

ZK Essentials

For ZK 7

Contents

Articles

ZK Essentials	1
Chapter 1: Introduction	1
Chapter 2: Project Structure	6
Chapter 3: User Interface and Layout	9
Chapter 4: Controlling Components	15
Chapter 5: Handling User Input	21
Chapter 6: Implementing CRUD	41
Chapter 7: Navigation and Templating	61
Chapter 8: Authentication	71
Chapter 9: Spring Integration	79
Chapter 10: JPA Integration	85

References

Article Sources and Contributors	93
Image Sources, Licenses and Contributors	94

ZK Essentials

Documentation:Books/ZK_Essentials

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

<comment>http://books.zkoss.org/wiki/ZK_Essentials</comment>

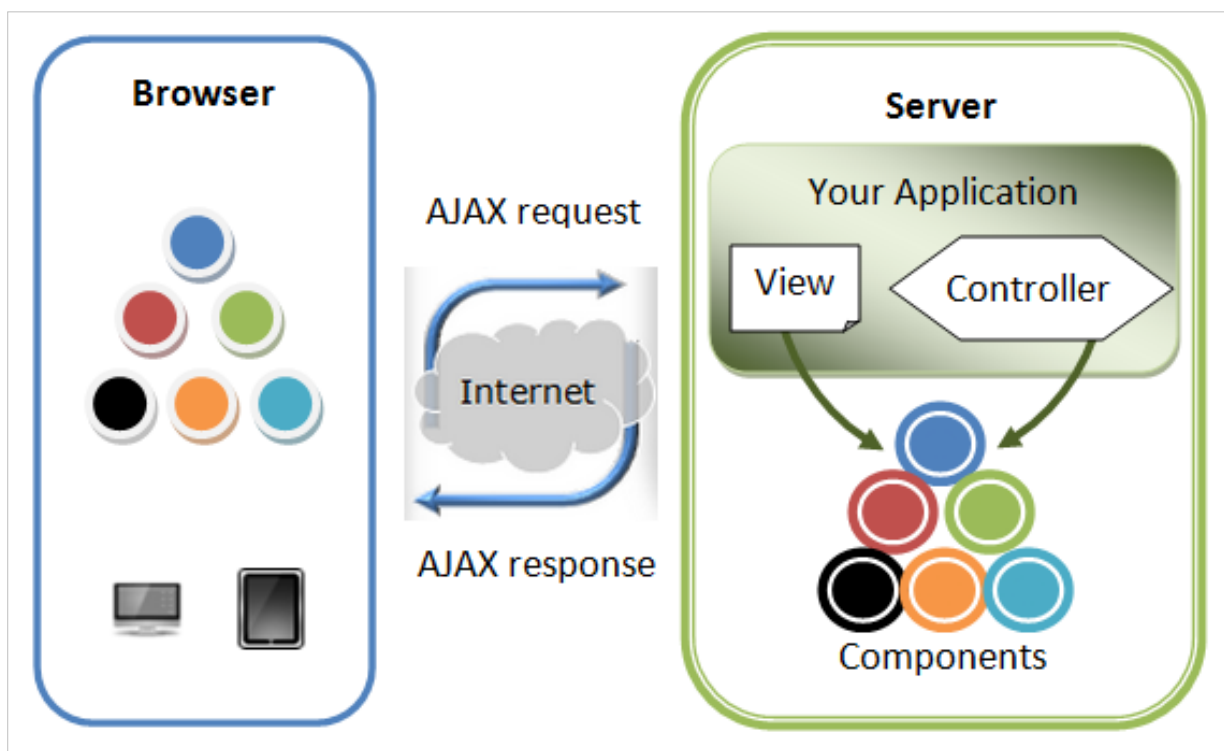
Chapter 1: Introduction

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^[1] 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 it for you. In addition to manipulating components directly i.e. MVC (Model-View-Controller) pattern^[2], ZK also supports another design pattern, MVVM (Model-View-ViewModel)^[3] which gives the Controller and View more separation. These two approaches are mutually interchangeable, and you can choose one of them upon your architectural consideration.

Architecture of ZK



Simple Architecture

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 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 client side.

About This Book

This book presents key concepts and suggested usage of ZK from the perspective of building a web application. Each chapter has a main topic, and we give one or more example applications to demonstrate each chapter's topic. Each chapter's applications are built upon previous chapter's application to 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 ZK Essentials/Chapter 2: Project Structure.

Chapter 1, we introduce the ZK itself including its strength, value, and architecture.

Chapter 2, we reveal the information of example application's source code and project structure.

Chapter 3, we introduce how to build a common layout which contains header, footer, and sidebar.

Chapter 4, we tell you how to control components programmatically.

Chapter 5, it describes how to collect, validate user input and response.

Chapter 6, we demonstrate how to implement common CRUD operations with a To-Do list application.

Chapter 7, we introduce 2 navigation ways in ZK, page-based and AJAX-based.

Chapter 8, it demonstrates a simple implementation to authenticate users.

Chapter 9, we describe how to integrate Spring framework into a ZK application.

Chapter 10, we demonstrate how to use JPA in a ZK application.

Example Application

The final result of this book is to build a small and rich application that has common features such as authentication, navigation, form input, and personal Todo list management. 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.

ek POWERED Application Name

Login with you name

Account: zkoss

Password:

Login

(use account='zkoss' and password='1234' to login)

ZK Essentials, you are using ZK 6.5.1
<http://www.zkoss.org>

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.

zk Application Name

POWERED

ZKOSS Logout

www.zkoss.org

ZK Demo

ZK Developer Reference

Profile (MVC)

Profile (MVVM)

Todo List (MVC)

Todo List (MVVM)

Profile (MVC)

Account: zkoss

Full Name: ZKOSS

Email: info@zkoss.org

Birthday:

Country:

Bio:

You are editing ZKOSS's profile.

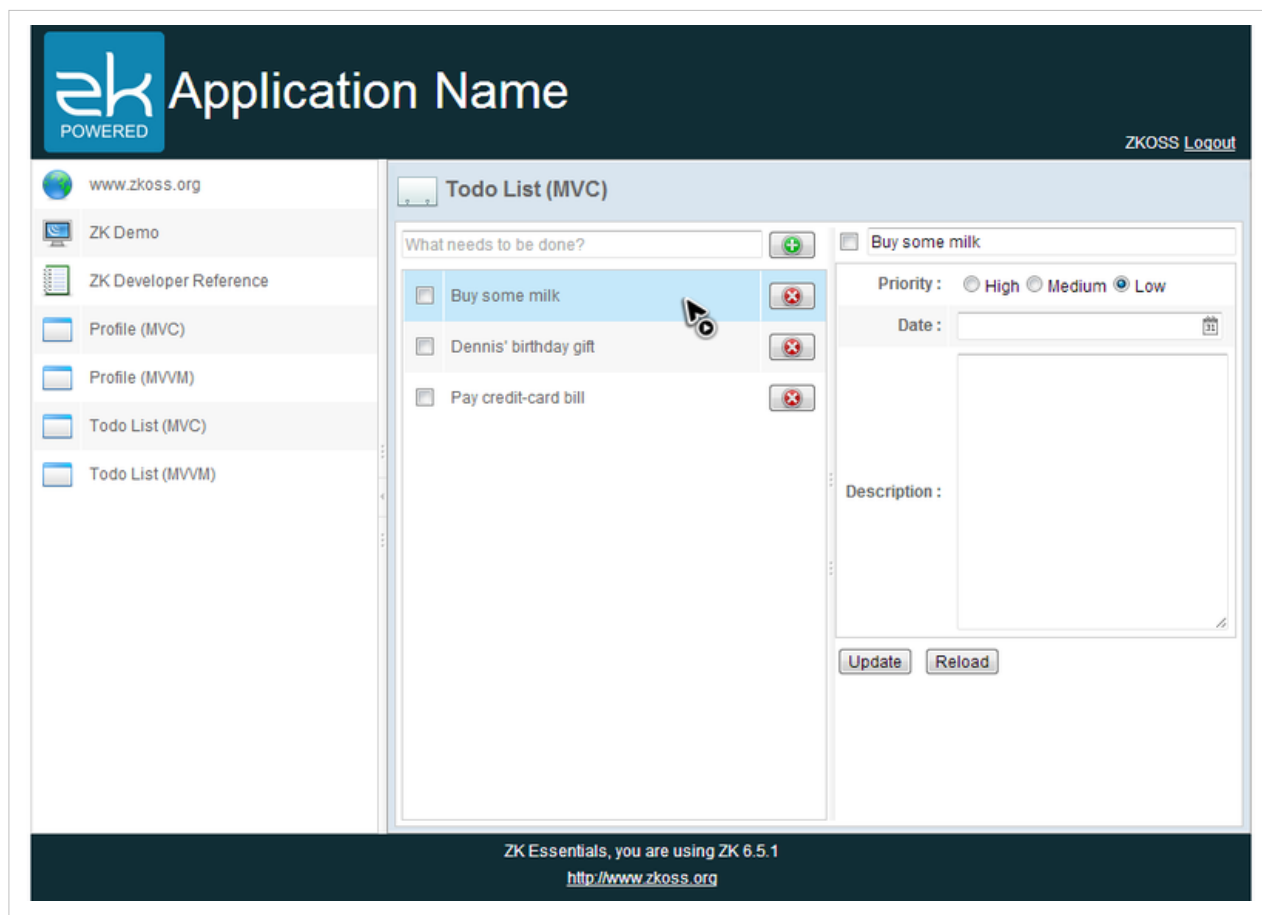
Save Reload

ZK Essentials, you are using ZK 6.5.1

<http://www.zkoss.org>

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

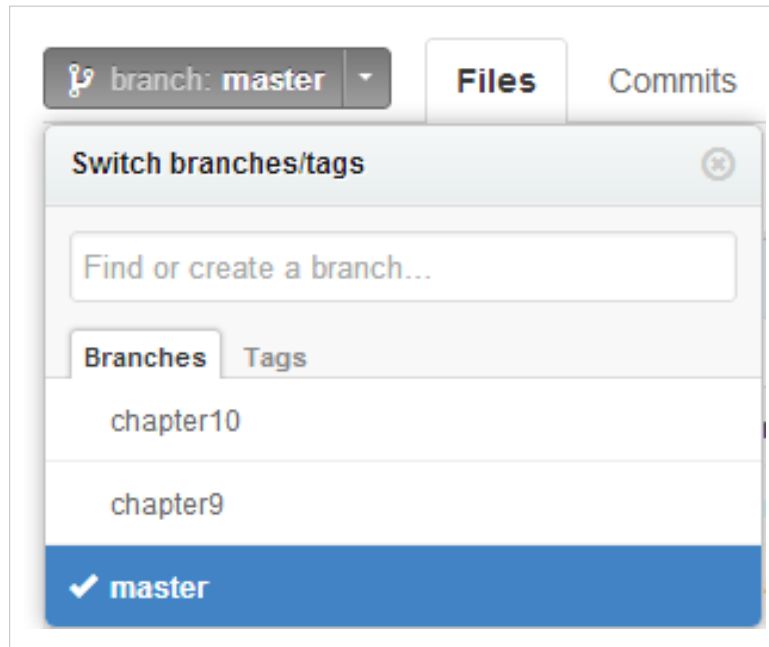
References

- [1] Browse components at ZK Demo (<http://www.zkoss.org/zkdemo/>)
- [2] ZK Developer's Reference MVC
- [3] ZK Developer's Reference MVVM

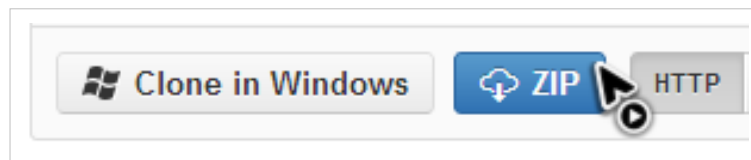
Chapter 2: Project Structure

Source Code

All source codes used in this book are available on github ^[1]. As our example application has 3 different configurations, our source code is divided into 3 branches: **master**, **chapter9**, and **chapter10**.



The **master** branch contains examples from chapter 3 to chapter 8. The **chapter9** branch has examples integrated with Spring and the **chapter10** branch contains examples which integrate with Spring and use JPA to persist data into a database.



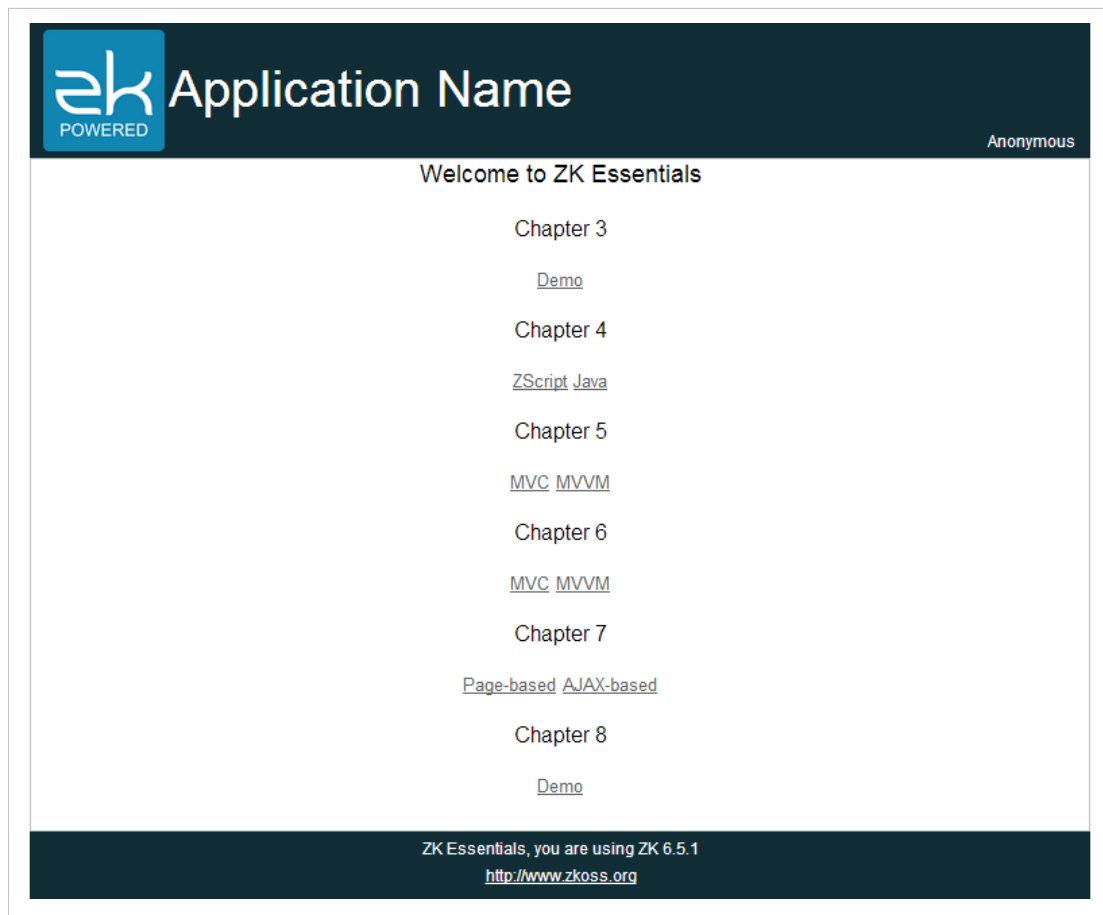
You can click the "ZIP" icon to download the current selected branch as a zip file.

