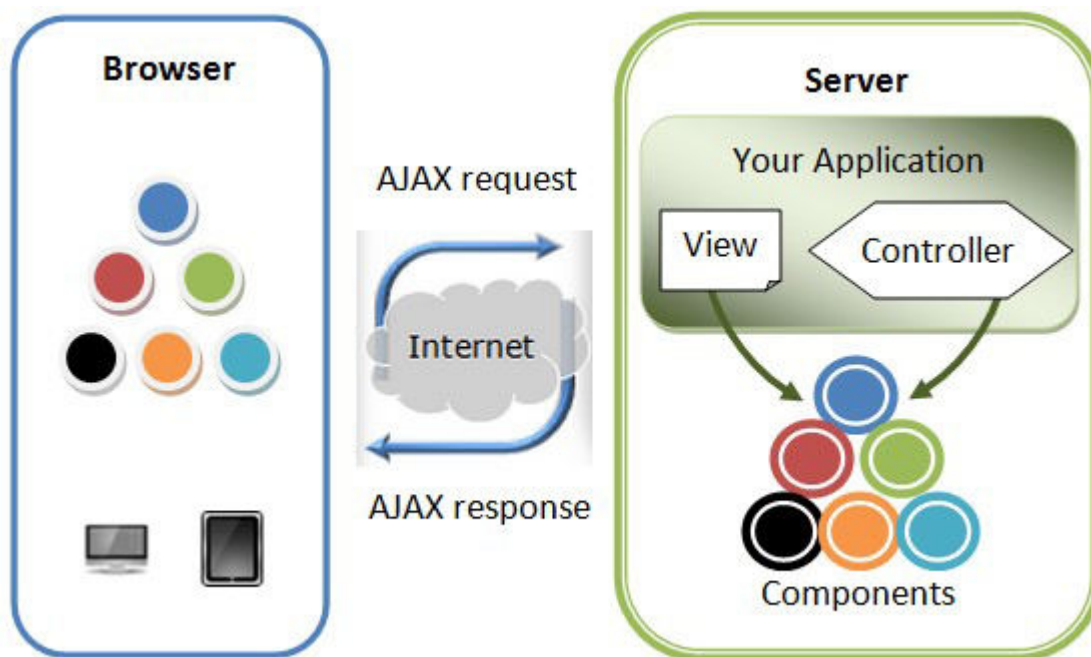


## ZK's Value and Strength

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<sup>1</sup> 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<sup>2</sup>, ZK also supports another design pattern, MVVM (Model-View-ViewModel)<sup>3</sup> 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

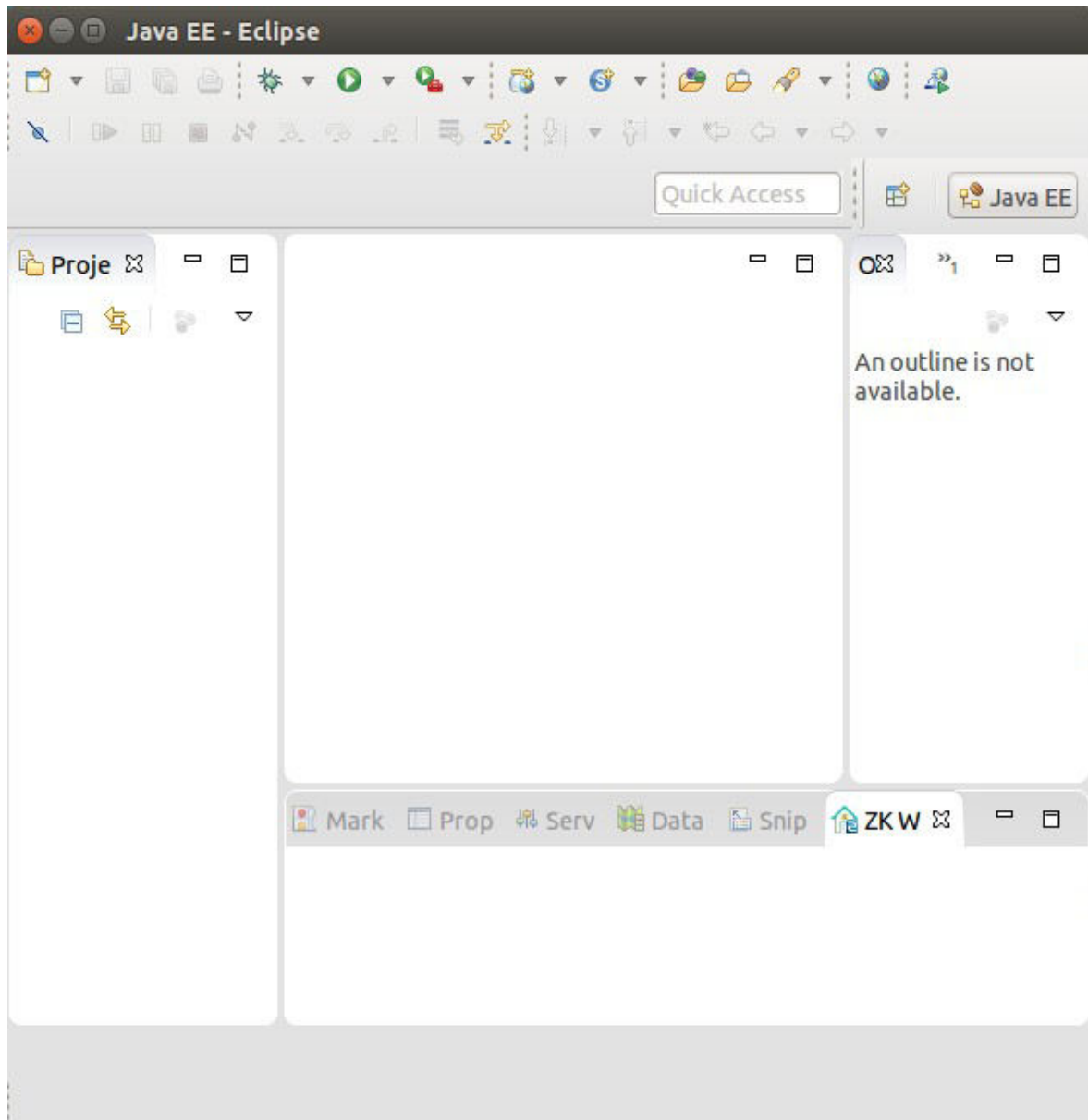


Above image is a simplified ZK architecture. When a browser visits a page of a ZK application, ZK creates components written in ZUL and renders them on the browser. You can manipulate components by your application's Controller to implement UI presentation logic. All changes you made on components will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

