From Technologies to Solutions

ZK Developer's Guide

Developing responsive user interfaces for web applications using AJAX, XUL, and the open-source ZK rich web client development framework

Markus Stäuble Hans-Jürgen Schumacher

PACKT PUBLISHING

ZK Developer's Guide

Developing responsive user interfaces for web applications using AJAX, XUL, and the open-source ZK rich web client development framework

Markus Stäuble

Hans-Jürgen Schumacher

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

ZK Developer's Guide

First published: March 2008

Production Reference: 1170308

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-847192-00-4

www.packtpub.com

Cover Image by Vinayak Chittar (vinayak.chittar@gmail.com)

Credits

Authors

Markus Stäuble

Hans-Jürgen Schumacher

Reviewers

Razvan Remus Popovici

Christianto Sahat Kurniawan Hutasoit

Development Editor

Rashmi Phadnis

Technical Editors

Bhupali Khule

Shailendra Shetye

Editorial Team Leader

Mithil Kulkarni

Project Manager

Abhijeet Deobhakta

Indexer

Monica Ajmera

Proofreader

Chris Smith

Production Coordinator

Aparna Bhagat

Cover Designer

Aparna Bhagat

About the Authors

Markus Stäuble is currently working as Senior Software Engineer. He has a Master's degree in Computer Science. He started with Java in the year 1999, since when he has gained much experience in building enterprise Java systems, especially web applications. He has a deep knowledge of the Java platform and the tools and frameworks around Java.

There are several people who have supported the writing of my first book. But there is especially one person to whom I want to say thank you, my wife Maria Elena. She supported the writing very much and gave me the power and energy to finish that work.

Hans-Jürgen Schumacher studied mathematics at the University of Karlsruhe, Germany. For 17 years he has been working as a Software Developer and Architect. Right now he has the position of a Senior Architect for J2EE. One of his special fields are GUIs for web applications as well as Improvements in the Software Build process.

I would like to thank my son Samuel, who was just born at the right time and gave me the power to finish this book.

About the Reviewers

Razvan Remus Popovici owns a BS degree in Computer Science from "Transylvania" University of Brasov, Romania in 1999. His software development experience consists of more than 10 years in application and database design, client-server or multi-tier application development in various domains such as networking, communications, accounting, statistics, management, or bioinformatics. His entrepreneurial background consists in a start-up with an accounting software company in Romania in 1996 (which is still selling!) and experience as an independent contractor in Germany.

Currently employed full time at Wayne State University in Detroit, Razvan is involved in architecture, design, and development of OntoTools, a software application for statistics analysis of microarray experiments used by genetics researchers.

Razvan owns Software Ingenieurbuero Popovici in Germany; the company is a sub-contractor for development of a product that enables enterprises to implement ITIL (Information Technology Infrastructure Library) concepts.

Previously, Razvan worked for many companies, such as Siemens, Nokia Siemens Netwoks, matrix42 AG, HLP, Exody GmbH, ROUTE66 BV, and RCS SA.

I would like to thank my wife Mihaela for her support while reviewing this book, and my parents.

Christianto Sahat Kurniawan Hutasoit is an independent Java developer. He has been playing with Java and JEE since 2001, and is already working for different Java projects in Indonesia, Germany, and Singapore. He can be reached at `csahat@gmail.com`.

For my mum, S. Resmiana Limbong. Thanks for your struggle.

Table of Contents

Preface

ZK is an open-source web development framework that enables web applications to have the rich user experiences and low development costs that desktop applications have had for years. ZK includes an AJAX-based event-driven engine, rich sets of XML User Interface Language (XUL), and XHTML components, and a markup language (ZUML).The ZK rich client framework takes the so-called server-centric approach: the content synchronization of components and the event pipelining between clients and servers are automatically done by the engine and AJAX plumbing codes are completely transparent to web application developers. Therefore, the end users get rich user interfaces with similar engaged interactivity and responsiveness to that of desktop applications, while for programmers, development remains similar in simplicity to that of desktop applications

What This Book Covers

In *Chapter 1* we give an introduction to, and take a look behind the ZK framework. In the last section of this chapter, we show some important issues from the ZK User Interface Language (ZUML).