Run Example Application

After you download the source code, you will find it is a Maven^[2] project with jetty plugin configured. Therefore, if you have Maven, you can run the example application with a simple command^[3] (The maven we use is **3.0.3**). Navigate to the root folder of your downloaded source code, say it's "zkessentials" and type the command:

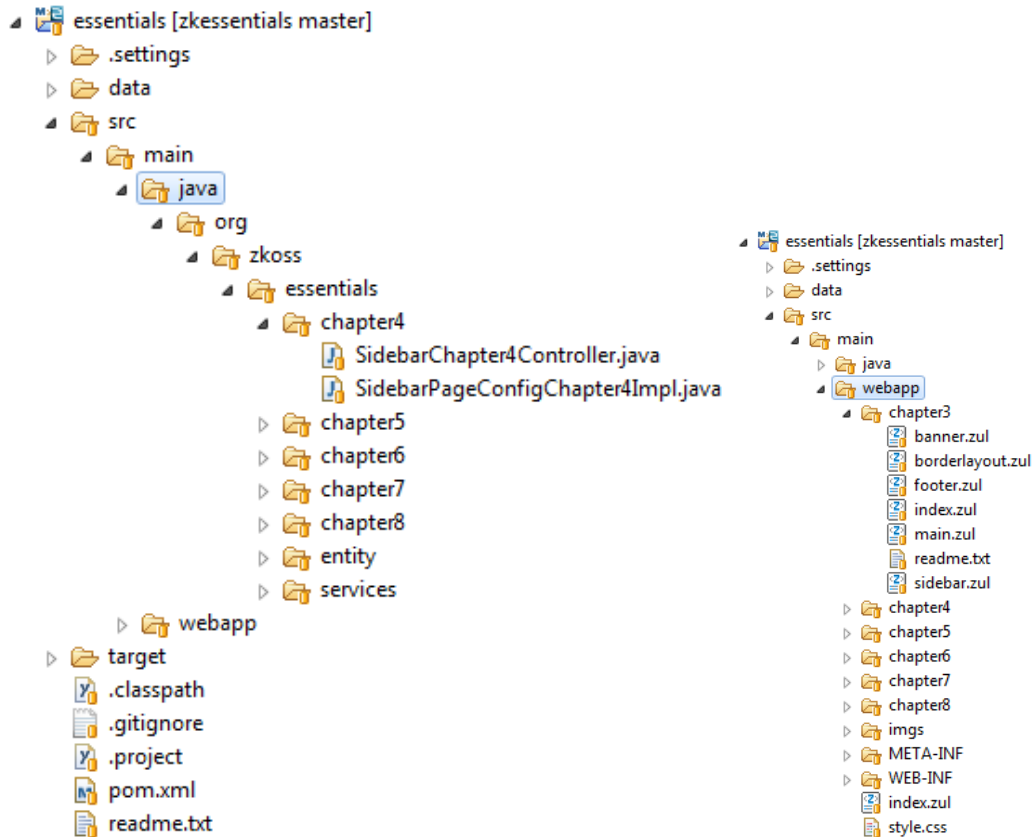
mvn jetty:run

Then visit the URL `http://localhost:8080/essentials/`, and you should see the screen below.



Project Structure

The 2 images below show the project structure of the example application. It's Maven default project structure, and all main source codes are under `src/main`. The left image shows the Java source code is under `src/main/java` and the right one shows the web application content is under `src/main/webapp`.



Project Structure: Java(left) and Webapp(right)

We name the source code packages according to each chapter and each package contains the classes used in the example of that chapter. Some common classes are separated to an independent package as they are used in multiple chapters. The classes under `org.zkoss.essentials.entity.*` are entity class. We also define some business interfaces under `org.zkoss.essentials.service.*` and different chapters have different implementations.

For ZUL pages, we put them in an independent folder 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.

References

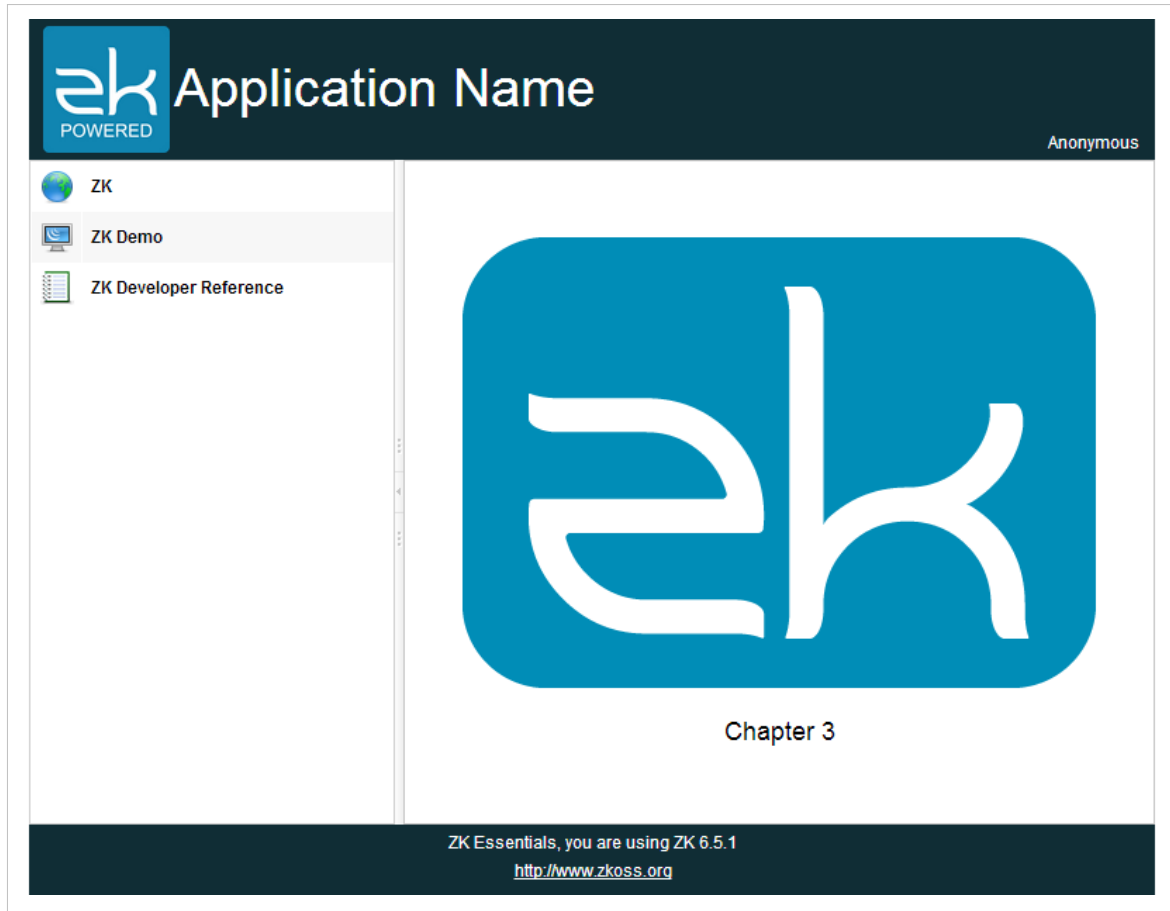
- [1] <https://github.com/zkoss/zkessentials>
- [2] Apache Maven (<http://maven.apache.org/>)
- [3] Start jetty in Maven (<http://docs.codehaus.org/display/JETTY/Maven+Jetty+Plugin>)

Chapter 3: User Interface and Layout

Design the Layout

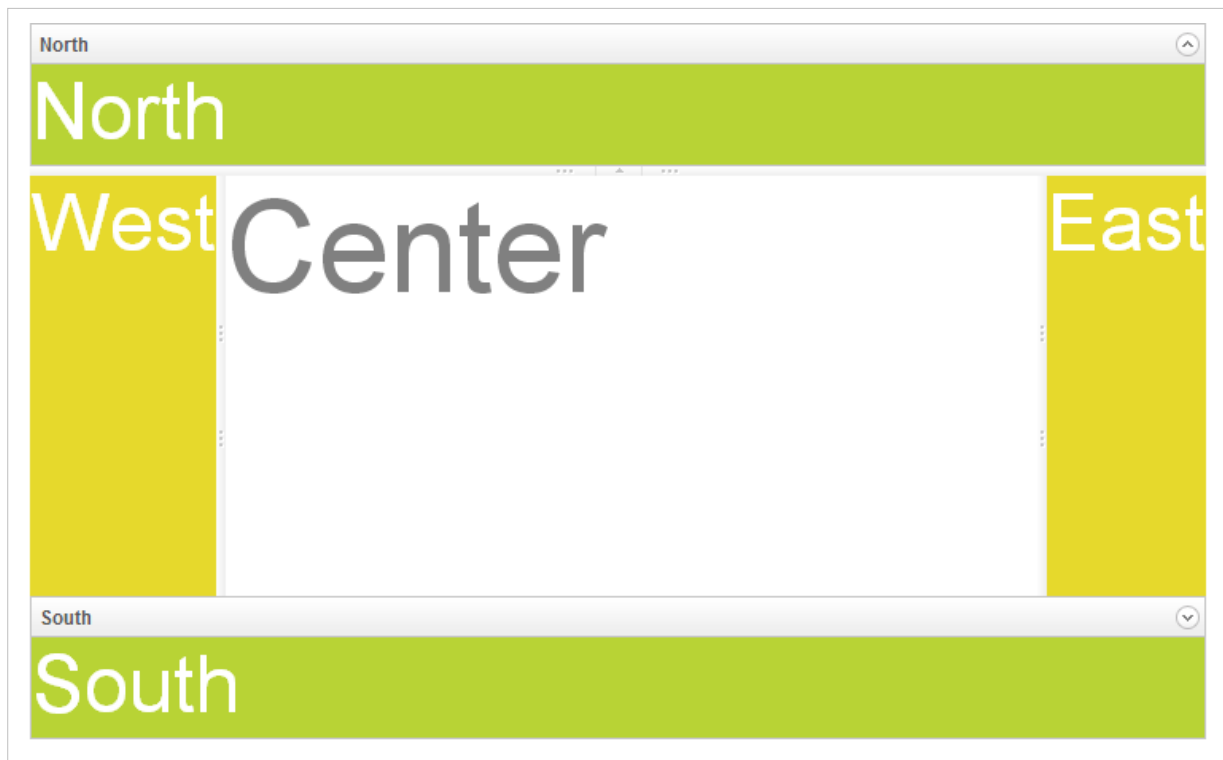
In the beginning, we will build the user interface which usually starts from designing the layout. ZK provides various components for different layout requirements, and you can even configure a component's attribute to adjust layout details.

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. 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^[1], 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*^[2] 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 function display.



Border Layout

Build the View

Building the View in ZK is basically creating components and there are two ways to do it: **Java** (programatic) and **XML-based** (declarative) approach. You can even mix these two approaches.

ZK allows you to compose a user interface in Java programmatically which is a feature called richlet, but we don't use this approach in this book.

ZK also provides a XML-formatted language called ZK User Interface Markup Language (ZUML). Each XML element instructs ZK Loader to create a component. Each XML attribute describes what value to be assigned to the created component. We will use this approach mainly in our example.