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

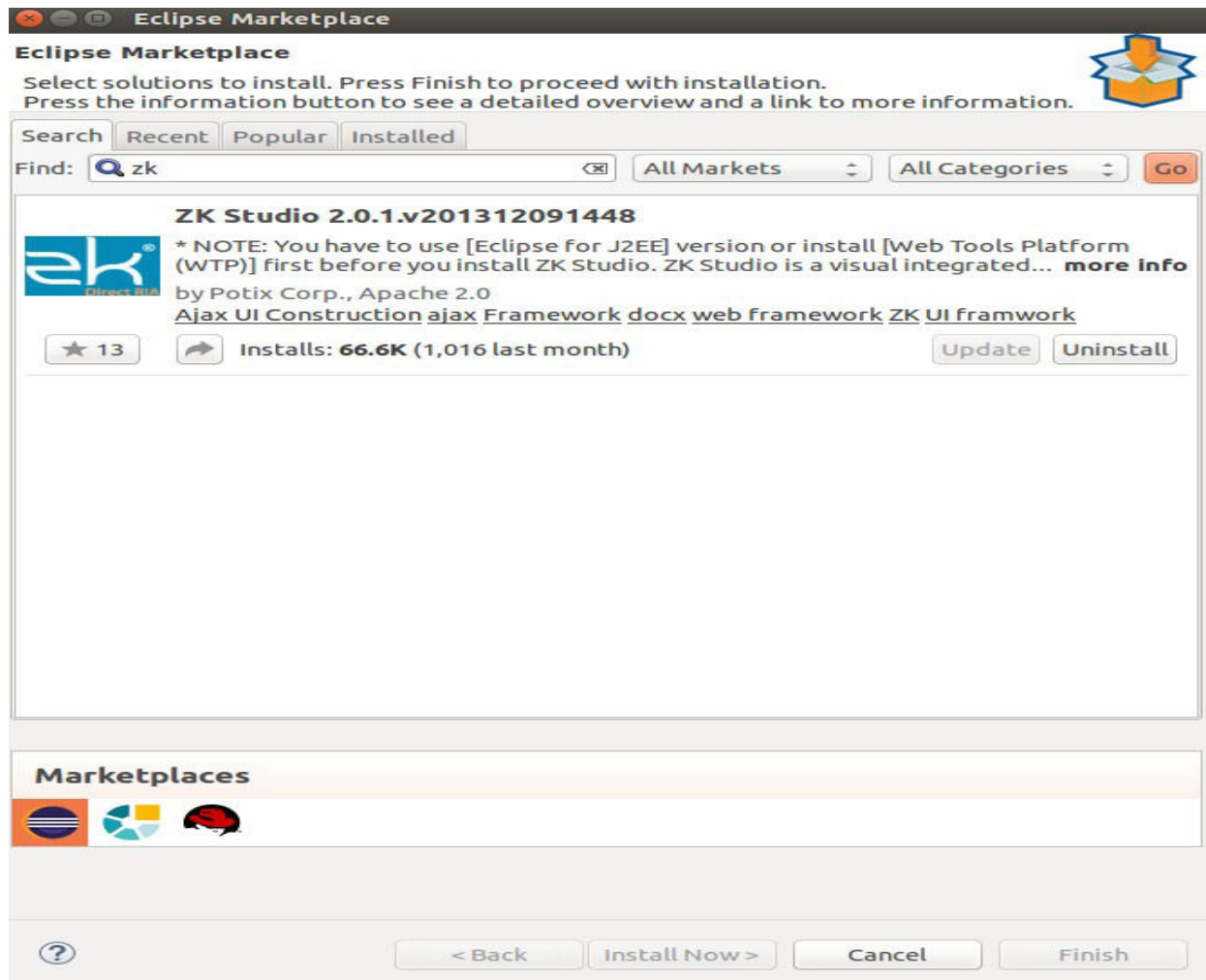
## 2 . Run Example Application

- open Eclips



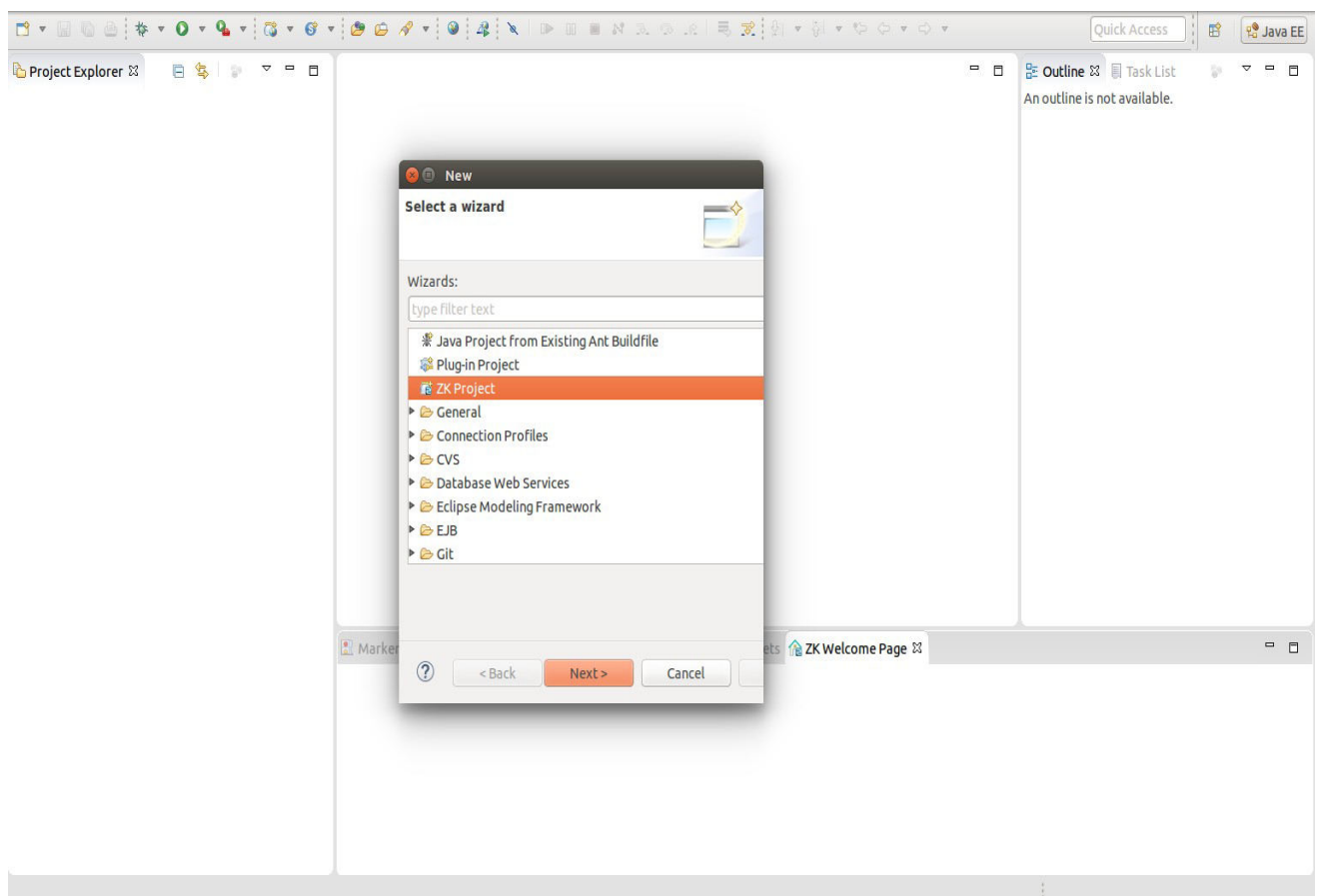
- click on Help Bar open Eclipse MarketPlace
- Type Zk in Find TextBox
- Zk Studio will be Appear
- Click on Install (it will come for You)
- Click on Finish