In *Chapter 2* we will implement a CRUD (Create—Read—Update—Delete) application. We will also design and implement the pages with the ZK framework.

At the beginning of *Chapter 3*, we start with a simple CRUD application. The first thing we do here is to add some AJAX features to the application (live data). Here, we will learn many cornerstones provided by the ZK framework, and that we only have to implement some interfaces to use these features. We will move the application from a mixed code approach to a Model-View-Controller Architecture.

Chapter 4 deals with the end of the third phase in the development of a CRUD application with the AJAX ZK framework. We start with a simple application, and extend it step by step. The application now has many features that you will need in other applications as well.

Chapter 5 introduces the advantages and disadvantages of ZK. Then we will see how to integrate ZK with the Spring Framework and also why it is useful to do so. We will then move on to Hibernate and JasperReport.

In *Chapter 6* we will learn how to customize existing components. We have the ability to use styles that we know from HTML to change the layout of the components. Additionally we will see that it is possibile to build new components (macro components) on the basis of existing components. And, last but not least, in the last section of this chapter we will learn how to create a complete custom component that is based on a `.dsp` file.

Chapter 7 introduces zk-bench. Zk-bench is a very useful tool and it supports much more than just designing ZUL pages. It simplifies the development of web applications a lot and it's build around the ZK framework.

The *Appendix A* contains information about the configuration files, which are important in the context of a ZK application. You should use this appendix in conjunction with Chapter 1 to get a better understanding of how to configure your ZK application.

What You Need for This Book

The following is the list of software that you need to install and configure in order to start working with ZK:

- ZK 2.3.0
- Tomcat version 5.5.x (apache.tomcat.org)
- The demo application from ZK 2.3.0 is a simple WAR file. Just copy the WAR file into the webapps directory of the tomcat installation.
- Eclipse version 3.3

Who is This Book For?

This book is a Developer's Guide that steps you through the ZK framework with examples. It starts with installing and configuring ZK and takes you on to integrate it with other frameworks. By the time you are through the book you will be able to build an application on your own.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

There are three styles for code. Code words in text are shown as follows: "In the sample there is a class `InitSample` in a package `sample` used."

A block of code will be set as follows:

```
<zscript><![CDATA[
  import sample.InitSample;
  ]]>
</zscript>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be made bold:

```
if (event.getValue().length() > 0 && event.getValue().trim()
          .length() > 0)
        {
     buttonToSayHello.setVisible(true);
        }
        else
        {
          buttonToSayHello.setVisible(false);
        }
```

New terms and **important words** are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "When the user clicks on the button **Start the thread,** the thread starts doing the work asynchronously".

[Important notes appear in a box like this.]

[Tips and tricks appear like this.]

Reader Feedback

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer Support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the Example Code for the Book

Visit `http://www.packtpub.com/files/code/2004_Code.zip` to directly downlad the example code.

The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or code—we would be grateful if you would report this to us. By doing this you can save other readers from frustration, and help to improve subsequent versions of this book. If you find any errata, report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the let us know link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to the list of existing errata. The existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with some aspect of the book, and we will do our best to address it.

1
Getting Started with ZK

The world of web application development grows and grows. We can read many articles and books that explain how to develop a web application. Nearly every week a new framework emerges in the sphere of development. However, each new technology and framework means that we have to learn it. Without learning, it's not possible to leverage the complete power of the chosen technique. There are some reasons for the emergence of such a wide range of possibilities for developing a web application. The evolution of technology is one reason, and another is the demand for a faster and more efficient way to develop a web application. Each developer has his or her own preference for the way of development. However, not only issues in development demand new ways of building a web application. The big picture is about Web 2.0 and the underlying techniques **AJAX (Asynchronous JavaScript and XML)** also need new ways.

With ZK the developer gets a new alternative for solving the daily problems that occur in web projects and for building Web 2.0 applications. Through the chapters of this book, we show how to develop applications with ZK. After reading it and trying the framework there should be no need for more convincing.

Before we dive into the ZK framework it's important to clarify that this introductory chapter is not a replacement for the developers' guide from ZK (see `http://www.zkoss.org/doc/ZK-devguide.pdf`). The aim of this chapter is to give you an understanding of the basics and some important aspects of ZK, which are essential for daily development with the framework.

The following browsers are supported by ZK:

- Internet Explorer 6+/7
- Firefox 1+
- Safari 2+
- Mozilla 1+
- Opera 9+
- Camino 1+

Browsers without decent support for DOM and JavaScript are not supported. ZK is offered with a dual licensing model. On one side there is the GPL license, and on the other there is a commercial license with commercial support, which is offered from ZK (see `http://www.zkoss.org/license/`). Actually there are some IDEs that support the ZK framework. For example, there is ZeroKoder and the commercial zk-bench. These tools are presented in Chapter 7 of this book. To start programming with ZK you only need a basic knowledge of Java and XUL (XML User Interface Language). It's very easy to develop a front-end to a database. There are existing integrations of some OR-Mappers to ZK (e.g. for Hibernate). At present there are no external providers for controls. However, there is a defined interface (zk-forge) for creating controls.

The benefits of ZK are:

- AJAX-based Rich User Interfaces
- Event-driven Model
- ZK User-interface Markup Language
- Server-Centric Processing
- Script in Java and EL Expressions
- Modal Dialogs
- Simple Thread Model
- Live Data
- GPL

The disadvantage of frameworks like JSF, or the Echo framework is lack of a good IDE support.

What is ZK?

If somebody asks about ZK, a good answer is: *ZK is AJAX without JavaScript*. That's an answer for some merchandising prospects. The technical answer to the question is that ZK is a component-based framework, which is driven through events.

There are three cornerstones:

1. An AJAX-based event-driven engine
2. A rich set of XUL and XHTML components
3. ZUML (ZK User Interface Markup Language)

Before we go on, we should get familiar with a few buzzwords, which we mentioned before.

XHTML

XHTML is the abbreviation of *Extensible HyperText Markup Language*. XHTML is very similar to HTML. Briefly, XHTML is HTML written in XML syntax. The following points are the most important differences:

- The notation (uppercase/lowercase) of an element/attribute is important in XHTML (unlike of HTML).
- Elements without content (e.g. `<br>`) must be closed in XHTML (e.g. `<br />`).

Example:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="de" xml:lang="de">
  <head>
    <title>A XHTML example</title>
  </head>
  <body>
    <h1>Testpage</h1>
    <p>A paragraph</p>
    <p>another<br />paragraph
    </p>
  </body>
</html>
```

XUL

XUL is the abbreviation for *XML User Interface Markup Language*. This "language" is not a new invention from the ZK team. It was originally defined by the Mozilla team. The project page of XUL is `http://www.mozilla.org/projects/xul/`. The intention of Mozilla is to have a platform-independent language to define user interfaces. And therefore, a ZK developer can benefit from the big community around XUL, and can use XUL widgets.

Example:

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
<window
    id="findfile-window"
    title="Find Files"
    orient="horizontal"
    xmlns="http://www.mozilla.org/keymaster/gatekeeper
         /there.is.only.xul">
  <button id="find-button" label="Find"/>
  <button id="cancel-button" label="Cancel"/>
</window>
```

ZUML

ZUML is the abbreviation for *ZK User Interface Markup Language*. ZUML is based on XUL. There are some extensions to the XUL standard like the possibility of embedding program code.