Write a ZUL

To create a component in ZK, we need to use a XML-based language named **ZUL** and all files written in ZUL should have the file extension ".zul". In zul files, one component can be represented as an XML element (tag) and you can configure each component's style, behavior, and function by setting the element's attributes.^[3] First, create a new text file with name *index.zul*, and type the following content:

Extracted from chapter3/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="100px">
      <label style="font-size:50px">West for sidebar</label>
    </west>
```

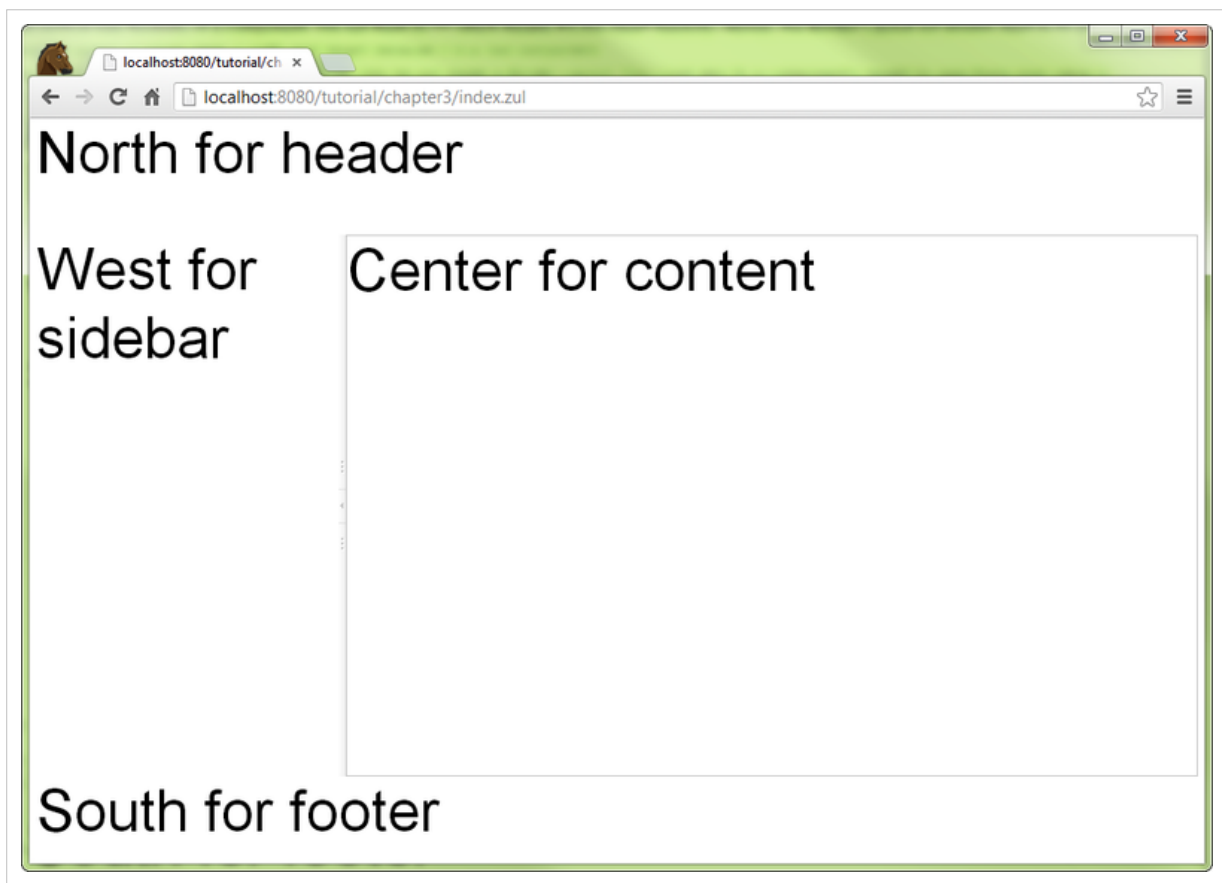
```

<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 *Border Layout* 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 *Border Layout*. You can also fix a component's height by specifying a pixel value to avoid its height changing due to browser sizes.
- Line 6: 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 9: Setting `autoscroll` to true will decorate the *Center* with a scroll bar when *Center* contains lots of information that exceed the its height.
- Line 4,7,10,13: These *Labels* 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.

chapter3/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:

chapter3/banner.zul

```
<div hflex="1" vflex="1" sclass="banner">
  <hbox hflex="1" vflex="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 `` 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.

chapter3/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 D
    </row>
    <row>
        <image src="/imgs/doc.png"/><a href="http://books.zkoss.org/wiki/ZK_De
    </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: Since we define two *Columns*, 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.

chapter3/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 Desktop^[4]'s Javadoc to find out available properties.

Next, we will combine these separated zul pages into *chapter3/index.zul*.

Include a Separate Page

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

For all areas, we use *Include* component to combine separated pages. This component can combine a separated zul for you when the parent zul is visited. This usage is presented below:

chapter3/index.zul

```

<?link rel="stylesheet" type="text/css" href="/style.css"?>
<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="100px">
            <include src="/chapter3/sidebar.zul"/>
        </west>
    </borderlayout>
</zk>

```

```

        <center id="mainContent" autoscroll="true">
            <include src="/chapter3/main.zul"/>
        </center>
        <south height="50px" border="none">
            <include src="/chapter3/footer.zul"/>
        </south>
    </borderlayout>
</zk>

```

- Line 1: This directive links a external style sheet under root folder.
- Line 5, 8, 11, 14: Specify a separated zul path at `src` attribute to include a page into current page.

After including 4 separated zul pages, we complete the example of this chapter. You can visit <http://localhost:8080/essentials/chapter3> to see the result. Since we set welcome file to "index.zul" in `web.xml`, <http://localhost:8080/essentials/chapter3/index.zul> will be visited by default.

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 attributes to apply CSS:

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

```
<label value="Chapter 3" style="font-weight: bold;"/>
```

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

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

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

1. `<style>` tag.

```

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

```

2. `<?link ?>` directive. It can link to a external style sheet which can apply to many pages. We use this way in the example to define CSS.

```

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

```


Source Code

- ZUL pages ^[5]

References

- [1] ZK Demo Layout (<http://www.zkoss.org/zkdemo/layout>). ZK releases multiple editions. Some layout components are only available in a specific edition, please refer to Feature & Edition (<http://www.zkoss.org/whyzk/features>)
- [2] Border Layout Demo (http://www.zkoss.org/zkdemo/layout/border_layout)
- [3] ZK Component Reference
- [4] <http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Desktop.html#>
- [5] <https://github.com/zkoss/zkessentials/tree/master/src/main/webapp/chapter3>

Chapter 4: Controlling Components

ZK's components are not only for constructing the user interface, we even can control them. In this chapter, we continue to use the last chapter's example but we will remove the 3 items with hyper links 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.

In Zscript

The simplest way to respond to a user's clicking is to write an event listener method and invoke it in the `onClick` attribute. We can define an event listener in Java inside a `<zscript>` element and those codes will be interpreted when the ZUL is visited. 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 evaluated. You can 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*.

Event listener redirect()

```
<grid hflex="1" vflex="1" sclass="sidebar">
  <zscript><![CDATA[
    //zscript code, it runs on server site, 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");

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

...

- Line 2: It's better to enclose your code with `<![CDATA[]]>`.
- Line 11: Define an event listener method like a normal Java method and it redirects a browser according to the passed variable.
- Line 14: The execution is an implicit variable which you can use it directly without declaration. It represents an execution of a client request that holds relevant information.

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*.

Invoke event listeners at "onClick"

```
<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 if you click a *Row* of the *Grid* in the sidebar, your browser will be redirected to a 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.

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.

The most commonly used architecture in web applications is *MVC (Model-View-Controller)* which separates an application into 3 parts. The Model is responsible for exposing data while performing business logic which is usually implemented by users, and the View is responsible for displaying data which is what ZUL does. The Controller can change the View's presentation and handle events from the View. The benefit of designing an application in MVC architecture is that your application is more modularized.

In ZK world, a Composer^[1] plays the same role as the Controller and you can assign it to a target component. Through the composer, you can listen events of the target component and manipulate target component's child components to change View's presentation according to your requirement. To create a Controller in ZK you simply create a class that inherits `SelectorComposer`^[2].

```
public class SidebarChapter4Controller extends SelectorComposer<Component>{
    //other codes...
}
```

Then "connect" 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 control of the controller.

chapter4/sidebar.zul

```
<grid hflex="1" vflex="1" sclass="sidebar"
      id="fnList"
      apply="org.zkoss.essentials.chapter4.SidebarChapter4Controller">
  <columns>
    <column width="36px"/>
    <column/>
  </columns>
  <rows/>
</grid>
```

- Line 2: A component id can be used to retrieve the component in a composer, please see the next section.
- Line 3: Apply a controller to a component.
- Line 8: Here we don't create 3 Rows in the zul because we need to add an event listener programmatically on each Row in the composer.

Wire Components

To control a component, we must retrieve it first. In `SelectorComposer`^[2], 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` will find the component whose id and type both equal to the variable name and type respectively.

```
public class SidebarChapter4Controller extends SelectorComposer<Component>{

    //wire components
    @Wire
    Grid fnList;

    ...
}
```

- Line 4,5 : `SelectorComposer` looks for a *Grid* whose id is "fnList" and assign it to the variable `fnList`.

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 Rows of the *Grid* for the sidebar, therefore we should override a composer life-cycle method `doAfterCompose(Component)`. The passed argument, `comp`, is the component that the composer applies to, which in our example is the *Grid*. This method will be called after all the child components under the component which has the composer applied to it are created, so we can change components' attributes or even create other components in it.

```
public class SidebarChapter4Controller extends SelectorComposer<Component>{

    //wire components
    @Wire
    Grid fnList;
```

```

//services
SidebarPageConfig pageConfig = new
SidebarPageConfigChapter4Impl();

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

    //initialize view after view construction.
    Rows rows = fnList.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 *Rows* with event listeners and put them into *Grid*. We will discuss them in next section.

Events & Event Listeners

A ZK event (Event^[3]) 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 clients.

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 *Rows* and add an event listener to each of them.

```

public class SidebarChapter4Controller extends SelectorComposer<Component>{

    //...

```

```

//wire components
@Wire
Grid fnList;

//services
SidebarPageConfig pageConfig = new
SidebarPageConfigChapter4Impl();

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

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

    for(SidebarPage page:pageConfig.getPages()){
        Row row =
constructSidebarRow(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");

    //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 21: Append a newly-created *Row* will make it become a child component of *Rows*.
- Line 28: The first step to create a component is instantiating its class.
- Line 32: Append a component to establish the parent-child relationship.
- Line 36: You can change a component's attributes by various setter methods and their method names correspond to tag's attribute name.
- Line 39: We create an `EventListener` anonymous class for convenience. Under a clustering environment, your event listener class should implement `SerializableEventListener` ^[4].
- Line 44: Implement the business logic in `onEvent()` method, and this method will be called if the listened event is sent to the server. Here we get current execution by `Executions` ^[5] and redirect a client to a new URL.
- Line 48: Apply the event listener to a *Row* for listening `Events.ON_CLICK` event which is triggered by a mouse clicking action.

In Line 28 ~ 36, those codes work equally to writing in 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 `http://localhost:8080/essentials/chapter4`".

Source Code

- ZUL pages ^[6]
- Java ^[7]

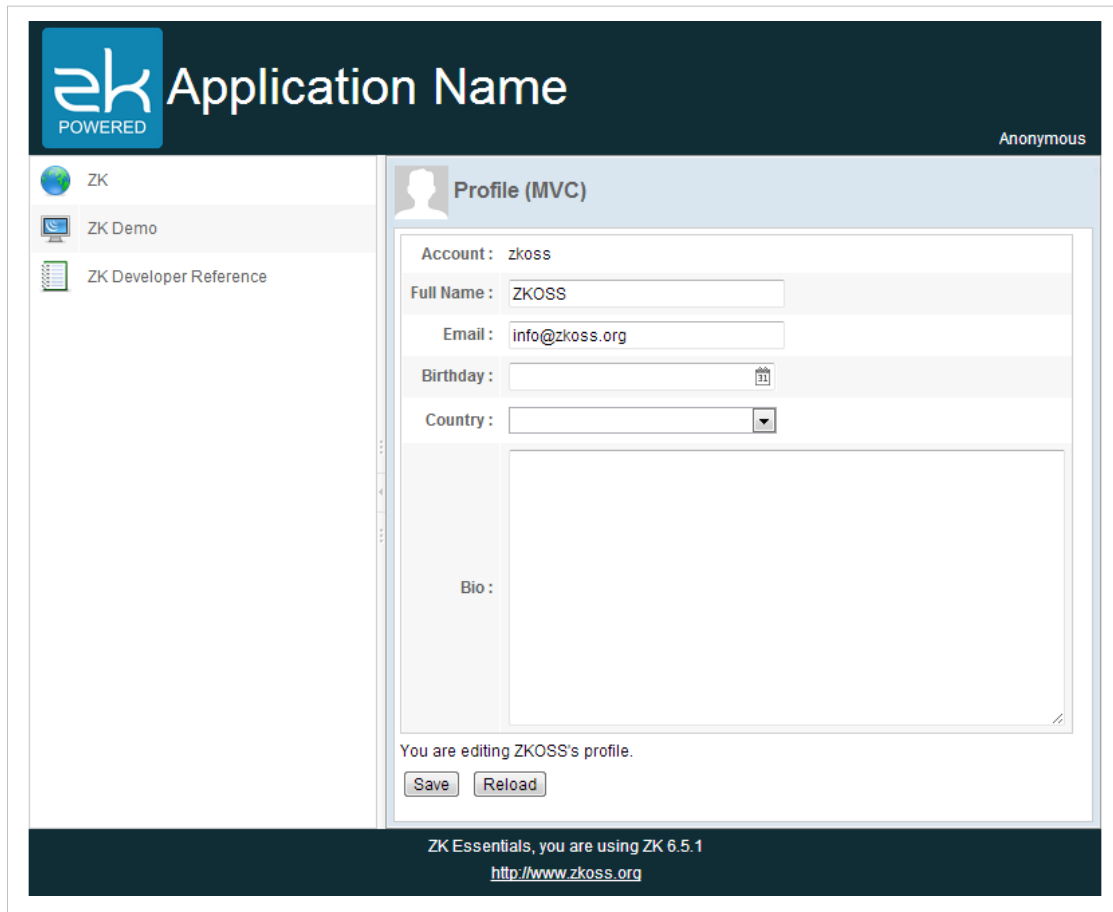
References

- [1] <http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Composer.html#>
- [2] <http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/SelectorComposer.html#>
- [3] <http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/event/Event.html#>
- [4] <http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/SerializableEventListener.html#>
- [5] <http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Executions.html#>
- [6] <https://github.com/zkoss/zkessentials/tree/master/src/main/webapp/chapter4>
- [7] <https://github.com/zkoss/zkessentials/tree/master/src/main/java/org/zkoss/essentials/chapter4>

Chapter 5: Handling User Input

Target Application

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



The screenshot displays a web application interface. At the top, a dark blue header contains the 'ek POWERED' logo on the left, the text 'Application Name' in the center, and 'Anonymous' on the right. Below the header, a light blue sidebar on the left lists three items: 'ZK' with a globe icon, 'ZK Demo' with a monitor icon, and 'ZK Developer Reference' with a document icon. The main content area is titled 'Profile (MVC)' and features a user profile form. The form includes the following fields: 'Account' (value: zkoss), 'Full Name' (value: ZKOSS), 'Email' (value: info@zkoss.org), 'Birthday' (with a calendar icon), and 'Country' (with a dropdown arrow). Below these is a large text area labeled 'Bio :'. At the bottom of the form, a message states 'You are editing ZKOSS's profile.' followed by 'Save' and 'Reload' buttons. A dark blue footer at the very bottom contains the text 'ZK Essentials, you are using ZK 6.5.1' and the URL 'http://www.zkoss.org'.

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 all 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 last chapter, it should be easy to construct a form style user interface as follows. It 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 the *Border Layout*.

chapter5/profile-mvc.zul