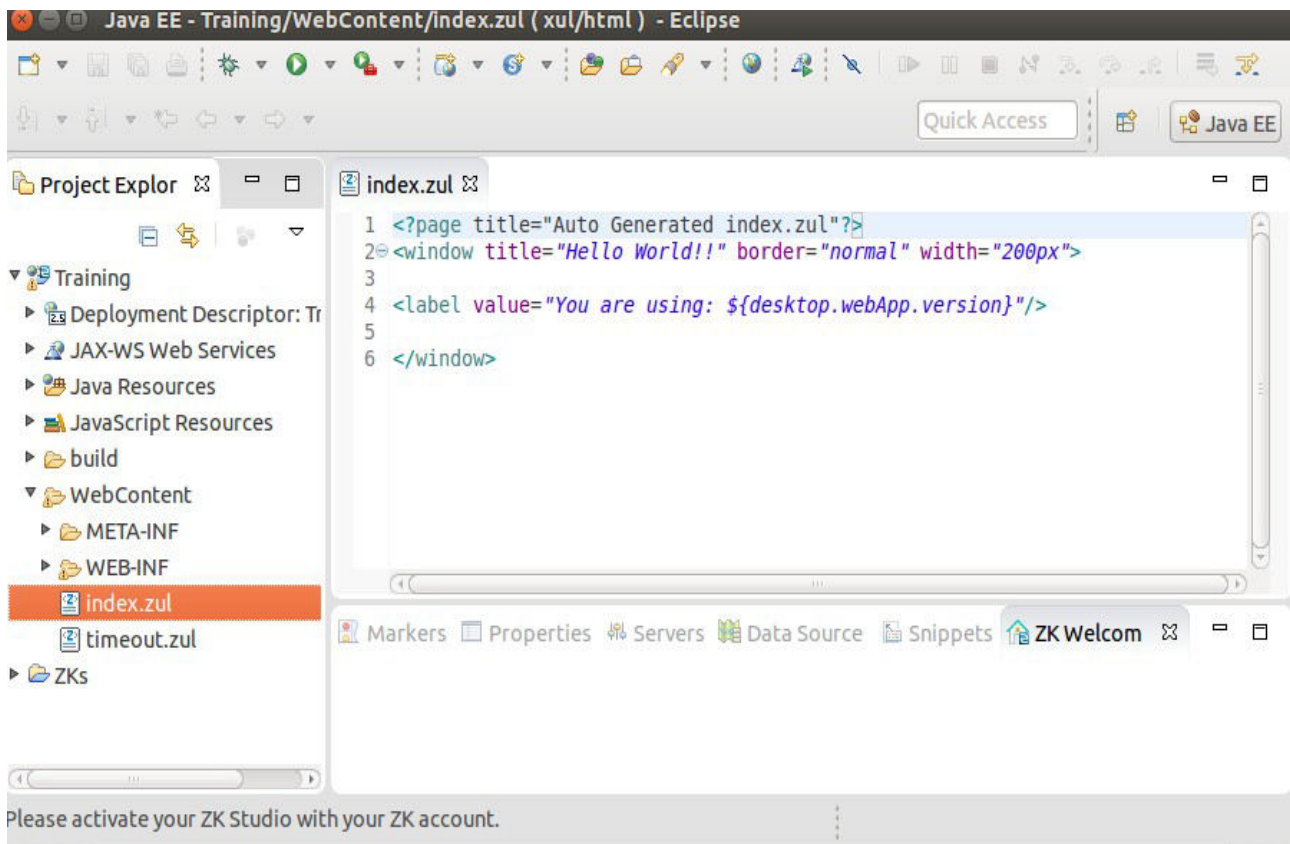


## Creating New Project

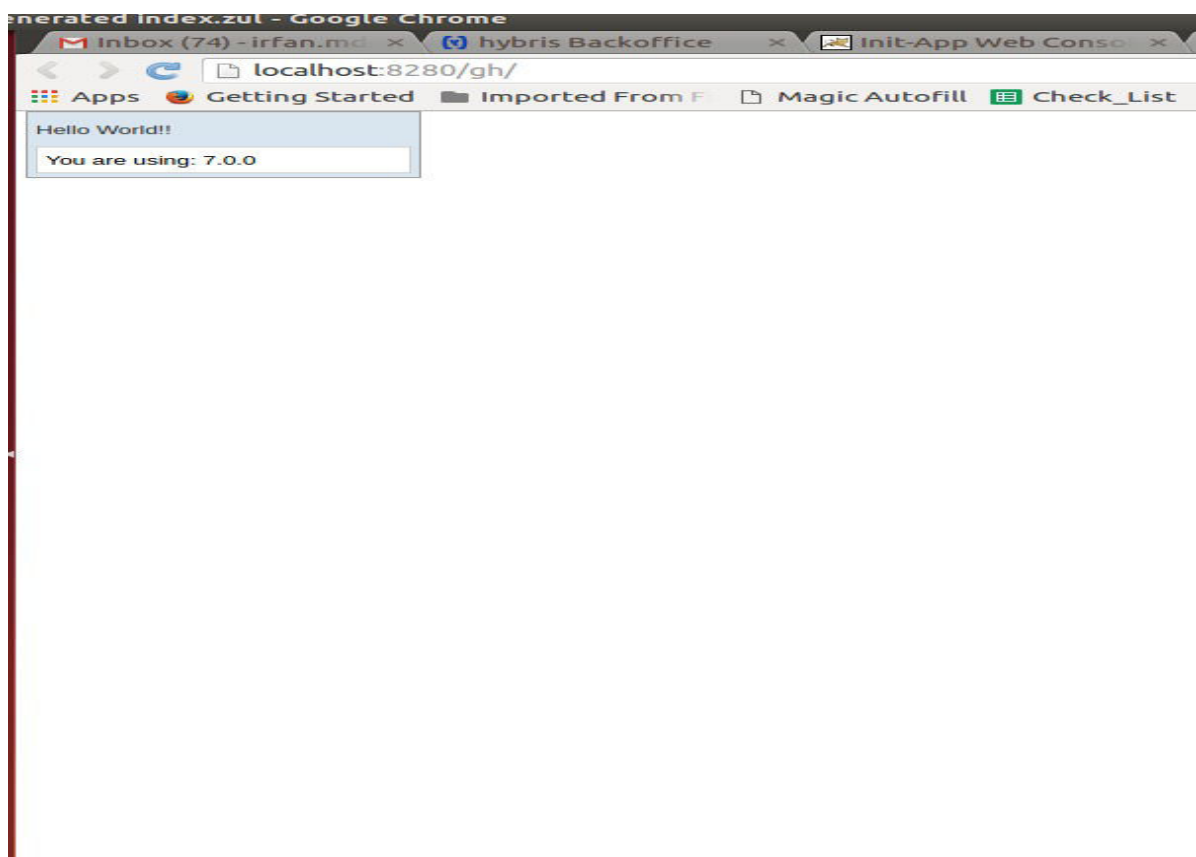
- click on File --->New -->other -->



- Click on Zk Project
- Click on next
- Write your Project Name
- Click on Finish
- New Project Will be Appear in your Eclipse



- Go inside WebContent Folder You Will See index.zul
- Now Right Click on Your Project Goto RunAs Choose Run As Server
- Click on Finish



## Build the View

Building the View in ZK is basically creating components, and there are two ways to do it: Java (programmatic) 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.<sup>1</sup> First, create a new text file with name `index.zul`, and type the following content:

#### `index.zul`

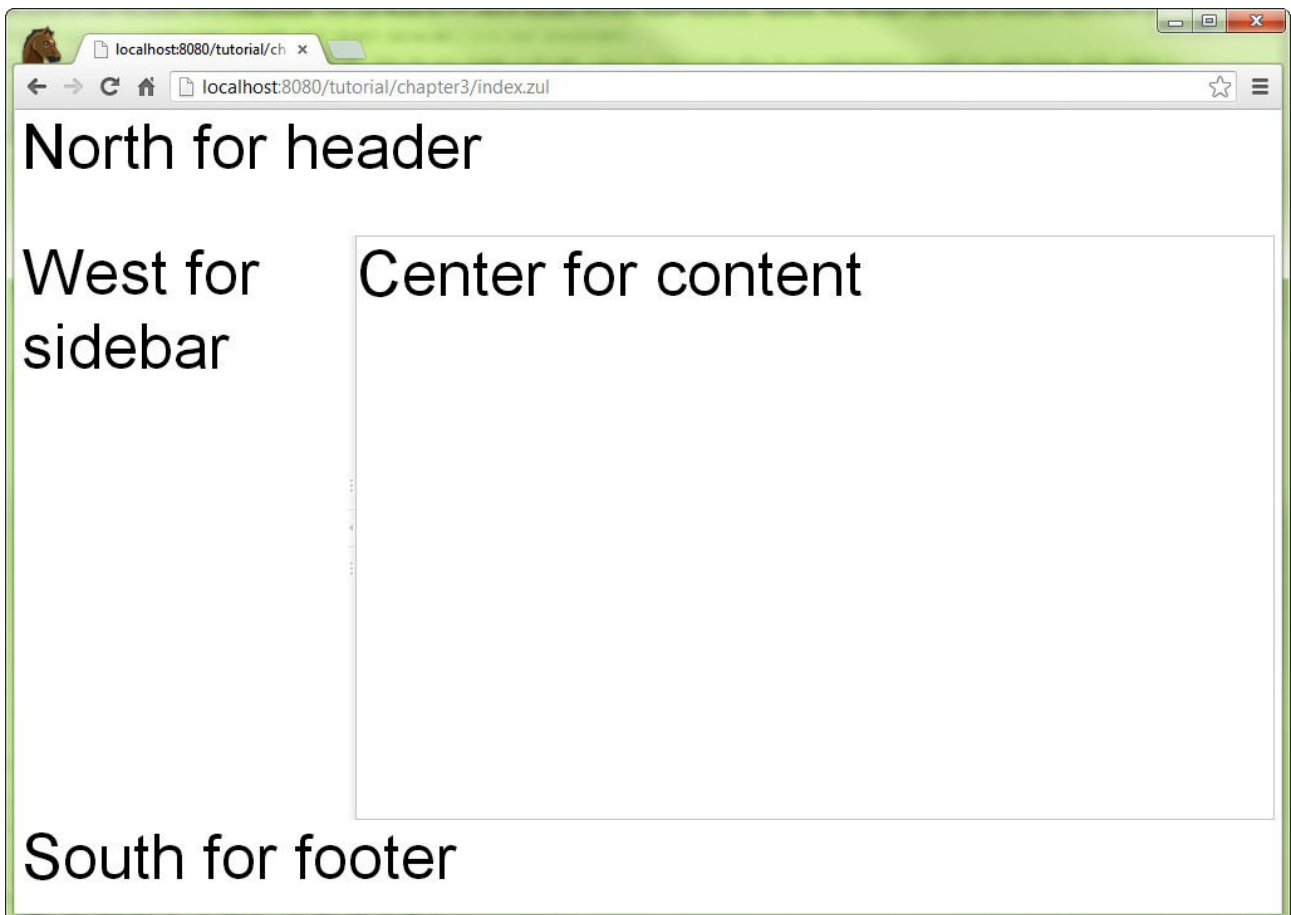
```
1  <zk>
2      <borderlayout hflex="1" vflex="1">
3          <north height="100px" border="none" >
4              <label style="font-size:50px">North for header</label>
5          </north>
6          <west width="260px" border="none" collapsible="true"
7              splittable="true" minsize="300">
8              <label style="font-size:50px">West for sidebar</label>
9          </west>
10         <center id="mainContent" autoscroll="true">
11             <label style="font-size:50px">Center for content</label>
12         </center>
13         <south height="50px" border="none">
14             <label style="font-size:50px">South for footer</label>
15         </south>
16     </borderlayout>
17 </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, 7: Setting `collapsible` to true allows you to collapse the *West* area by clicking an arrow button. Setting `splittable` to true allows you to adjust the width of *West* and `minsize` limits the minimal size of width you can adjust.
- Line 10: Setting `autoscroll` to true will decorate the *Center* with a scroll bar when *Center* contains lots of information that exceed the its height.