```
<window title="Hello" border="normal">
  Hello World!
</window>
```

For the user, the application feels like a desktop application. For the developer, the programming is like a standalone application. ZK uses an event-driven paradigm, which encapsulates the request-response paradigm from a web application. These technical infrastructure things are done through the framework. Therefore, no JavaScript usually needs to be written for the browser side. Here, it's important to say that the framework automatically includes some JavaScript libraries, and generates the JavaScript code for the user. Therefore, the developer has nothing to do with JavaScript.

> It's important to know that ZK does only things that should be done from a UI Framework. This means that we do not have a new framework that solves all development problems in the world. Only the UI layer is addressed. Therefore, things like persistence must be done with other frameworks. However, there are many possibilities for integrating other frameworks (e.g. Hibernate, `http://www.hibernate.org`) into ZK.

First Step: Say Hello to ZK

To get a feel of working with ZK, it's wise to do a small example. To follow the long tradition of many programming technologies, we will start with the traditional "Hello World" example.

For the examples, we are using version 2.3.0 of the ZK framework.

The best way to start with ZK is to download the zkdemo application from the ZK team. The application can be downloaded from: `http://www.zkoss.org/download/`. The application comes in a ZIP archive. This archive contains four files:

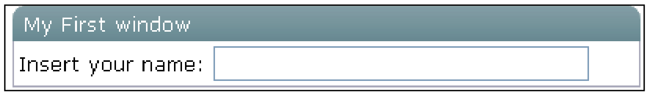
1. zkdemo-all.war
2. zkdemo.war
3. zkdemos-all.ear
4. zkdemos.ear

The WAR files are for deploying inside a J2EE web container. The EAR files are for deploying inside a J2EE application server. To start, just throw the `zkdemo.war` inside the webapps directory of Tomcat. After this the application is available with the URL `http://localhost:8080/zkdemo` (if Tomcat runs at port 8080). Now create a new file with the name `hello.zul`, and copy this page to the decompressed application zkdemo. The page can be accessed with `http://localhost:8080/zkdemo/hello.zul`.

```
<window title="Hello" border="normal">
   Hello World!
</window>
```

The downloaded archive of the demo contains `zkdemo.war`, `zkdemo-all.war`, `zkdemos-all.ear`, and `zkdemos.ear`. The files without the suffix `-all` does not contain a complete application. The libraries from ZK are missing. For a first start you can copy the `zkdemo-all.war` to your Tomcat webapps directory. If you want to use the `zkdemo.war` please copy all libraries from your zk-2.3.0 (directory `zk-2.3.0/dist/lib)` into the expanded directory of the web application. Without a copy of the libraries you get an error while executing your application.

After this small example, we will make a second example with an input element inside. This application consists of one main screen. This screen has a label (**Insert your name**), an input field, and a hidden button (**Say Hello**). The following screenshot shows the screen.

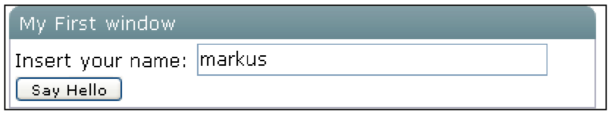


The definition of this screen is done in a single file. Its name is `helloworld.zul`. Both the layout and logic is done in that page. The page is realized with the previously mentioned ZK User Interface Markup Language.

Now it's time to draw the curtain and show the implementation of this simple application as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<zk xmlns="http://www.zkoss.org/2005/zul">
  <window title="My First window" border="normal" width="400px">
    <label value="Insert your name:" />
    <textbox width="60%" id="username">
    <!-- implement an event listener for the even onChanging -->
      <attribute name="onChanging"><![CDATA[
        //The eventcode is pure Java and therefore we can use the
              standard comment signs
        //Caution: We can not use the ampersand for the if-clause
               because we have to produce valid XML and
        //the ampersand is the indicator of an entity in XML
        /** Comments spanning
            more than one line
            are possible, too **/
        if (event.getValue().length() > 0 && event.getValue().trim()
          .length() > 0)
        {
          buttonToSayHello.setVisible(true);
        }
        else
        {
          buttonToSayHello.setVisible(false);
        }
        ]]>
      </attribute>
    </textbox>
    <!-- The alert is not a javascript alert, it’s
                                            a global function -->
    <button id="buttonToSayHello" label="Say Hello"  visible="false">
      <attribute name="onClick"><![CDATA[
        alert('Hello '+username.value);
        ]]>
      </attribute>
    </button>
  </window>
</zk>
```