```
<?link rel="stylesheet" type="text/css" href="/style.css"?>
<window apply="org.zkoss.essentials.chapter5.mvc.ProfileViewController"
  border="normal" hflex="1" vflex="1" contentType="overflow:auto">
  <caption src="/imgs/profile.png" sclass="fn-caption" label="Profile (MVC)"/>
  <vlayout>
    <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"
              constraint="no empty: Please enter
your full name" width="200px"/>
          </cell>
        </row>
        <row>
          <cell sclass="row-title">Email :</cell>
          <cell>
            <textbox id="email"
              constraint="/.+@.+\. [a-z]+/: Please
enter an e-mail address" width="200px"/>
          </cell>
        </row>
        <row>
          <cell sclass="row-title">Birthday :</cell>
          <cell>
            <datebox id="birthday">
```



```

                                constraint="no future"
width="200px"/>
                                </cell>
                                </row>
                                <row>
                                <cell sclass="row-title">Country :</cell>
                                <cell>
                                <listbox id="country" mold="select" width="200px">
                                <template name="model">
                                <listitem label="{each}" />
                                </template>
                                </listbox>
                                </cell>
                                </row>
                                <row>
                                <cell sclass="row-title">Bio :</cell>
                                <cell><textbox id="bio" multiline="true" hflex="1" height="
                                </row>
                                </rows>
                                </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: *Caption* can be used to build compound header with an image for a *Window*.
- Line 5: *Vlayout* is a light-weight layout component which arranges child components vertically without splitter, align, and pack support.
- Line 13: *Cell* is used inside *Row*, *Hbox*, or *Vbox* for fully controlling style and layout.
- Line 20, 27, 34: Specify `constraint` attribute of an input component can activate input validation feature and we will discuss it in later section.
- Line 40: Some components have multiple molds, and each mold has its own style.
- Line 41: *Template* component can create its child components repeatedly upon the data model of parent component, *Listbox*, and doesn't has a corresponding visual widget itself. The `name` attribute is required and has to be "model" in default case.
- Line 42: The `{each}` is an implicit variable that you can use without declaration inside *Template*, and it represents an object of the data model list for each iteration when rendering. We use this variable at component attributes to reference the object's property with dot notation. In our example, we just set it at a *Listitem*'s label.
- Line 54: *Hlayout*, like *Vlayout*, but arranges child components horizontally.

User Input Validation

We can specify the `constraint` attribute of an input component with constraint rule to activate its input validation feature and the feature can work without writing any code in a controller. For example:

```
<textbox id="fullName" constraint="no empty: Plean en
```

- 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]+/: Plea
```

- 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.

The screenshot shows a web form titled "Profile (MVC)" with a user profile icon. The form contains the following fields and elements:

- Account:** anonymous
- Full Name:** Anonymous
- Email:** anonumous (This field has a red border and a red error message box next to it that says "Please enter an e-mail address".)
- Birthday:** A date picker showing the 31st.
- Country:** A dropdown menu.
- Bio:** A large text area.
- Buttons:** "Save" and "Reload" buttons at the bottom.

Initialize Profile Form

We want to create a drop-down list that contains a list of countries for selection. When a user visit 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.

Country List

This is made using a *Listbox* in "select" mold. The *Listbox*'s data is a list of country name strings provided by the controller. In ZK, all data components are designed to accept a separate data model that contains data to be rendered. You only have to provide such a data model and a data component will render the information as specified in the *Template*. This increases the data model's re-usability and decouples the data from a component's implementation.

For a *Listbox*, we can provide a `ListModelList`^[1] object.

Initialize data model for a *Listbox*

```
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()`.

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

```

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
AuthenticationServiceChapter5Impl();
    UserInfoService userInfoService = new
UserInfoServiceChapter5Impl();

    @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 cre = authService.getUserCredential();
    User user = userInfoService.findUser(cre.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 4: Wire ZK components as we talked in chapter 4.
- Line 17: Service classes that are used to perform business operations or get necessary data.
- Line 28: Load saved data to input components to initialize the View, so we should call it after initializing country list.
- Line 33: This method reloads the saved data from service classes to input components.
- Line 42~46: Push saved user data to components by `setValue()`.
- Line 48: Use `ListModelList.addToSelection()` to control the *Listbox*'s selection,

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` annotation instead of calling `addEventListener()` method (mentioned in chapter 4).

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, please refer to ZK Developer's Reference/MVC/Controller/Wire Event Listeners.

Listen "Save" button's 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 presentation in the event listener. In `doSaveProfile()`, we get user's input from input components and save the data to a `User` object. Then show the notification to the client.

Handle "Save" button's clicking

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

    @Listen("onClick=#saveProfile")
    public void doSaveProfile(){
        UserCredential cre = authService.getUserCredential();
        User user = userInfoService.findUser(cre.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 pre-defined to "Anonymous". We will write a real case 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.

To wire the event listener for "Reload" button's is similar as previous one, and it pushes saved user data to components using `setValue()`.

```

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;

    ...
    @Listen("onClick=#reloadProfile")
    public void doReloadProfile(){
        refreshProfileView();
    }

    ...
}

```

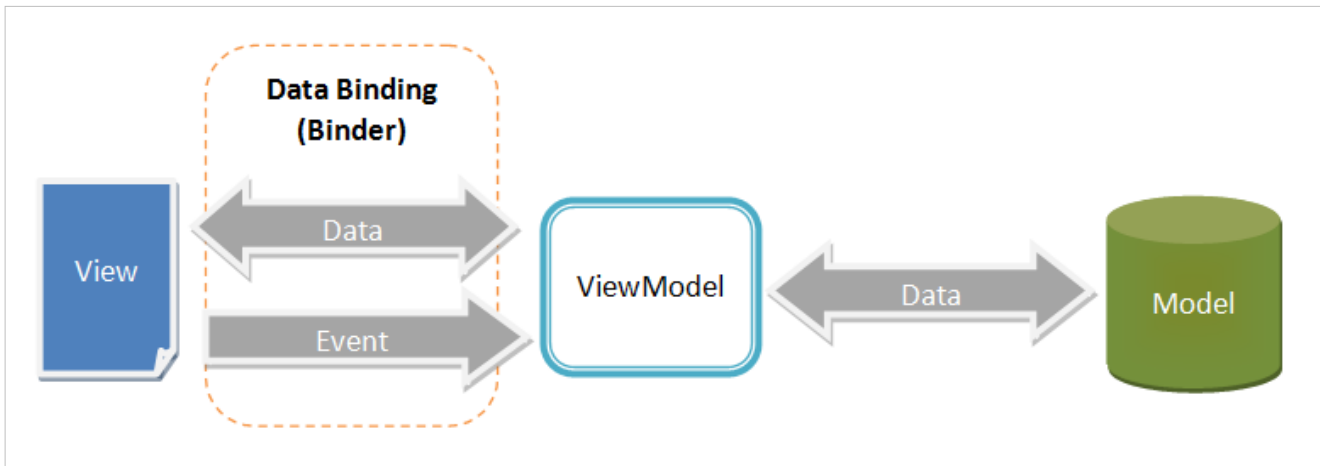
- Line 21: This method is listed in previous section.

After the above steps, we have finished all functions of the target application. Quite simple, right? You can see the result at <http://localhost:8080/essentials/chapter5/index.zul>.

MVVM Approach

In addition to the MVC approach, ZK also allows you to design your application using another architecture: *MVVM* (*Model-View-ViewModel*). This architecture also divides an application into 3 parts: View, Model, and ViewModel. The View and Model plays the same roles as they do in MVC. The ViewModel in MVVM acts like a *special Controller* for the View which is responsible for exposing data from the Model to the View and for providing required action and logic for user requests from the View. The ViewModel is a *View abstraction*, which contains a View's state and behavior. The biggest difference from the Controller in the MVC is that *ViewModel should not contain any reference to UI components* and knows nothing about the View's visual elements. Hence this clear separation between View and ViewModel decouples ViewModel from View and makes ViewModel more reusable and more abstract.

Since the ViewModel contains no reference to UI components, you cannot control components directly e.g. to get value from them or set value to them. Therefore we need a mechanism to synchronize data between the View and ViewModel. Additionally, this mechanism also has to bridge events from the View to the action provided by the ViewModel. This mechanism, the kernel operator of the MVVM design pattern, is a data binding system called "ZK Bind" provided by the ZK framework. 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.



This section we will demonstrate how to implement the same target application under MVVM approach.

Construct a View as MVC Approach

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

Extracted from chapter5/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 constraint="no empty: Plean enter your full
        </row>
        <row>
          <cell sclass="row-title">Email :</cell>
          <cell><textbox constraint="/.+@.+\. [a-z]+/: Please enter an
        </row>
        <row>
          <cell sclass="row-title">Birthday :</cell>
          <cell><datebox constraint="no future" width="200px"/></cell>
        </row>
        <row>
          <cell sclass="row-title">Country :</cell>
          <cell>
```

```

        <listbox mold="select" width="200px">
            <template name="model">
                <listitem />
            </template>
        </listbox>
    </cell>
</row>
<row>
    <cell sclass="row-title">Bio :</cell>
    <cell><textbox multiline="true" hflex="1" height="200px" />
</row>
</rows>
</grid>
<div>You are editing <label />'s profile.</div>
<hlayout>
    <button label="Save"/>
    <button label="Reload"/>
</hlayout>
</vlayout>
</window>

```

- Line 32: You might notice that there is no EL expression `${each}` as we will use data binding to access it.

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. ViewModel is also 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, the 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 and country list in the *Listbox*. The ViewModel should look like the following:

Define properties in a ViewModel

```

public class ProfileViewModel implements Serializable{

    //services
    AuthenticationService authService = new
AuthenticationServiceChapter5Impl();
    UserInfoService userInfoService = new
UserInfoServiceChapter5Impl();

    //data for the view
    User currentUser;

    public User getCurrentUser() {
        return currentUser;
    }
}

```

```

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

    @Init // @Init annotates a initial method
    public void init() {
        UserCredential cre = authService.getUserCredential();
        currentUser = userInfoService.findUser(cre.getAccount());
        if(currentUser==null) {
            //TODO handle un-authenticated access
            return;
        }
        ...
    }
}

```

- 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 define current user profile data 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.
- Line 18: There is a marker annotation `@Init` for a method which should be at most one in each ViewModel and ZK will invoke this method after instantiating a ViewModel class. We should perform initialization in it, e.g. get user credential to initialize `currentUser`.

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

```

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

```

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`.

Apply a ViewModel on a Component

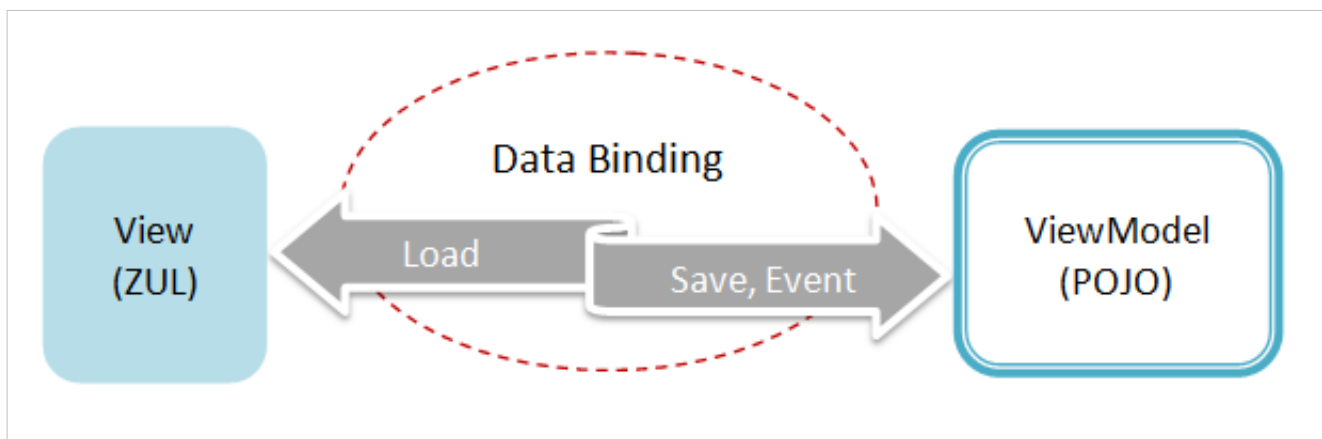
Before data binding can work, we must apply a composer called **org.zkoss.bind.BindComposer**. It will create a **binder** for the ViewModel and instantiate the ViewModel's class. Then we should bind a ZK component to our ViewModel by setting its **viewModel** attribute with the ViewModel's id in `@id` and the ViewModel's full-qualified class name in `@init`. The id is used to reference the ViewModel's properties, e.g. `vm.name`, whilst the full-qualified class name is used to instantiate the ViewModel object itself. So that component becomes the *Root View Component* for the ViewModel. All child components of this Root View Component can be bound to the same ViewModel and its properties, so we usually bind the root component of a page to a ViewModel.

```
<window apply="org.zkoss.bind.BindComposer"
        viewModel="@id('vm')
@init('org.zkoss.essentials.chapter5.mvvm.ProfileViewModel') "
        border="normal" hflex="1" vflex="1" contentStyle="overflow:auto">
...
</window>
```

- Line 1: Under MVVM approach, the composer we apply is fixed **org.zkoss.bind.BindComposer**.
- Line 2: Specify ViewModel's id with `@id` and the its full-qualified class name in `@init` for the binder.