- Line 4,8,11,14: These *Labels* are just for identifying *BorderLayout*'s areas and we will remove them in the final result.

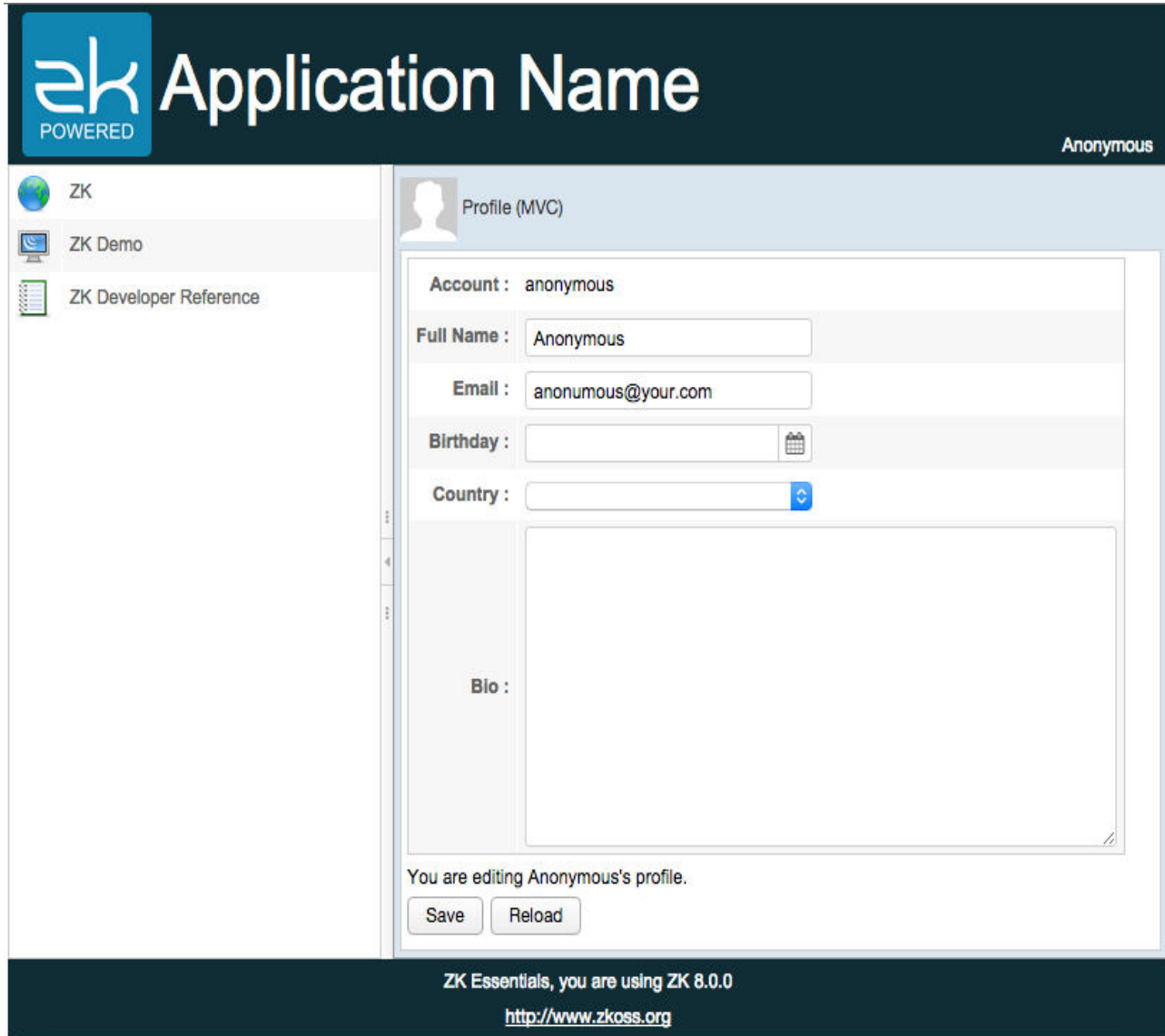
Then, you can view the result from your browser as below:





# 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' logo with 'POWERED' underneath, followed by the text 'Application Name' in large white font, and the word 'Anonymous' in the top right corner. A left sidebar lists navigation 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 : anonymous', 'Full Name : Anonymous' (text input), 'Email : anonumous@your.com' (text input), 'Birthday : ' (text input with a calendar icon), and 'Country : ' (dropdown menu). Below these is a 'Bio : ' label followed by a large text area. At the bottom of the form, a message states 'You are editing Anonymous's profile.' with 'Save' and 'Reload' buttons. A footer bar at the bottom of the page reads 'ZK Essentials, you are using ZK 8.0.0' and provides 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.



# 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*.

### profile-mvc.zul