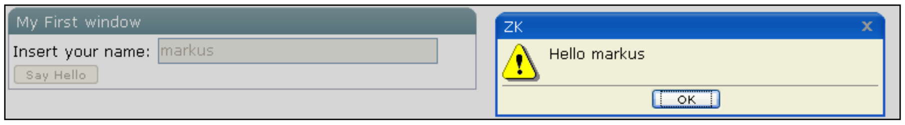
It was mentioned before that the page contains one hidden button (**Say Hello**). This button has the `buttonToSayHello` `id`. The idea behind this is that the button is only shown if there is some input in the textbox with the `username` `id`. After the input of some content (at least one character that is not a white space character) the output looks like the following screenshot.



One way to define this "event listener" is to define the `onChanging` attribute for the textbox `username`. The implementation of the event is done in Java.

> Each main element should have an `id` attribute. With the `id` of an element it's easy to access such an element. In the example, we set the button visible with `buttonToSayHello.setVisible(true)`.

If we click on the **Say Hello** button a window with a message is opened (see the following screenshot).

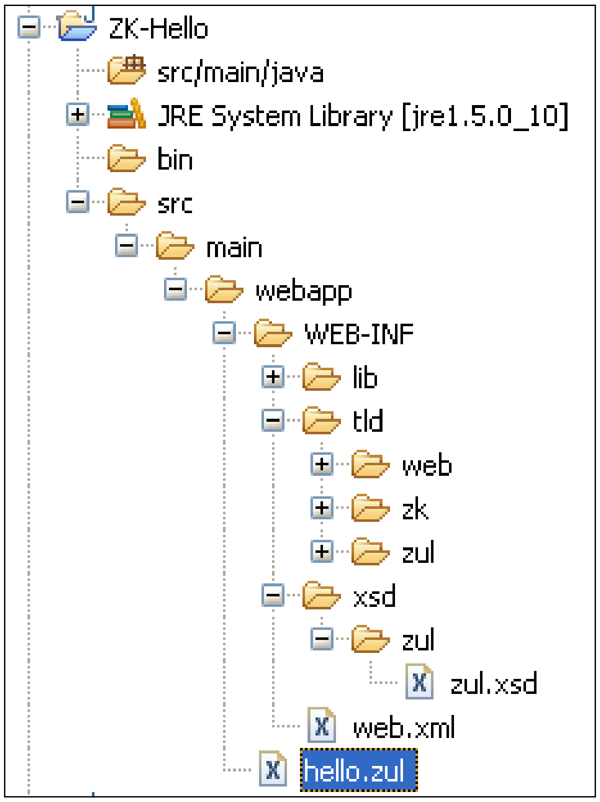


To show this messagebox the `onClick` attribute of the button is defined as `alert(&quot;Hello &quot;+username.value)`. We used the globally defined method `alert` to show the message. The parameter for the method is a text string. We first have a constant **Hello** text, and then the value of the username (`username.value`) textbox.

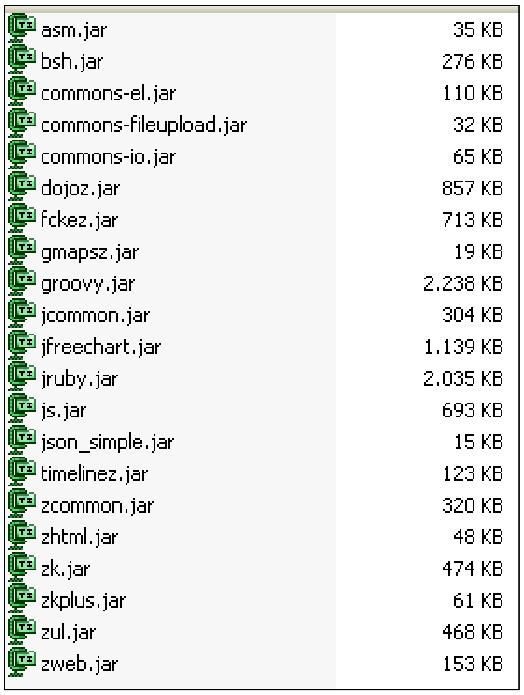
> Always in development, it's important to remember that a ZUL file is just an XML file. Also if we want to use Java in that file, we have to keep in mind that some characters could conflict with XML. Therefore, we have to use the XML entities for special characters. In the example, we used `&quot;` for `'` and `&amp;` for the `&` sign.