Data Binding to 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 and save) data between components and the ViewModel for us automatically.



Let's demonstrate how to make *ListBox* load a list of country name from the ViewModel. We have talked about the data model concept in MVC approach section, and we also need to prepare a model object that are defined in one of our ViewModel's properties, `countryList`. You might find `getCountryList()` return a `List` instead of a `ListModelList`, but don't worry. ZK will convert it automatically. We use `@load` to load a ViewModel's property to a component's attribute and `@save` to save an attribute value into a ViewModel's property (usually for an input component). If both loading and saving are required, we could use `@bind`.

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

```

        </template>
    </listbox>
</cell>
...

```

- Line 3: We setup a load binding with `@load`. The `vm` is the ViewModel's id we specified at `@id` in previous section and the target property (`countryList`) can be referenced in dot notation.
- Line 4: *Template* component, we have explained in MVC approach section, can create its child components repeatedly upon the data model of parent component.
- Line 5: The implicit variable `each` which you can use without declaration inside *Template* represents each object in the data model for each iterative rendering (It represents String object of a country name in this example). We use this variable to access objects of data model. In our example, we just make it as a *Listitem*'s label.

In MVC approach, we have to call an input component's getter method (e.g. `getValue()`) to collect user input. But in MVVM approach, ZK will 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="@bind(vm.currentUser.fullName)" constraint="no empty: Please e

```

For the property `currentUser`, we want to both save user input back to the ViewModel and load value from the ViewModel, so we should use the `@bind` at `value` attribute. Notice that you can bind `selectedItem` to a property, then the user's selection can be saved automatically to the ViewModel.

```

...
<rows>
    <row>
        <cell sclass="row-title">Account :</cell>
        <cell>
            <label value="@load(vm.currentUser.account)"/>
        </cell>
    </row>
    <row>
        <cell sclass="row-title">Full Name :</cell>
        <cell>
            <textbox value="@bind(vm.currentUser.fullName)" constraint="
        </cell>
    </row>
    <row>
        <cell sclass="row-title">Email :</cell>
        <cell>
            <textbox value="@bind(vm.currentUser.email)" constraint="/.+@.+\./
        </cell>
    </row>
    <row>
        <cell sclass="row-title">Birthday :</cell>
        <cell>
            <datebox value="@bind(vm.currentUser.birthday)" constraint="no fu
        </cell>
    </row>

```

```

        <row>
            <cell sclass="row-title">Country :</cell>
            <cell>
                <listbox model="@load(vm.countryList)" selectedItem="@bind(vm.currentUser.country)">
                    <template name="model">
                        <listitem label="@load(each)" />
                    </template>
                </listbox>
            </cell>
        </row>
        <row>
            <cell sclass="row-title">Bio :</cell>
            <cell>
                <textbox value="@bind(vm.currentUser.bio)" multiline="true" hflex="1" />
            </cell>
        </row>
    </rows>

```

- Line 12,18,24,40: Use `@bind` to save user input back to the ViewModel and load value from the ViewModel.
- Line 30: Bind `selectedItem` to `vm.currentUser.country` and the selected country will be saved 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:

```

...
    <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 Away Unsaved Input

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 for example, 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 maybe it would mislead users, making them think they have changed their profile, so we don't want this.

The screenshot shows a web form titled "Profile (MVVM)". It contains several input fields: "Account" with the value "anonymous", "Full Name" with "Anonymous Somebody", "Email" with "anonumous@your.com", "Birthday" with a calendar icon, and "Country" with a dropdown arrow. Below these is a "Bio" label and a large text area. At the bottom of the form, there is a status message: "You are editing Anonymous Somebody's profile." with "Save" and "Reload" buttons. A mouse cursor is pointing at the status message.

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 chapter5/profile-mvvm.zul


```

...
<grid width="500px" form="@id('fx') @load(vm.currentUser) @save(vm.currentUser, bef
...
<rows>
  <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=
        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" h
  </row>
</rows>

```

```

</grid>
<div>You are editing <label value="@load(vm.currentUser.fullName)"/>'s profile.</div>
...

```

- Line 2: 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 7,12, 19, 26,33, 43: We should bind attributes to middle object's properties to avoid altering `ViewModel`'s properties.
- Line 47: 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`).

The screenshot shows a web application titled "Profile (MVVM)". It features a form with the following fields:

- Account:** anonymous
- Full Name:** Mose
- Email:** anonumous@your.com
- Birthday:** 31
- Country:** (dropdown menu)
- Bio:** (large text area)

At the bottom of the form, it states "You are editing Anonymous's profile." and includes "Save" and "Reload" buttons. A mouse cursor is pointing at the "Save" button.

Unsaved Input Doesn't Change Data

After completing above steps, visit <http://localhost:8080/essentials/chapter5/index-mvvm.zul> to see the result.

Source Code

- ZUL pages ^[2]
- Java ^[3]

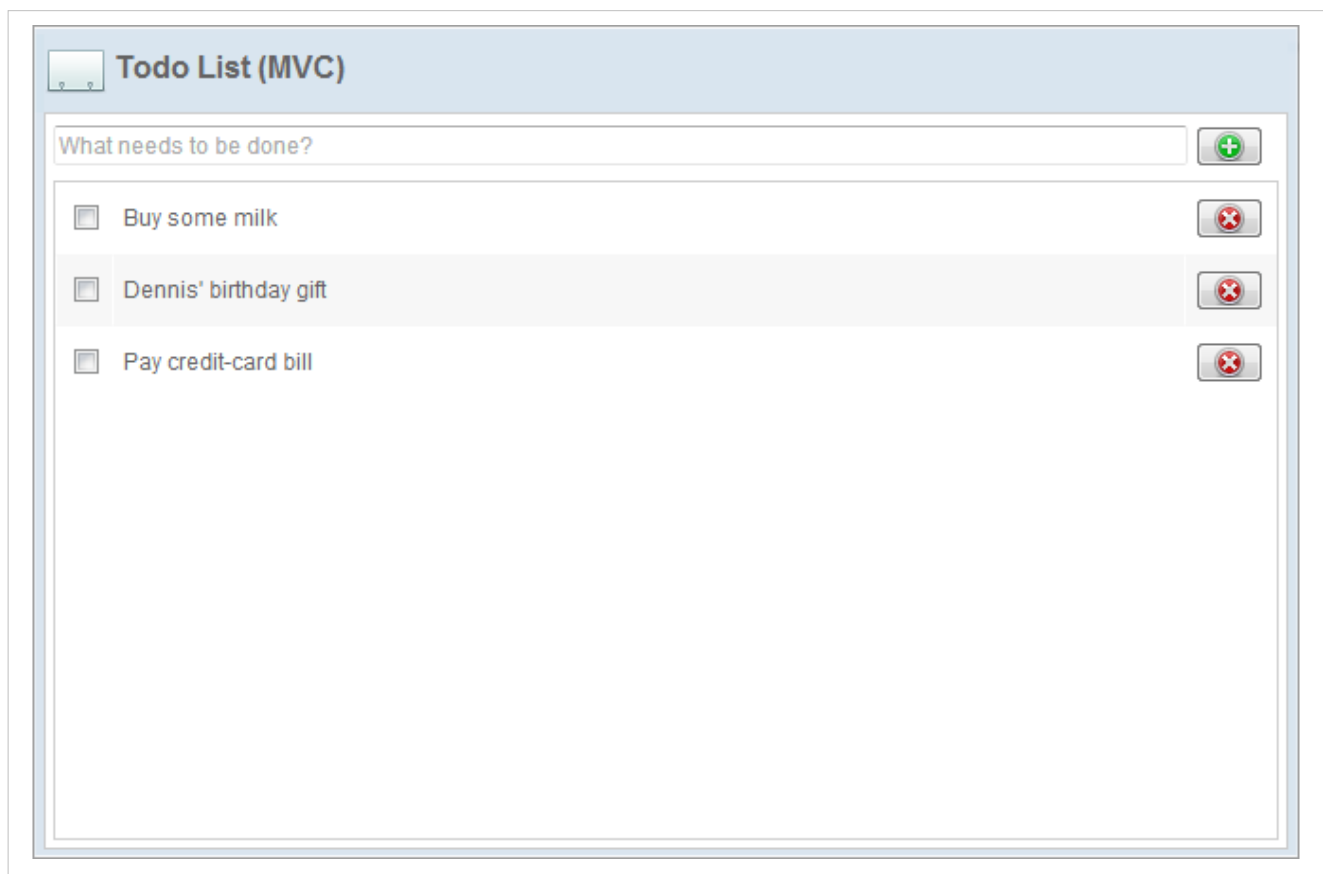
References

- [1] <http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zul/ListModelList.html#>
[2] <https://github.com/zkoss/zkessentials/tree/master/src/main/webapp/chapter5>
[3] <https://github.com/zkoss/zkessentials/tree/master/src/main/java/org/zkoss/essentials/chapter5>

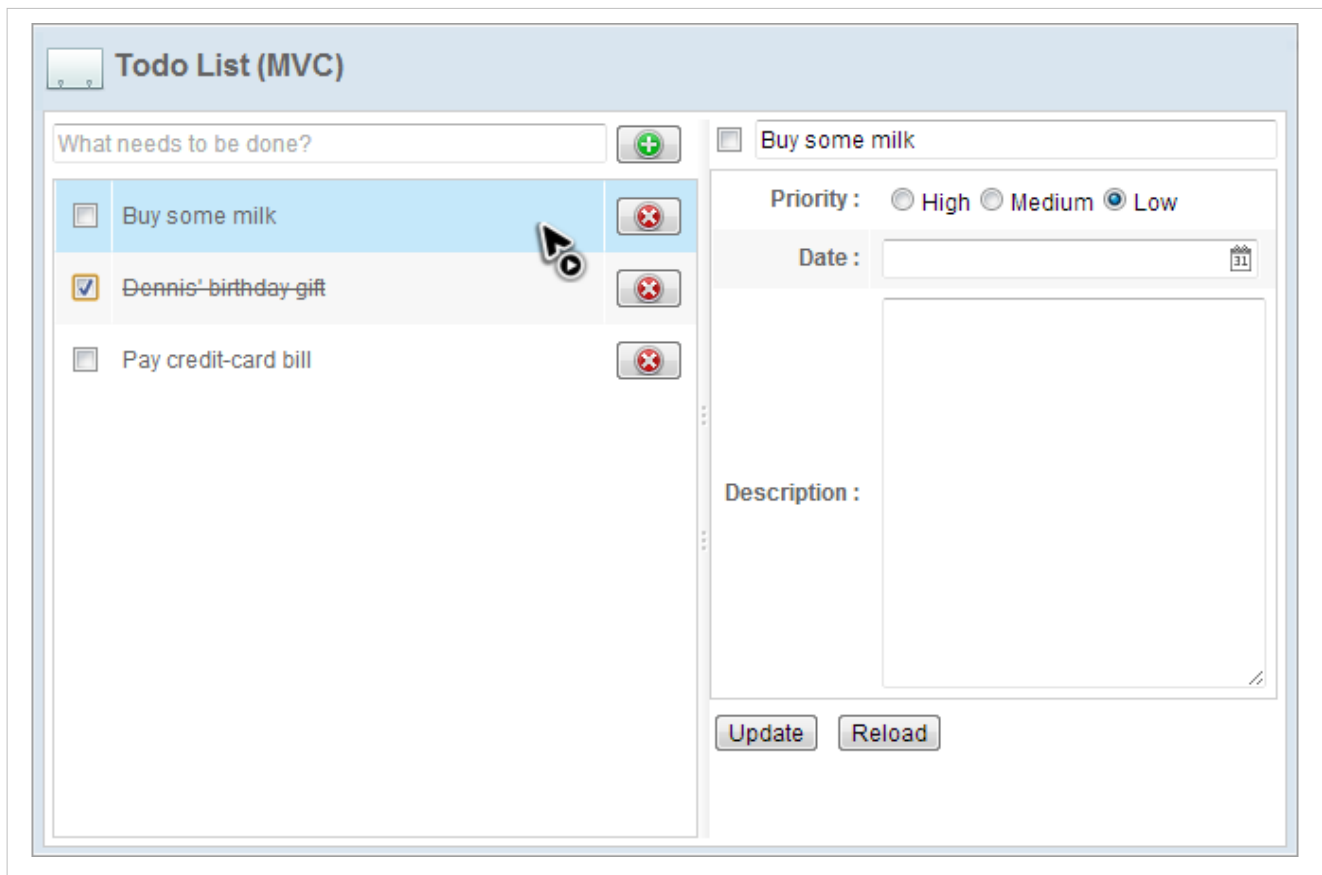
Chapter 6: Implementing CRUD

Target Application

In this chapter, we are going to build a todo list management application with 4 basic operations, Create, Read, Update, and Delete (CRUD). The application's user interface looks like the images below:



Select an Item:



Select a Todo Item

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 chapter6/todolist-mvc.zul

```
<?link rel="stylesheet" type="text/css" href="/style.css"?>
<window apply="org.zkoss.essentials.chapter6.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="true"
      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: 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="/
    </listcell>
</listitem>
</template>
</listbox>
...

```

- Line 8: The default value for the required attribute name is "model".
- Line 9: 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 11: 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 11, 17: 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
    TodoListServiceChapter6Impl();

    //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);

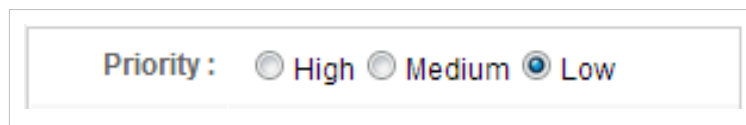
        ...