```
1  <?link rel="stylesheet" type="text/css" href="/style.css"?>
2  <window apply="org.zkoss.essentials.chapter3.mvc.ProfileViewController"
3      border="normal" hflex="1" vflex="1" contentType="overflow:auto">
4      <caption src="/imgs/profile.png" sclass="fn-caption"
5          label="Profile (MVC)"/>
6      <vlayout>
7          <grid width="500px">
8              <columns>
9                  <column align="right" hflex="min"/>
10                 <column/>
11             </columns>
12             <rows>
13                 <row>
14                     <cell sclass="row-title">Account :</cell>
15                     <cell><label id="account"/></cell>
16                 </row>
17                 <row>
18                     <cell sclass="row-title">Full Name :</cell>
19                     <cell>
20                         <textbox id="fullName"
21                             constraint="no empty: Please enter your full name"
22                             width="200px"/>
23                     </cell>
24                 </row>
25                 <row>
26                     <cell sclass="row-title">Email :</cell>
27                     <cell>
28                         <textbox id="email"
29                             constraint="/.+@.+.[a-z]+/: Please enter an e-mail address"
30                             width="200px"/>
31                     </cell>
32                 </row>
33                 <row>
34                     <cell sclass="row-title">Birthday :</cell>
35                     <cell>
36                         <datebox id="birthday"
37                             constraint="no future" width="200px"/>
38                     </cell>
39                 </row>
40             </rows>
41         </grid>
42     </vlayout>
43 </window>
```

```

38         </cell>
39     </row>
40     <row>
41         <cell sclass="row-title">Country :</cell>
42         <cell>
43             <listbox id="country" mold="select" width="200px">
44                 <template name="model">
45                     <listitem label="{each}" />
46                 </template>
47             </listbox>
48         </cell>
49     </row>
50     <row>
51         <cell sclass="row-title">Bio :</cell>
52         <cell><textbox id="bio" multiline="true"
53             hflex="1" height="200px" />
54         </cell>
55     </row>
56 </rows>
57 </grid>
58 <div>You are editing <label id="nameLabel"/>'s profile.</div>
59 <hlayout>
60     <button id="saveProfile" label="Save"/>
61     <button id="reloadProfile" label="Reload"/>
62 </hlayout>
63 </vlayout>
64 </window>

```

- Line 4, 5: [Caption](#) can be used to build compound header with an image for a [Window](#).
- Line 6: [Vlayout](#) is a light-weight layout component which arranges child components vertically without splitter, align, and pack support.
- Line 14: [Cell](#) is used inside *Row*, *Hbox*, or *Vbox* for fully controlling style and layout.
- Line 21, 22, 29, 30, 37: Specify `constraint` attribute of an input component can activate input validation feature and we will discuss it in later section.
- Line 43: Some components have multiple molds, and each mold has its own style.
- Line 44: *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 45: 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 59: [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:

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

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

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

```
1 | <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)". The form contains several input fields: "Account" (text: anonymous), "Full Name" (text: Anonymous), "Email" (text: anonumous), "Birthday" (calendar icon), "Country" (dropdown menu), and "Bio" (text area). A red error message box is displayed next to the "Email" field, containing the text "Please enter an e-mail address". At the bottom of the form, there are "Save" and "Reload" buttons.

## 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.

**Profile (MVC)**

Account : anonymous

Full Name : Anonymous

Email : anonumous@your.com

Birthday :

Country :

- Afghanistan
- Aland Islands
- Albania
- Algeria
- American Samoa
- Andorra
- Angola
- Anguilla
- Antarctica
- Antigua and Barbuda
- Argentina
- Armenia
- Aruba
- Australia
- Austria
- Azerbaijan
- Bahamas
- Bahrain
- Bangladesh
- Barbados

Bio :

### 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 `org.zkoss.zul.ListModelList` object.

""Initialize data model for a *Listbox* ""

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 `org.zkoss.zul.ListModelList` object.

""Initialize data model for a *Listbox* ""

```

1 public class ProfileViewController extends SelectorComposer<Component>{
2     ...
3     @Wire
4     Listbox country;
5
6     @Override
7     public void doAfterCompose(Component comp) throws Exception{
8         super.doAfterCompose(comp);
9
10        ListModelList<String> countryModel = new ListModelList<String>(CommonInfoServi
11        country.setModel(countryModel);
12
13        ...
14    }
15
16    ...
17
18 }

```

- 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.



```

21
22     @Override
23     public void doAfterCompose(Component comp) throws Exception{
24         super.doAfterCompose(comp);
25
26         ListModelList<String> countryModel = new ListModelList<String>(CommonInfoServi
27         country.setModel(countryModel);
28
29         refreshProfileView();
30     }
31
32     ...
33
34     private void refreshProfileView() {
35         UserCredential cre = authService.getUserCredential();
36         User user = userInfoService.findUser(cre.getAccount());
37         if(user==null){
38             //TODO handle un-authenticated access
39             return;
40         }
41
42         //apply bean value to UI components
43         account.setValue(user.getAccount());
44         fullName.setValue(user.getFullName());
45         email.setValue(user.getEmail());
46         birthday.setValue(user.getBirthday());
47         bio.setValue(user.getBio());
48
49         ((ListModelList)country.getModel()).addToSelection(user.getCountry());
50         ...
51     }
52 }

```

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.

The screenshot shows a web form titled "Profile (MVC)". It contains several input fields: "Account" (anonymous), "Full Name" (Anonymous), "Email" (anonumous@your.com), "Birthday" (Mar 4, 1981), and "Country" (Costa Rica). A teal notification box with an information icon and the text "Your profile is updated" is overlaid on the form. Below the form, there is a text label "You are editing Anonymous's profile." and two buttons: "Save" and "Reload". A mouse cursor is clicking the "Save" button.

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

```

1 public class ProfileViewController extends SelectorComposer<Component>{
2
3     @Listen("onClick=#saveProfile")
4     public void doSaveProfile(){
5         ...
6     }
7     ...
8 }

```



- 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

```

1 public class ProfileViewController extends SelectorComposer<Component>{
2
3
4     @Listen("onClick=#saveProfile")
5     public void doSaveProfile(){
6         UserCredential cre = authService.getUserCredential();
7         User user = userInfoService.findUser(cre.getAccount());
8         if(user==null){
9             //TODO handle un-authenticated access
10            return;
11        }
12
13        //apply component value to bean
14        user.setFullName(fullName.getValue());
15        user.setEmail(email.getValue());
16        user.setBirthday(birthday.getValue());
17        user.setBio(bio.getValue());
18
19        Set<String> selection = ((ListModelList)country.getModel()).getSelection();
20        if(!selection.isEmpty()){
21            user.setCountry(selection.iterator().next());
22        }else{
23            user.setCountry(null);
24        }
25
26        userInfoService.updateUser(user);
27
28        Clients.showNotification("Your profile is updated");
29    }
30    ...
31 }