It's notable that the `alert` method is not JavaScript, it's Java. The code within a ZUL page is interpreted inside the BeanShell interpreter (`http://www.beanshell.org`). To use JavaScript for the `onClick` event handler we have to explicitly prefix the command with `javascript:`. The difference between a JSP and ZUL page is that the code inside a ZK page is run inside the BeanShell, whereas a JSP is compiled to a Java class.

The ZK framework itself runs in a JEE (Java Enterprise Edition) Web container. And to fulfill the requirements to run into such a container, we have to do (unfortunately) more than just creating such a page. The succeeding screenshot shows the project layout for the sample project.



One part of the project is the ZK framework itself (which can be downloaded under `http://www.zkoss.org`). Here, we have to copy the libraries, **tld** and **xsd** files to our project. The following libraries (inside the lib directory) are necessary to run the application:



asm.jar	35 KB
bsh.jar	276 KB
commons-el.jar	110 KB
commons-fileupload.jar	32 KB
commons-io.jar	65 KB
dojoz.jar	857 KB
fckez.jar	713 KB
gmapsz.jar	19 KB
groovy.jar	2.238 KB
jcommon.jar	304 KB
jfreechart.jar	1.139 KB
jruby.jar	2.035 KB
js.jar	693 KB
json_simple.jar	15 KB
timelinez.jar	123 KB
zcommon.jar	320 KB
zhtml.jar	48 KB
zk.jar	474 KB
zkplus.jar	61 KB
zul.jar	468 KB
zweb.jar	153 KB

The last artifact that we have to create is the deployment descriptor for the web application, `web.xml`. The descriptor for the demo application is depicted below. The `.zul` files are interpreted by a servlet defined inside the `zk.jar`. The ZK framework also provides session management. Therefore, a predefined `web.xml` file is provided.

```
<?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 Demo]]></description>
  <display-name>zk-hello</display-name>
  <!--
    <icon>
      <small-icon></small-icon>
      <large-icon></large-icon>
    </icon>
  -->
```

```
<!-- //// -->
<!-- ZK -->
<listener>
  <description>
     Used to cleanup when a session is destroyed
  </description>
  <display-name>ZK Session Cleaner</display-name>
  <listener-class>
    org.zkoss.zk.ui.http.HttpSessionListener
  </listener-class>
</listener>
<servlet>
  <description>ZK loader for ZUML pages</description>
  <servlet-name>zkLoader</servlet-name>
  <servlet-class>
    org.zkoss.zk.ui.http.DHtmlLayoutServlet
  </servlet-class>
  <!-- Required. Specifies URI of the update engine
      (DHtmlUpdateServlet).It must be the same as <url-pattern>
      for the update engine.-->
  <init-param>
    <param-name>update-uri</param-name>
    <param-value>/zkau</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>zkLoader</servlet-name>
  <url-pattern>*.zul</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>zkLoader</servlet-name>
  <url-pattern>*.zhtml</url-pattern>
</servlet-mapping>
<!-- Optional. Uncomment it if you want to use richlets.-->
<servlet-mapping>
  <servlet-name>zkLoader</servlet-name>
  <url-pattern>/zk/*</url-pattern>
</servlet-mapping>
<servlet>
  <description>The asynchronous update engine for ZK</description>
  <servlet-name>auEngine</servlet-name>
  <servlet-class>
    org.zkoss.zk.au.http.DHtmlUpdateServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>auEngine</servlet-name>
  <url-pattern>/zkau/*</url-pattern>
```

```
    </servlet-mapping>
    <!-- /////////// -->
    <!-- Miscellaneous -->
    <session-config>
      <session-timeout>120</session-timeout>
    </session-config>
    <welcome-file-list>
      <welcome-file>index.zul</welcome-file>
      <welcome-file>index.zhtml</welcome-file>
      <welcome-file>index.html</welcome-file>
      <welcome-file>index.htm</welcome-file>
    </welcome-file-list>
  </web-app>
```