    }

    ...
}

```

- 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

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
        TodoListServiceChapter6Impl();

        //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.

```

public class TodoListController extends SelectorComposer<Component>{

    //wire components
    @Wire
    Textbox todoSubject;

    //services
    TodoListService todoListService = new
    TodoListServiceChapter6Impl();

    //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.

```

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 on `Select` 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

```

//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 chapter6/todolist-mvc.zul

```

...
<listbox id="todoListbox" vflex="1">
...
    <template name="model">
        <listitem sclass="{each.complete?'complete-todo':''}" value="{each}">
            <listcell>
                <checkbox forward="onCheck=todoListbox.onTodoCheck" checked=
            </listcell>
            <listcell>
                <label value="{each.subject}"/>
            </listcell>
            <listcell>
                <button forward="onClick=todoListbox.onTodoDelete" image="/
            </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 `ForwardEvent`^[1]. 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=
            </listcell>
            <listcell>
                <label value="{each.subject}"/>
            </listcell>
            <listcell>
                <button forward="onClick=todoListbox.onTodoDelete" image="/
            </listcell>
        </listitem>
    </template>
...

```

- Line 12: 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, visit <http://localhost:8080/essentials/chapter6/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
    TodoListServiceChapter6Impl();

    //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(vm.toggleComplete,each)" />
            </listcell>
            <listcell>
                <label value="@bind(each.subject)" />
            </listcell>
            <listcell>
                <button onClick="@command('deleteTodo',todo=each)" image="@bind(each.image)" />
            </listcell>
        </listitem>
    </template>
</listbox>

```

- Line 1: 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 8: 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 10,13: 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')
    <button onClick="@command('addTodo')" image="/imgs/plus.png"
</hbox>
```

- Line 2~3: 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=
            <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" />
        </cell>
      </row>
    </rows>
  </grid>
</vlayout>
```

```

        </row>
    </rows>
</grid>
<hlayout>
    <button onClick="@command('updateTodo') " label="Update"/>
    <button onClick="@command('reloadTodo') " label="Reload"/>
</hlayout>
</vlayout>

```

- Line 2: 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 4,5,16,25,30: Binding each input field to each property of the middle object through `fx`.
- Line 16: 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 resetting 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

```
//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 `AbstractValidator`^[2] 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.

```
<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.

```
@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
        //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('compl
        </listcell>
        ...
    </listitem>
</template>
```

- Line 4: 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)" />
        </listcell>
        <listcell>
            <label value="@bind(each.subject)" />
        </listcell>
        <listcell>
            <button onClick="@command('deleteTodo',to
        </listcell>
    </listitem>
</template>

```

- Line 10: 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/essentials/chapter6/todolist-mvvm.zul> to see the result.

Source Code

- ZUL pages ^[3]
- Java ^[4]

References

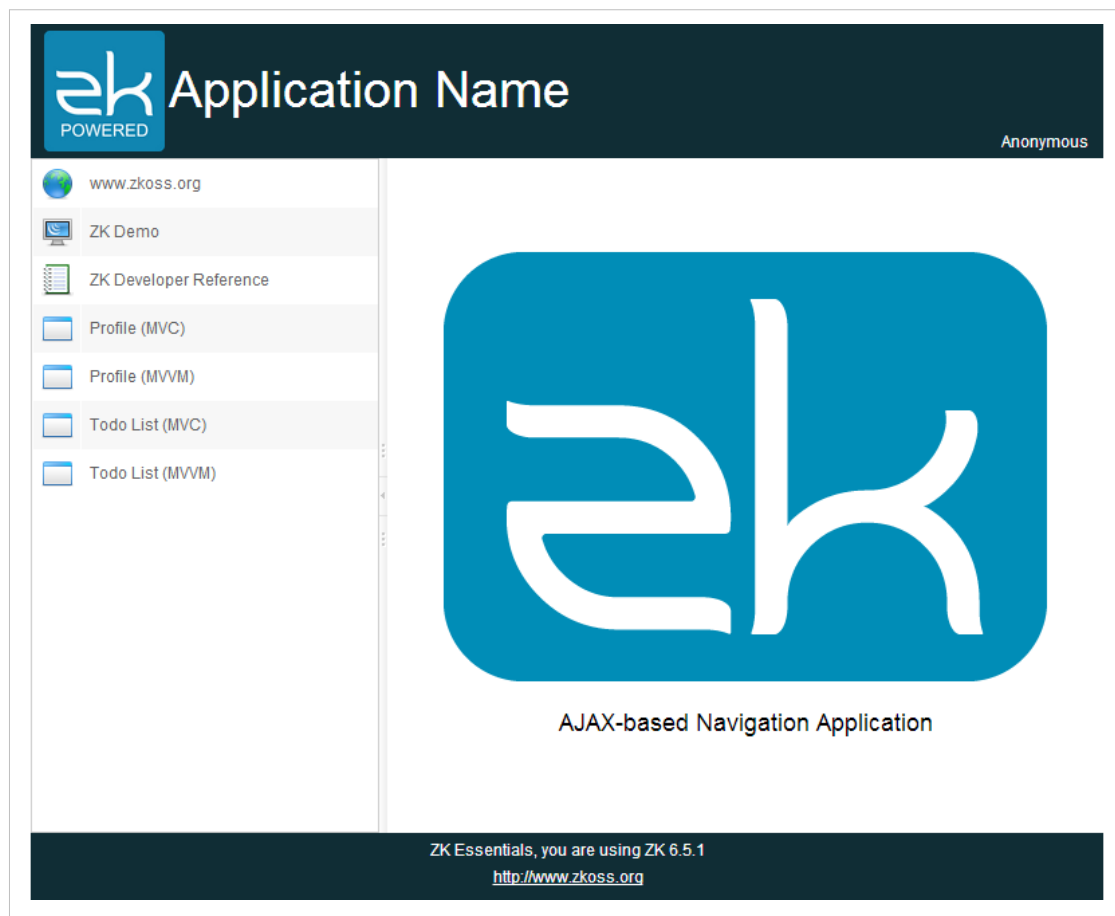
- [1] <http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/event/ForwardEvent.html#>
- [2] <http://www.zkoss.org/javadoc/latest/zk/org/zkoss/bind/validator/AbstractValidator.html#>
- [3] <https://github.com/zkoss/zkessentials/tree/master/src/main/webapp/chapter6>
- [4] <https://github.com/zkoss/zkessentials/tree/master/src/main/java/org/zkoss/essentials/chapter6>

Chapter 7: Navigation and Templating

Overview

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 where 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 ZK provides **Templating** 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. The lower 4 menu items lead you to different functions and they only change the central area's content. All other areas are unchanged to maintain a consistent layout style among

functions.

Templating

In our example application, we want to keep the header, the sidebar, and the footer 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 have a consistent style. One way is to copy the duplicated part from one zul to another, but it is hard to maintain. Fortunately, ZK provides a Templating technique that lets you define a template zul and apply it to multiple zul pages afterwards. All zul pages that apply the same template zul have the same layout, so changing the template zul can change the layout of all pages once.

The steps to use templating are:

1. Create a template zul and define anchors
2. Apply the template in the target zul and define zul fragments for anchors

Then when you visit the target zul page, ZK will insert those fragments to corresponding anchors upon anchor names and combine them as one page.

Create a Template ZUL File

Creating a template is nothing different from creating a normal zul, but you should define one or more anchors by specifying annotation `@insert()` at `self`. You can give any name to identify an anchor that will be used to insert a zul fragment with the same anchor name later.

chapter7/pagebased/layout/template.zul

```
<zkg>
    <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="100px">
            <include src="/chapter7/pagebased/layout/sidebar.zul"/>
        </west>
        <center id="mainContent" autoscroll="true" border="none" self="@insert(content)">
        </center>
        <south height="50px" border="none">
            <include src="/chapter3/footer.zul"/>
        </south>
    </borderlayout>
</zkg>
```

- Line 9: Define an anchor with name `content`

Apply the Template

After creating a template zul, we can apply it in another zul with directive

```
<?init class="org.zkoss.zk.ui.util.Composition" arg0="template_path"?>
```

This directive tells ZK that this page uses the specified template. Then we also have to define zul fragments that are inserted to the anchor in the template zul with annotation `@define(anchorName)` at `self` attribute. The `anchorName` you specify should correspond to one of anchors defined in the template zul.

chapter7/pagebased/index.zul

```
<?link rel="stylesheet" type="text/css" href="/style.css"?>
<?init class="org.zkoss.zk.ui.util.Composition" arg0="/chapter7/pagebased/layout/template
<zk>
    <include self="@define(content)" src="/chapter7/pagebased/home.zul"/>
</zk>
```

- Line 2: Tell ZK that we want to use a template zul for current page and give the path of template zul.
- Line 4: The anchor name `content` correspond to the anchor name defined in the template zul in previous section.

After above steps, when you visit <http://localhost:8080/essentials/chapter7/pagebased/index.zul>, ZK will render the page based on `template.zul` and attach the *Include* component to `<center>`.

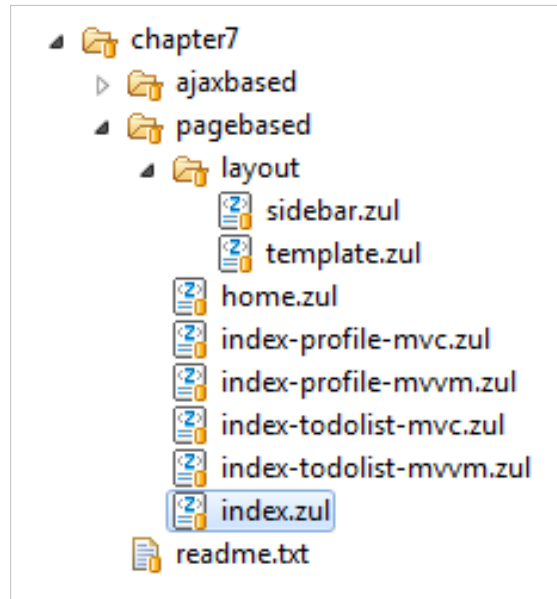
Page-based Navigation

Traditional web applications are usually designed with page-based navigation. 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 pages for those items in the sidebar.



From above image, you can see there are four zul pages which correspond to items in the sidebar under "chapter7/pagebased" (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.

/chapter7/pagebased/index-profile-mvc.zul

```
<?link rel="stylesheet" type="text/css" href="/style.css"?>
<?init class="org.zkoss.zk.ui.util.Composition" arg0="/chapter7/pagebased/layout/template
<zk>
    <include self="@define(content)" src="/chapter5/profile-mvc.zul"/>
</zk>
```

- Line 2: Apply the template in /chapter7/pagebased/layout/template.zul.
- Line 4: Define a zul fragment for the anchor content.

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"
        , "/chapter7/pagebased/index-profile-mvc.zul"));
        pageMap.put("fn2", new SidebarPage("fn2", "Profile
(MVVM) ", "/imgs/fn.png"
        , "/chapter7/pagebased/index-profile-mvvm.zul"));
        pageMap.put("fn3", new SidebarPage("fn3", "Todo List
(MVC) ", "/imgs/fn.png"
        , "/chapter7/pagebased/index-todolist-mvc.zul"));
        pageMap.put("fn4", new SidebarPage("fn4", "Todo List
(MVVM) ", "/imgs/fn.png"
        , "/chapter7/pagebased/index-todolist-mvvm.zul"));
    }

    ...

}

```

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

The following codes show how to redirect a user to an independent page when they 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 = fnList.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;
    }
}

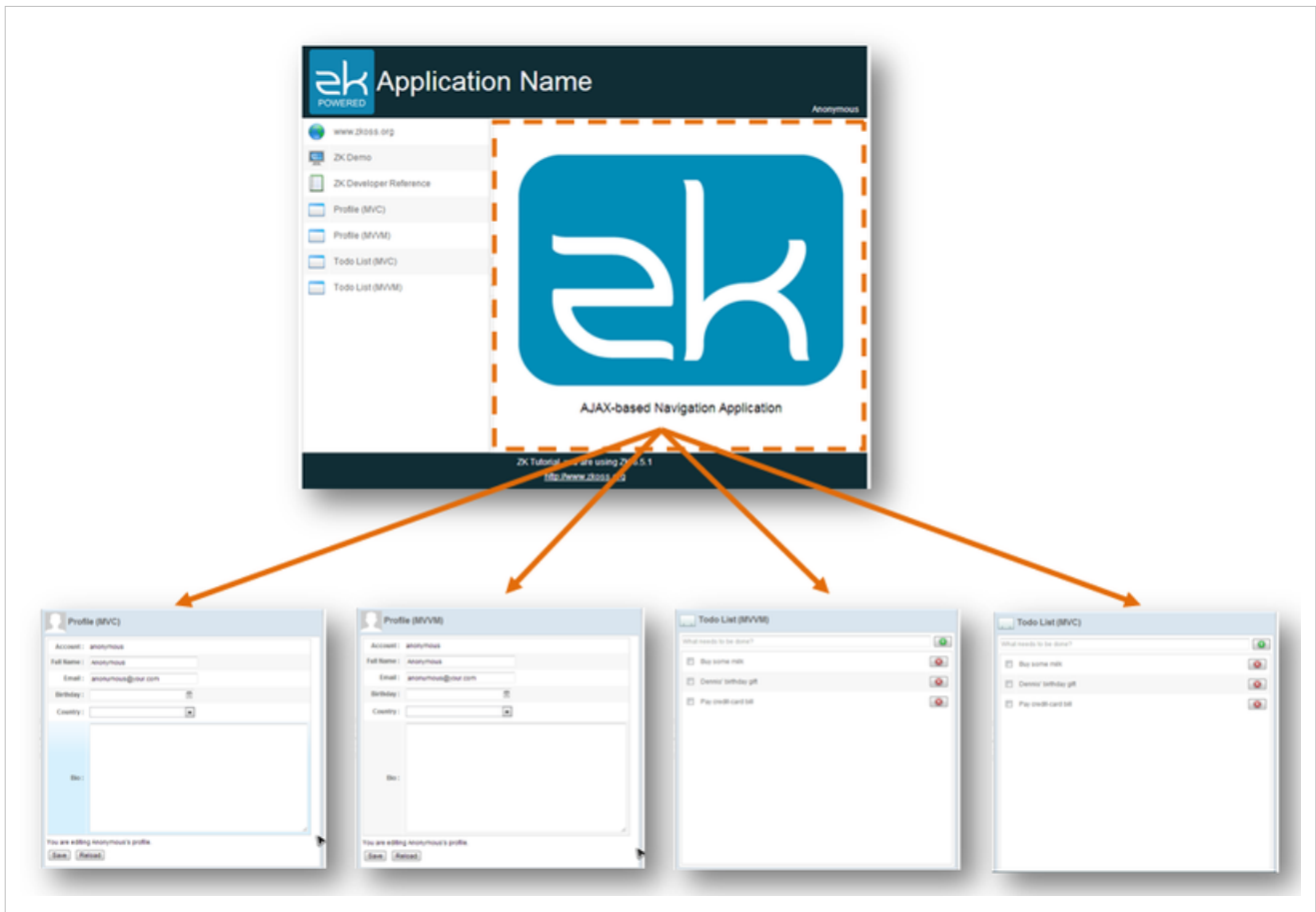
```

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

Visit <http://localhost:8080/essentials/chapter7/pagebased/index.zul>. You will see the URL changes and whole page reloads each time you click a different menu item.

