todo> list = new ArrayList<Todo>();
        for(Todo todo:todoList){
            list.add(Todo.clone(todo));
        }
        return list;
    }

...
}
```

- Line 4, 12: Store objects in a static list and perform all data operation on it.

Originally we perform all persistence operations on the list, but we will replace this part with a real database and a persistence framework, *JPA*.

*Java Persistence API* (JPA) is a POJO-based persistence specification. It offers an *object-relational mapping* solution to enterprise Java applications. In this chapter, we don't create new applications but re-write data persistence part based on chapter 9 with JPA. We will create a simple database with HSQL and implement a persistence layer using the DAO (Data Access Object) pattern to encapsulate all database related operations. We also have to annotate all entity classes that will be stored in the database with JPA annotations. To make the example close to a real application, we keep using the Spring framework and demonstrate how to integrate Spring with JPA.

## Source Code

As we mentioned in the Project Struecture, our source code has 3 branches in github. The source code of this chapter's example belongs to the branch: **zk8-jpa**.

We don't create new examples in this chapter, but we add 2 DAO classes implemented in JPA under the package `org.zkoss.essentials.services.impl` .

# Configuration

## Maven

When using a database, JPA, and integration of JPA and Spring, we should add following dependencies based on chapter 9's configuration: (We add `spring-web` and `cblib` for Spring framework which is explained in previous chapter.)

```
<properties>
    <zk.version>8.0.0-Eval</zk.version>
    <maven.build.timestamp.format>yyyy-MM-dd</maven.build.timestamp.format>
    <packname>-${project.version}-FL-${maven.build.timestamp}</packname>
    <spring.version>3.1.2.RELEASE</spring.version>
</properties>

...

    <!-- Spring 3 dependencies -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>cglib</groupId>
        <artifactId>cglib</artifactId>
        <version>2.2.2</version>
    </dependency>
    <!-- JPA(Hibernate) and HSQL dependencies -->
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-entitymanager</artifactId>
        <version>4.0.0.Final</version>
    </dependency>
    <dependency>
        <groupId>org.hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <version>2.2.6</version>
    </dependency>
...
```

- Line 5, 16~17: Spring provides a module to integrate several ORM (Object Relation Mapping) frameworks, integrating JPA requires this dependency.
- Line 27~28: There are various implementations of JPA specification, we choose Hibernate's one which is the most popular.
- Line 32~33: For using HSQL, we should add its JDBC driver.

## Persistence Unit Configuration

We should describe persistence unit configuration in an XML file called `persistence.xml` and we need to specify `name`, `transaction-type`, and `properties`. The `properties` are used by persistence provider (Hibernate) to establish database connection and setup vendor specific configurations.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">
    <persistence-unit name="myapp" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <properties>
            <property name="hibernate.dialect"
                        value="org.hibernate.dialect.HSQLDialect" />
            <property name="hibernate.connection.driver_class"
                        value="org.hsqldb.jdbcDriver" />
            <property name="hibernate.connection.username" value="sa" />
            <property name="hibernate.connection.password" value="" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.connection.url"
                        value="jdbc:hsqldb:file:data/store" />
            <property name="hibernate.hbm2ddl.auto" value="create" />
        </properties>
    </persistence-unit>
</persistence>
```

- Line 3: The persistence unit name `myapp` will be used later in Spring configuration.

# Deployment Descriptor

You aren't required to add any special configuration for JPA to work. Here we add a Spring provided `OpernEntityMangerInViewFilter` to resolve an issue caused by lazy-fetching in one-to-many mapping. Please refer to Developer's Reference for this issue in more detail.

**Extracted from web.xml**

```xml
...
    <!-- Spring configuration -->
    <!-- Initialize spring context -->
    <listener>
        <listener-class>
        org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <!-- Enable webapp Scopes-->
     <listener>
        <listener-class>
        org.springframework.web.context.request.RequestContextListener
        </listener-class>
    </listener>

    <filter>
        <filter-name>OpenEntityManagerInViewFilter
        </filter-name>
        <filter-class>
        org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>OpenEntityManagerInViewFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
...
```

# Entity Annotation

Before storing objects into a database, we should specify OR (object-relation) mapping for Java classes with meta data. After JDK 5.0, we can specify OR mapping as annotations instead of XML files. JPA supports *configuration by exception* which means that it defines default for most cases of application and users only need to override the configuration value when it is exception to the default, not necessary.