```

- 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().

```

1 public class ProfileViewController extends SelectorComposer<Component>{
2
3
4     //wire components
5     @Wire
6     Label account;
7     @Wire
8     Textbox fullName;
9     @Wire
10    Textbox email;
11    @Wire
12    Datebox birthday;
13    @Wire
14    Listbox country;
15    @Wire
16    Textbox bio;
17
18    ...
19    @Listen("onClick=#reloadProfile")
20    public void doReloadProfile(){
21        refreshProfileView();
22    }
23
24    ...
25 }

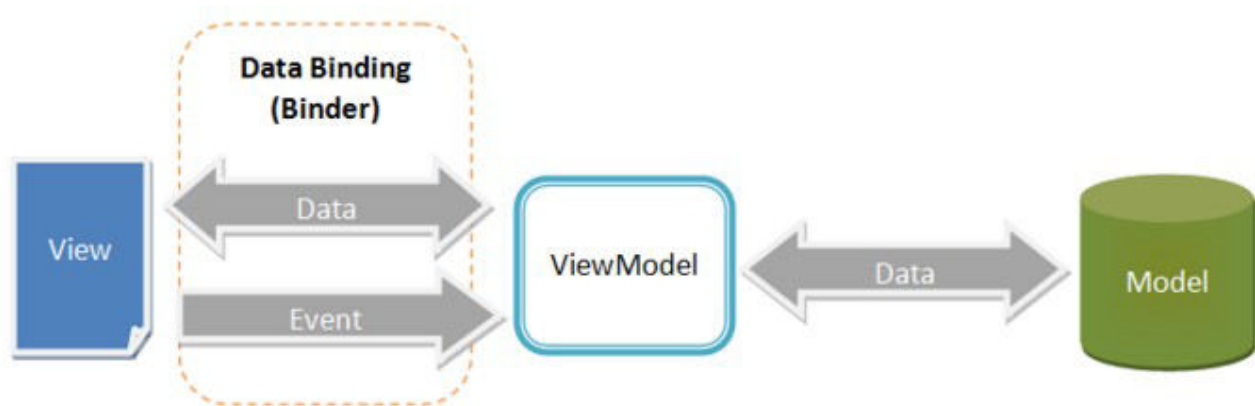
```

- 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/chapter3/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 [chapter3/profile-mvvm-property.zul](#)

```

1  <?link rel="stylesheet" type="text/css" href="/style.css"?>
2  <window border="normal" hflex="1" vflex="1" contentStyle="overflow:auto">
3      <caption src="/imgs/profile.png" sclass="fn-caption"
4          label="Profile (MVVM)"/>
5      <vlayout>
6          <grid width="500px" >
7              <columns>
8                  <column align="right" hflex="min"/>
9                  <column/>
10             </columns>
11             <rows>
12                 <row>
13                     <cell sclass="row-title">Account :</cell>
14                     <cell><label/></cell>
15                 </row>
16                 <row>
17                     <cell sclass="row-title">Full Name :</cell>
18                     <cell>
19                         <textbox
20                             constraint="no empty: Plean enter your full name"
21                             width="200px"/>
22                     </cell>
23                 </row>
24                 <row>
25                     <cell sclass="row-title">Email :</cell>
26                     <cell>
27                         <textbox
28                             constraint="/.+@.+\.[a-z]+/: Please enter an e-mail address"
29                             width="200px"/>
30                     </cell>
31                 </row>
32                 <row>
33                     <cell sclass="row-title">Birthday :</cell>
34                     <cell><datebox constraint="no future" width="200px"/>
35                     </cell>
36                 </row>
37                 <row>
38                     <cell sclass="row-title">Country :</cell>
39                     <cell>
40                         <listbox mold="select" width="200px">
41                             <template name="model">
42                                 <listitem />
43                             </template>
44                         </listbox>
45                     </cell>
46                 </row>
47                 <row>
48                     <cell sclass="row-title">Bio :</cell>
49                     <cell>
50                         <textbox
51                             multiline="true" hflex="1" height="200px" />
52                     </cell>
53                 </row>
54             </rows>
55         </grid>
56         <div>You are editing <label />'s profile.</div>
57         <hlayout>
58             <button label="Save"/>
59             <button label="Reload"/>
60         </hlayout>
61     </vlayout>
62 </window>

```

- Line 41: 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

```
1  public class ProfileViewModel implements Serializable{
2
3
4      //services
5      AuthenticationService authService = new AuthenticationServiceChapter3Impl();
6      UserInfoService userInfoService = new UserInfoServiceChapter3Impl();
7
8      //data for the view
9      User currentUser;
10
11     public User getCurrentUser(){
12         return currentUser;
13     }
14
15     public List<String> getCountryList(){
16         return CommonInfoService.getCountryList();
17     }
18
19     @Init // @Init annotates a initial method
20     public void init(){
21         UserCredential cre = authService.getUserCredential();
22         currentUser = userInfoService.findUser(cre.getAccount());
23         if(currentUser==null){
24             //TODO handle un-authenticated access
25             return;
26         }
27     }
28     ...
29 }
```

- 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

```

1 |
2 | public class ProfileViewModel implements Serializable{
3 |     ...
4 |
5 |     @Command //@Command annotates a command method
6 |     public void save(){
7 |         currentUser = userInfoService.updateUser(currentUser);
8 |         Clients.showNotification("Your profile is updated");
9 |     }
10 |
11 |     @Command
12 |     public void reload(){
13 |         UserCredential cre = authService.getUserCredential();
14 |         currentUser = userInfoService.findUser(cre.getAccount());
15 |     }
16 | }

```

- 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

```
1 |
2 | public class ProfileViewModel implements Serializable{
3 |     ...
4 |
5 |     @Command //@Command annotates a command method
6 |     @NotifyChange("currentUser") //@NotifyChange annotates data changed notification a
7 |     public void save(){
8 |         currentUser = userInfoService.updateUser(currentUser);
9 |         Clients.showNotification("Your profile is updated");
10 |    }
11 |
12 |    @Command
13 |    @NotifyChange("currentUser")
14 |    public void reload(){
15 |        UserCredential cre = authService.getUserCredential();
16 |        currentUser = userInfoService.findUser(cre.getAccount());
17 |    }
18 | }
```

- 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.



```

1
2 <window apply="org.zkoss.bind.BindComposer"
3     viewModel="@id('vm') @init('org.zkoss.essentials.chapter3.mvvm.ProfileViewModel')"
4     border="normal" hflex="1" vflex="1" contentStyle="overflow:auto">
5     ...
6 </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 previous 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`.