AJAX-based Navigation

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 changing `src` attribute of *Include* component. It can change only partial content of an 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 use the same layout example to demonstrate AJAX-based navigation.

Below is the index page, its content is nearly the same as the index page of page based example except it includes a different zul.

chapter7/ajaxbased/index.zul

```
<?link rel="stylesheet" type="text/css" href="/style.css"?>
<?init class="org.zkoss.zk.ui.util.Composition" arg0="/chapter7/ajaxbased/layout/template
<zk>
    <include id="mainInclude" self="@define(content)" src="/chapter7/ajaxbased/home.zul
</zk>
```

- Line 4: 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.zkoss.org/wiki/ZK_Developer's_Reference"));

        pageMap.put("fn1", new SidebarPage("fn1", "Profile
(MVC)", "/imgs/fn.png", "/chapter5/profile-mvc.zul"));
        pageMap.put("fn2", new SidebarPage("fn2", "Profile
(MVVM)", "/imgs/fn.png", "/chapter5/profile-mvvm.zul"));
        pageMap.put("fn3", new SidebarPage("fn3", "Todo List
(MVC)", "/imgs/fn.png", "/chapter6/todolist-mvc.zul"));
        pageMap.put("fn4", new SidebarPage("fn4", "Todo List
(MVVM)", "/imgs/fn.png", "/chapter6/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* and change its `src` according to the menu item's URL.

Controller for AJAX-based navigation

```

public class SidebarAjaxbasedController extends SelectorComposer<Component>{

    @Wire
    Grid fnList;

    //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 = fnList.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
    ActionListener<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(fnList.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/essentials/chapter7/ajaxbased/index.zul> to see the result.

Source Code

- ZUL pages ^[1]
- Java ^[2]

References

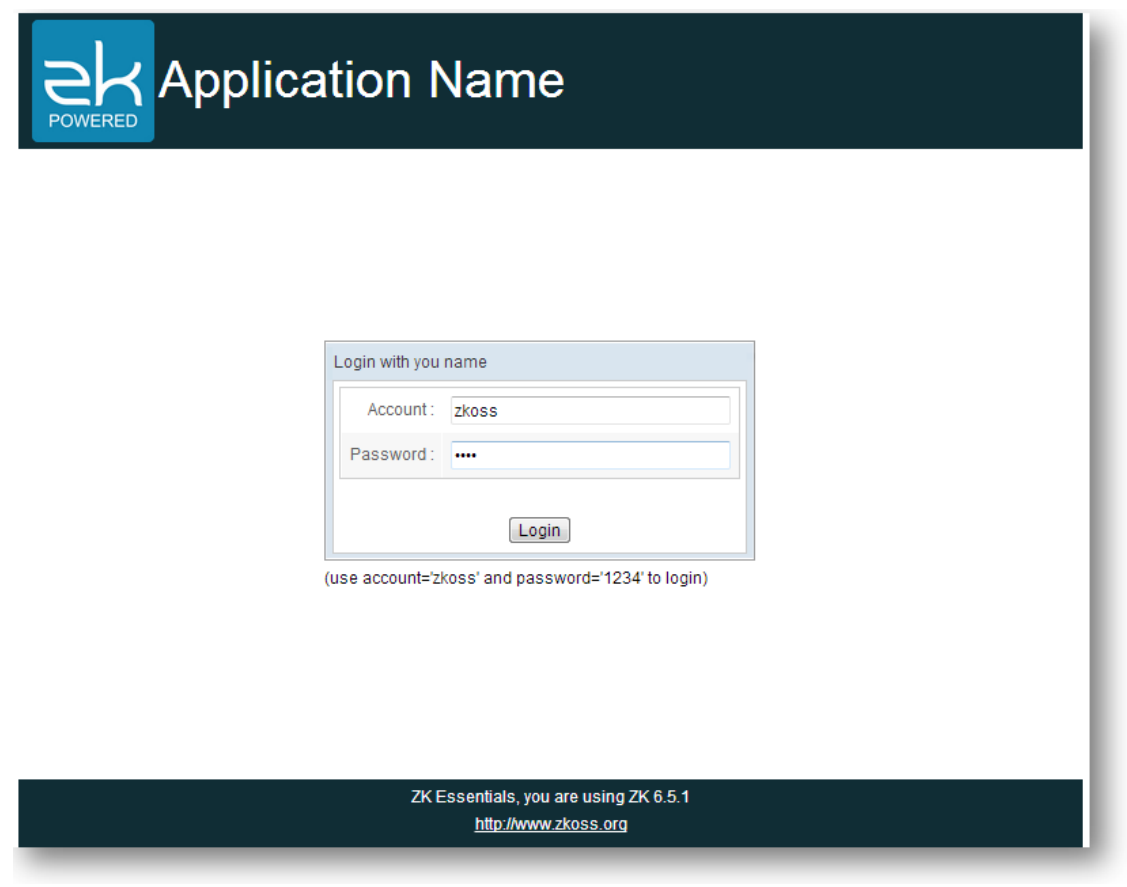
[1] <https://github.com/zkoss/zkessentials/tree/master/src/main/webapp/chapter7>

[2] <https://github.com/zkoss/zkessentials/tree/master/src/main/java/org/zkoss/essentials/chapter7>

Chapter 8: Authentication

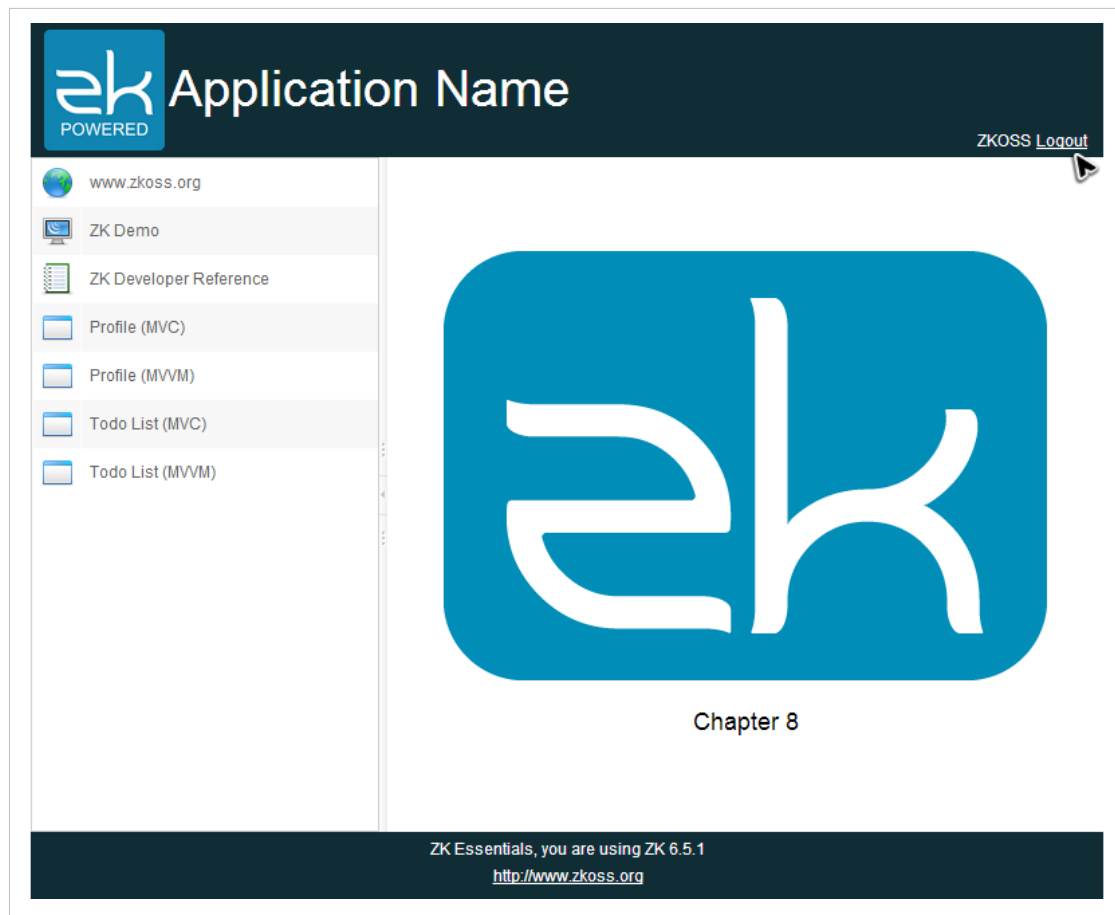
Target Application

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:



The screenshot shows a login page for an application. At the top, there is a dark blue header bar with the 'ek POWERED' logo on the left and the text 'Application Name' on the right. In the center of the page, there is a light blue login box titled 'Login with you name'. Inside this box, there are two input fields: 'Account:' with the value 'zkoss' and 'Password:' with four dots. Below these fields is a 'Login' button. Underneath the login box, there is a note: '(use account='zkoss' and password='1234' to login)'. At the bottom of the page, there is a dark blue footer bar containing the text 'ZK Essentials, you are using ZK 6.5.1' and a link to 'http://www.zkoss.org'.

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



Authentication & 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 Session^[1] 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.

Secure Your Pages

Even if you have setup 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

```
public class AuthenticationServiceChapter5Impl 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 5, 6: As we mentioned, we can get current session and check user's credentials to verify its authentication status.

Log in/ log out

```
public class AuthenticationServiceChapter8Impl extends
AuthenticationServiceChapter5Impl{

    UserInfoService userInfoService = new
UserInfoServiceChapter5Impl();

    @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 initialize a zul page by implementing an Initiator ^[2]. When we apply an initiator to a zul, ZK will use it to perform initialization 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 login page.

Page initiator to check a user's credentials

```

public class AuthenticationInit implements Initiator {

    //services
    AuthenticationService authService = new
AuthenticationServiceChapter8Impl();

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

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

```

- Line 1, 6: A page initiator class should implement Initiator ^[2] 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.

chapter8/index.zul

```
<?link rel="stylesheet" type="text/css" href="/style.css"?>
<!-- protect page by the authentication init -->
<?init class="org.zkoss.essentials.chapter8.AuthenticationInit"?>
<!-- authentication init have to locate before composition -->
<?init class="org.zkoss.zk.ui.util.Composition" arg0="/chapter8/layout/template.zul"?>

<zk>
    <include id="mainInclude" self="@define(content)" src="/chapter8/home.zul"/>
</zk>
```

- 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/essentials/chapter8/index.zul> without successful authentication, you still see the login page.

The "if" Attribute

We can use an EL expression in the `if` attribute to determine a component's creation according to a user's credentials in the session. If the evaluation result of the EL expression is true, the component will show otherwise the component will not be created.

chapter8/layout/template.zul

```
<zk>
    <!-- create only when the currentUser is not an anonymous -->
    <borderlayout hflex="1" vflex="1" if="${not sessionScope.userCredential.anonymous}">
        ...
    </borderlayout>
    <div if="${sessionScope.userCredential.anonymous}">
        Redirect to login page.....
    </div>
</zk>
```

- Line 3,6: `sessionScope` is an implicit variable that allows you to access session's attributes. The *Borderlayout* is only created when `userCredential` is not anonymous, otherwise it shows the *Div* for redirect.

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, it has no sidebar because users, without logging in, should not be able to access main functions.

chapter8/layout/template-anonymous.zul

```
<zk>
    <!-- free to access template, without sidebar -->
    <borderlayout hflex="1" vflex="1">
        <north height="100px" border="none" >
            <include src="/chapter8/layout/banner.zul"/>
        </north>
```

```

        <center id="mainContent" autoscroll="true" border="none" self="@insert (content)
            <!-- the main content will be insert to here -->
        </center>
        <south height="50px" border="none">
            <include src="/chapter3/footer.zul"/>
        </south>
    </borderlayout>
</zk>

```

- Line 3: As this page is made for anonymous users, we don't have to protect this page with `if` attribute.
- Line 7: Define an anchor named `content`.

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

chapter/8/login.zul

```

<?link rel="stylesheet" type="text/css" href="/style.css"?>
<!-- it is a login page, no authentication protection and use anonymous template -->
<?init class="org.zkoss.zk.ui.util.Composition" arg0="/chapter8/layout/template-anonymous
<zk>
    <hbox self="@define(content)" vflex="1" hflex="1" align="center"
        pack="center" spacing="20px">
        <vlayout>
            <window id="loginWin"

apply="org.zkoss.essentials.chapter8.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="#{160;" />
            <button id="login" label="Login" />

```

```

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

```

- Line 3: Apply a template zul with `<?init ?>`.
- Line 5: Define a fragment to be inserted in the anchor content.
- Line 24: 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 chapter8/login.zul

```

public class LoginController extends SelectorComposer<Component> {

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

    //services
    AuthenticationService authService = new
AuthenticationServiceChapter8Impl();

    @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("/chapter8/");

    }
}

```

- 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.

chapter8/layout/banner.zul

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

        <hbox apply="org.zkoss.essentials.chapter8.LogoutController"
            hflex="1" vflex="1" pack="end" align="end" >
            <label value="${sessionScope.userCredential.name}" if="${not sessionScope.userCredential}" />
            <label id="logout" value="Logout" if="${not sessionScope.userCredential}" />
        </hbox>
    </hbox>
</div>
```

- Line 7: The `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.

chapter8/layout/banner.zul

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

        <hbox apply="org.zkoss.essentials.chapter8.LogoutController"
            hflex="1" vflex="1" pack="end" align="end" >
            <label value="${sessionScope.userCredential.name}" if="${not sessionScope.userCredential}" />
            <label id="logout" value="Logout" if="${not sessionScope.userCredential}" />
        </hbox>
    </hbox>
</div>
```

- Line 8: We listen `onClick` event on logout label to perform logout action.

LogoutController.java

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

    //services
    AuthenticationService authService = new
    AuthenticationServiceChapter8Impl();
```



```
@Listen("onClick=#logout")
public void doLogout() {
    authService.logout();
    Executions.sendRedirect("/chapter8/");
}
}
```

- 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/essentials/chapter8> to see the result.