First, annotate the class with `@Entity` to turn it into an entity, and annotate the member field for primary key with `@Id` . All other annotations are optional and we use them to override default values.

**Todo class used in todo-list management**

```java
@Entity
@Table(name="apptodo")
public class Todo implements Serializable, Cloneable {


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

    boolean complete;

    @Column(nullable=false,length=128)
    String subject;

    @Column(nullable=false,length=128)
    @Enumerated(EnumType.ORDINAL)
    Priority priority;

    @Temporal(TemporalType.TIMESTAMP)
    Date date;

    String description;

    //omit getter, setter, hashCode(), equals() and other methods

}
```

# Spring Beans Configuration

Spring JPA , available under the `org.springframework.orm.jpa` package, offers integration support for Java Persistence API. According to Spring framework reference, there are three options for JPA setup. We choose the simplest one.

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

    <context:component-scan base-package="org.zkoss.essentials" />

    <!-- jpa(hibernate) configuration -->
    <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="myapp"/>
    </bean>

    <bean id="transactionManager"
```

```
        class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>

    <tx:annotation-driven />
</beans>
```

- Line 16,17: The simplest Spring setup for JPA is to add a `LocalEntityManagerFactoryBean` which create a `EntityManagerFactory` for simple deployment environments and specify its `persistenceUnitName` property.
- Line 18: Set property `persistenceUnitName` with the name we specified in `persistence.xml` .
- Line 21,26: Enable Spring's declarative transaction management.

# DAO Implementation

The *Data Access Object (DAO)* pattern is a good practice to implement a persistence layer and it encapsulates data access codes from the business tier. A DAO object exposes an interface to a business object and performs persistence operation relating to a particular persistent entity. Now, we can implement persistence operations like CRUD in a DAO class with JPA and `EntityManager` injected by Spring.

```java
@Repository
public class TodoDao {

    @PersistenceContext
    private EntityManager em;

    @Transactional(readOnly=true)
    public List<Todo> queryAll() {
        Query query = em.createQuery("SELECT t FROM Todo t");
        List<Todo> result = query.getResultList();
        return result;
    }

    ...

    @Transactional
    public Todo save(Todo todo){
        em.persist(todo);
        return todo;
    }

    @Transactional
    public Todo update(Todo todo){
        todo = em.merge(todo);
        return todo;
    }

    @Transactional
    public void delete(Todo todo){
        Todo r = get(todo.getId());
        if(r!=null){
            em.remove(r);
        }
    }
}
```

- Line 1: We register `TodoDao` as a Spring bean with `@Repository` because it is a DAO class according to Spring's suggestion.
- Line 4: As Spring manages our entity manager factory, it can understand `@PersistenceContext` and inject a transaction scope `EntityManager` for us. Hence, we don't need to create `EntityManager` by our own.
- Line 7: We have enabled Spring's declarative transaction management so that we can apply `@Transactional` on a methods.

After completing DAO classes, we can inject them to our service class with Spring's `@Autowired` because they are all Spring beans.

```java
@Service("todoListService")
@Scope(value="singleton",proxyMode=ScopedProxyMode.TARGET_CLASS)
public class TodoListServiceImpl implements TodoListService {

    @Autowired
    TodoDao dao;
```

```
    public List<Todo>getTodoList() {
        return dao.queryAll();
    }

    ...
}
```

Completing the above steps, we have created a dependency relationship among the controller, service, and persistence classes as follows:



Each of these classes encapsulates cohesive functions and has decoupled relationships with others. You can easily expand the architecture by adding more classes or create dependencies between two layers.

You can visit http://localhost:8080/zkessentials to see the result.

current:
Improve this Doc

# Conclusion

Our book ends here. But the adventure toward ZK just begins. This book opens you a door and introduces you some basic concepts and usages of ZK. You can start to deploy ZK to your server according to ZK Installation Guide which contains information you will use in deploying to web servers. For further development help, the ZK Developer's Reference contains complete references for various topics in developing ZK based applications (i.e. for this chapter: Middleware Layer/Spring or Persistence Layer/JPA. ZUML Reference describes syntax and EL expression used in a zul. If you want to know details of a component, please refer to the ZK Component Reference. ZK also provides lots of configuration, you can take a look at ZK Configuration Reference.

We hope you can make use of what you learn here to obtain an even greater knowledge of ZK.