```

1 ...
2 <cell>
3     <listbox model="@load(vm.countryList)" mold="select" width="200px">
4         <template name="model">
5             <listitem label="@load(each)" />
6         </template>
7     </listbox>
8 </cell>
9 ...

```

- 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.

```
1 <textbox value="@bind(vm.currentUser.fullName)"
2 constraint="no empty: Plean enter your full name" width="200px"/>
```

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.

```
1 ...
2 <rows>
3   <row>
4     <cell sclass="row-title">Account :</cell>
5     <cell>
6       <label value="@load(vm.currentUser.account)"/>
7     </cell>
8   </row>
9   <row>
10    <cell sclass="row-title">Full Name :</cell>
11    <cell>
12      <textbox value="@bind(vm.currentUser.fullName)"
13        constraint="no empty: Plean enter your full name"
14        width="200px"/>
15    </cell>
16  </row>
17  <row>
18    <cell sclass="row-title">Email :</cell>
19    <cell>
20      <textbox value="@bind(vm.currentUser.email)"
21        constraint="/.+@.+\.[a-z]+/: Please enter an e-mail address"
22        width="200px"/>
23    </cell>
24  </row>
25  <row>
26    <cell sclass="row-title">Birthday :</cell>
27    <cell>
28      <datebox value="@bind(vm.currentUser.birthday)"
29        constraint="no future" width="200px"/>
30    </cell>
```

```

31     </row>
32     <row>
33         <cell sclass="row-title">Country :</cell>
34         <cell>
35             <listbox model="@load(vm.countryList)"
36                 selectedItem="@bind(vm.currentUser.country)"
37                 mold="select" width="200px">
38                 <template name="model">
39                     <listitem label="@load(each)" />
40                 </template>
41             </listbox>
42         </cell>
43     </row>
44     <row>
45         <cell sclass="row-title">Bio :</cell>
46         <cell>
47             <textbox value="@bind(vm.currentUser.bio)"
48                 multiline="true" hflex="1" height="200px" />
49         </cell>
50     </row>
51 </rows>
52
53 ...

```

- Line 12, 20, 28, 47: Use `@bind` to save user input back to the ViewModel and load value from the ViewModel.
- Line 36: 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:

```

2     <hlayout>
3         <button onClick="@command('save')" label="Save"/>
4         <button onClick="@command('reload')" label="Reload"/>
5     </hlayout>

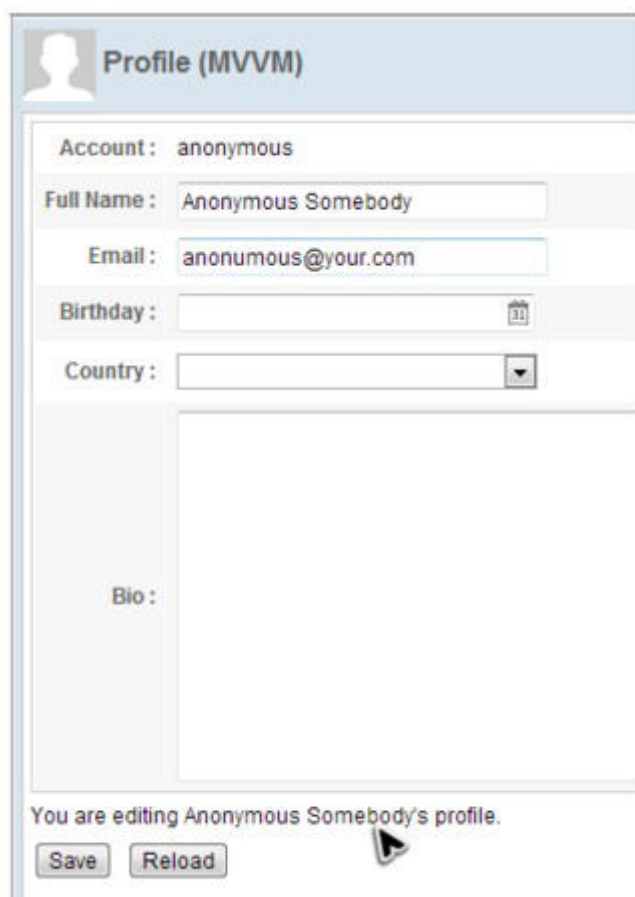
```

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)" with a user icon. The form contains the following fields:

- Account : anonymous
- Full Name : Anonymous Somebody
- Email : anonumous@your.com
- Birthday : [calendar icon]
- Country : [dropdown menu]
- Bio : [text area]

At the bottom of the form, there is a status message: "You are editing Anonymous Somebody's profile." Below this message are two buttons: "Save" and "Reload". A mouse cursor is pointing at the status message.

### Unsaved Input Changes Data

</div> 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).

### profile-mvvm.zul

```
31         <row>
32             <cell sclass="row-title">Country :</cell>
33             <cell>
34                 <listbox model="@load(vm.countryList)"
35                     mold="select" width="200px"
36                     selectedItem="@bind(fx.country)">
37                     <template name="model">
38                         <listitem label="@load(each)"/>
39                     </template>
40                 </listbox>
41             </cell>
42         </row>
43         <row>
44             <cell sclass="row-title">Bio :</cell>
45             <cell><textbox value="@bind(fx.bio)" multiline="true"
46                 hflex="1" height="200px" />
47             </cell>
48         </row>
49     </rows>
50 </grid>
51 <div>You are editing
52     <label value="@load(vm.currentUser.fullName)"/>'s profile.
53 </div>
54 ...
25         <cell sclass="row-title">Birthday :</cell>
26         <cell>
27             <datebox value="@bind(fx.birthday)" width="200px"
28                 constraint="no future" />
29         </cell>
30     </row>
```

- Line 2, 3: Define a form binding at form attribute and give the middle object's id fx. Specify @load(vm.currentUser) makes the binder load currentUser's properties to the middle object and @save(vm.currentUser, before='save') makes the binder save middle object's data back

to `vm.currentUser` before executing the command `save`.

- Line 8, 13, 20, 27, 36, 45: We should bind attributes to middle object's properties to avoid altering ViewModel's properties.
- Line 51, 52, 53: The label bound to `vm.currentUser.fullName` is not affected when `fx` is changed.

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

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

### Unsaved Input Doesn't Change Data

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



# Iterate a Collection

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

chapter5/index.zul

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

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

## ZK Component Reference

<https://www.zkoss.org/wiki/ZK%20Component%20Reference>