Source Code

- ZUL pages ^[3]
- Java ^[4]

References

- [1] <http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/Session.html#>
- [2] <http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/util/Initiator.html#>
- [3] <https://github.com/zkoss/zkessentials/tree/master/src/main/webapp/chapter8>
- [4] <https://github.com/zkoss/zkessentials/tree/master/src/main/java/org/zkoss/essentials/chapter8>

Chapter 9: Spring Integration

Overview

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();

    ...
}
```

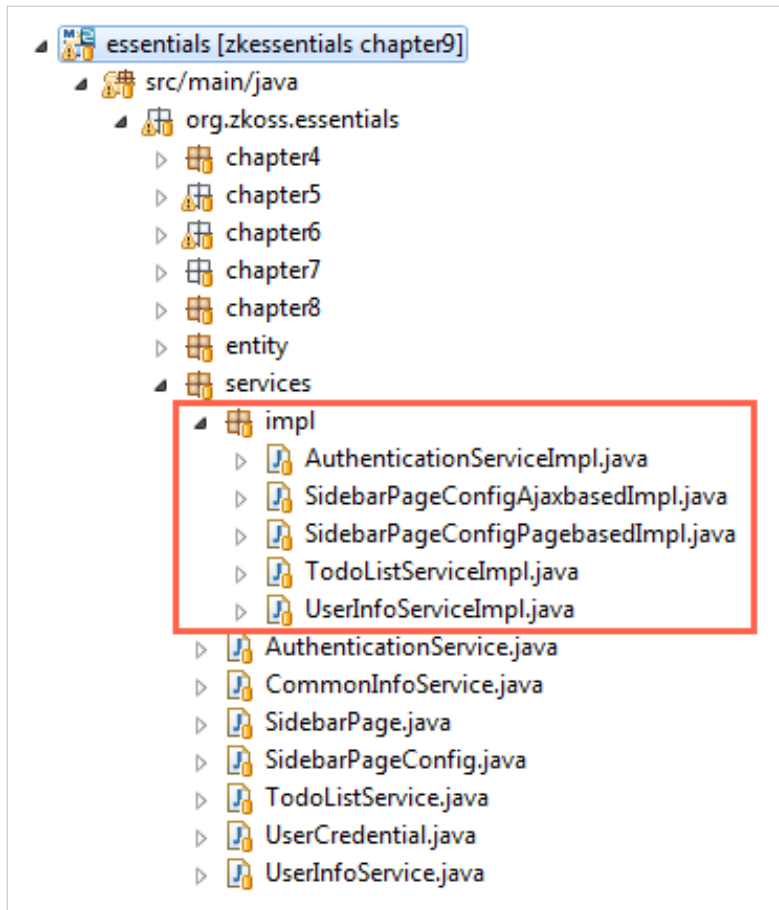
- ProfileViewController depends on AuthenticationService.

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 Chapter 2, our source code has 3 branches in github. The source code of this chapter's example belongs to the branch: **chapter9**. You can select the "chapter9" 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`.



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

```
<properties>
  <zk.version>6.5.1</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 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: The `ContextLoaderListener` reads Spring configuration, and default location is `WEB-INF/applicationContext.xml`.
- Line 20: 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. For `base-package`, you should specify a common parent package or a comma-separated list that includes all candidate classes.

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 `DelegatingVariableResolver` ^[1]. 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 `@Autowired`. 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 fnList;

    //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 `Selectors`^[2] 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/essentials> to see the result.

Source Code

- ZUL pages^[3]
- Java^[4]

References

- [1] <http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zkplus/spring/DelegatingVariableResolver.html#>
- [2] <http://www.zkoss.org/javadoc/latest/zk/org/zkoss/zk/ui/select/Selectors.html#>
- [3] <https://github.com/zkoss/zkessentials/tree/chapter9/src/main/webapp/>
- [4] <https://github.com/zkoss/zkessentials/tree/chapter9/src/main/java/org/zkoss/essentials>

Chapter 10: JPA Integration

Overview

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

```
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.

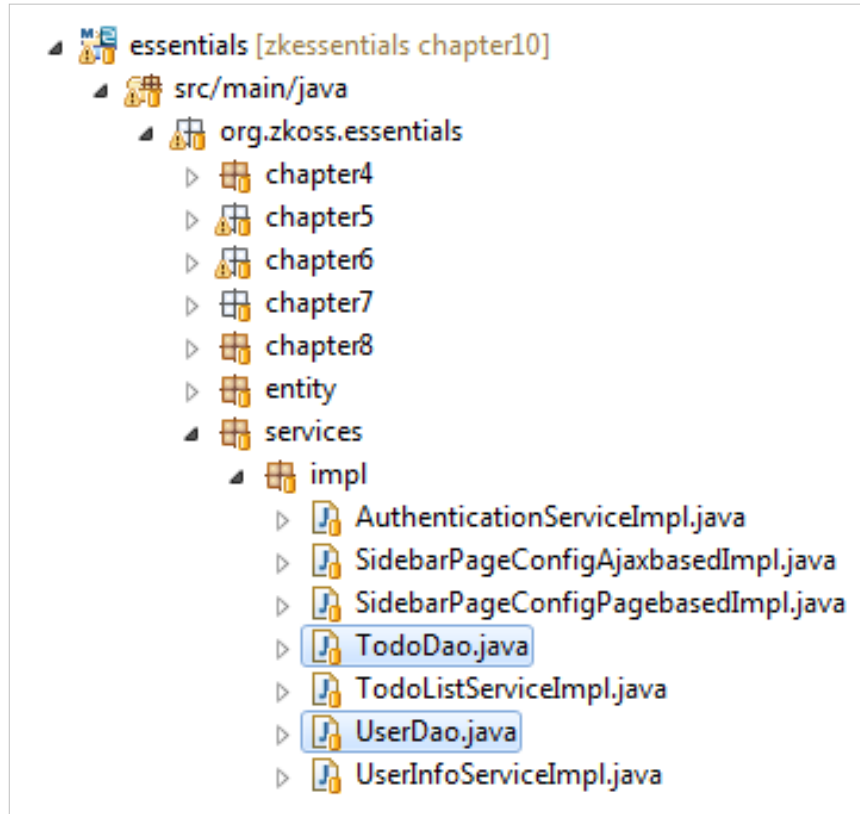
Originally we perform all persistence operations on the list, but we will replace this part with a real database and a persistence framework, *JPA*.

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 Chapter 2, our source code has 3 branches in github. The source code of this chapter's example belongs to a different branch in the github: **chapter10**.

We don't create new examples in this chapter, but we add 2 DAO classes written 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 `cbllib` for Spring framework which is explained in chapter 9.)

```
<properties>
  <zk.version>6.5.1</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 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/

```

```

        <property name="hibernate.hbm2ddl.auto" value="create" />
    </properties>
</persistence-unit>
</persistence>

```

- Line 4: 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 `OpenEntityManagerInViewFilter` 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`

```

...
<!-- 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

```

@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.

```

<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: 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 17: Set property `persistenceUnitName` with the name we specified in `persistence.xml`.
- Line 20,24: 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.

```

@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.

```

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

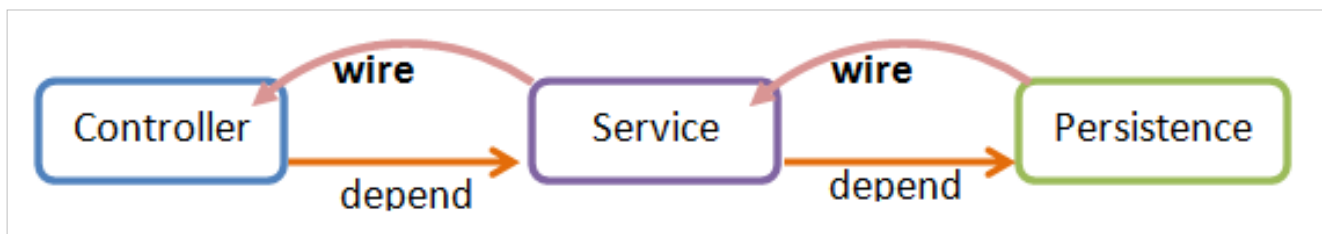
    @Autowired
    TodoDao dao;

    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/essentials> to see the result.

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. 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.

Source Code

- ZUL pages ^[1]
- Java ^[2]

References

[1] <https://github.com/zkoss/zkessentials/tree/chapter10/src/main/webapp/>

[2] <https://github.com/zkoss/zkessentials/tree/chapter10/src/main/java/org/zkoss/essentials>

Article Sources and Contributors

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

Chapter 1: Introduction *Source:* http://new.zkoss.org/index.php?title=ZK_Essentials/Chapter_1:_Introduction *Contributors:* Hawk, Tmillsclare

Chapter 2: Project Structure *Source:* http://new.zkoss.org/index.php?title=ZK_Essentials/Chapter_2:_Project_Structure *Contributors:* Dennischen, Hawk, Tmillsclare

Chapter 3: User Interface and Layout *Source:* http://new.zkoss.org/index.php?title=ZK_Essentials/Chapter_3:_User_Interface_and_Layout *Contributors:* Hawk, Tmillsclare

Chapter 4: Controlling Components *Source:* http://new.zkoss.org/index.php?title=ZK_Essentials/Chapter_4:_Controlling_Components *Contributors:* Hawk, Tmillsclare

Chapter 5: Handling User Input *Source:* http://new.zkoss.org/index.php?title=ZK_Essentials/Chapter_5:_Handling_User_Input *Contributors:* Hawk, Tmillsclare

Chapter 6: Implementing CRUD *Source:* http://new.zkoss.org/index.php?title=ZK_Essentials/Chapter_6:_Implementing_CRUD *Contributors:* Hawk, Tmillsclare

Chapter 7: Navigation and Templating *Source:* http://new.zkoss.org/index.php?title=ZK_Essentials/Chapter_7:_Navigation_and_Templating *Contributors:* Hawk, Tmillsclare

Chapter 8: Authentication *Source:* http://new.zkoss.org/index.php?title=ZK_Essentials/Chapter_8:_Authentication *Contributors:* Hawk, Tmillsclare

Chapter 9: Spring Integration *Source:* http://new.zkoss.org/index.php?title=ZK_Essentials/Chapter_9:_Spring_Integration *Contributors:* Hawk, Tmillsclare

Chapter 10: JPA Integration *Source:* http://new.zkoss.org/index.php?title=ZK_Essentials/Chapter_10:_JPA_Integration *Contributors:* Hawk, Tmillsclare

Image Sources, Licenses and Contributors

File: tutorial-ch1-simple-architecture.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch1-simple-architecture.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch8-login.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch8-login.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch1-profile.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch1-profile.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch1-todo.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch1-todo.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch2-3branches.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch2-3branches.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch2-download-zip.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch2-download-zip.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch2-index.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch2-index.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch2-project-structure-java.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch2-project-structure-java.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch2-project-structure-webapp.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch2-project-structure-webapp.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch3-page-layout.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch3-page-layout.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch3-borderlayout.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch3-borderlayout.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch3-layout.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch3-layout.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch5-app.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch5-app.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch5-email-constraint.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch5-email-constraint.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch5-collection.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch5-collection.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch5-save.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch5-save.png> *License:* unknown *Contributors:* Hawk

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

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

File: Tutorial-ch5-unsaved.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch5-unsaved.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch5-form-binding.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch5-form-binding.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch6-app.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch6-app.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch6-app-selected.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch6-app-selected.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch6-plus.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch6-plus.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch6-cross.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch6-cross.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch6-priority.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch6-priority.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch7-ajax-based.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch7-ajax-based.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch7-page-based-navigation.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch7-page-based-navigation.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch7-pagebased.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch7-pagebased.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch7-ajax-based-navigation.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch7-ajax-based-navigation.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch8-index.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch8-index.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch9-source.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch9-source.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch10-source.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch10-source.png> *License:* unknown *Contributors:* Hawk

File: Tutorial-ch10-dependencies.png *Source:* <http://new.zkoss.org/index.php?title=File:Tutorial-ch10-dependencies.png> *License:* unknown *Contributors:* Hawk