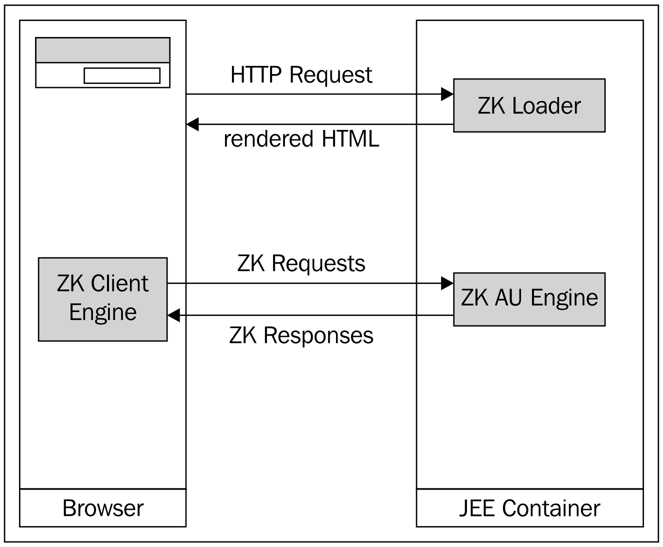
Now the application is ready to run in a JEE Web container, such as Apache Tomcat (`http://tomcat.apache.org`).

Inside ZK—How ZK Works

Before we start building a sample application, it's time to explain the internals of ZK. When building a simple application, or using a good tool, it is not important to understand its architecture. However, in case of an exception (Bad news: There are exceptions in programming with ZK, too.) it's better to know where you can look for the error.There are three components in the ZK architecture that can be identified as the main elements of the framework.

ZK Loader	This component is responsible for the loading and interpretation of a ZK page. The selection of the page is done based on the incoming URL Request from the client browser. The result is rendered as an HTML page and send back to the browser.
ZK Client Engine	This component sends "ZK Requests" to the server, and gets "ZK Responses" from the server. With these responses the DOM (Document Object Model) tree on the client side is updated. Therefore, we could call that the client part of the AJAX.
ZK AU (Asynchronous Update) Engine	The ZK AU Engine is the server part of the AJAX.

The following Insert figure gives a short overview to the architecture of ZK.



The sending of **ZK Requests** and **ZK Responses**, and the resulting updates constitute the mentioned event-driven programming model.

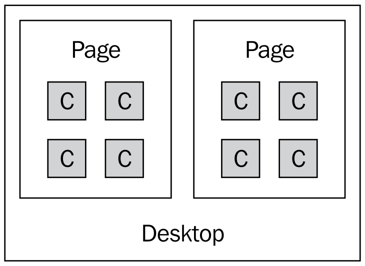
> One of the first things every Java programmer gets to know from his or her new programming heaven is that there is no need to care about memory, because the JVM (Java Virtual Machine) does the job for the developer. However, after that good message the next thing a Java programmer has to learn about is Garbage Collection. The ZK framework has no method for destroying a component. Since a page is detached from a page, the framework doesn't have any control over that component. After the detachment, the JVM takes care of the memory that was occupied by the component.

The Three Relatives—Desktop, Page, and Component

On the user side there are only HTML pages. In ZK these pages are build on desktops, pages, and components.

Component	Framework representation: `org.zkoss.zk.ui.Component` A component is a UI object (for example: button, textbox, or label). The component besides being a Java object on the server has a visual representation in the browser. This is created at the moment of attachment to a page. At the moment of detachment the visual part is removed. A component could have children. A component without any parent is called a **root component**.
Page	Framework representation: `org.zkoss.zk.ui.Page` A page contains components. Therefore, a page is a container for components. If the ZK loader is interpreting a ZUML page, a `org.zkoss.zk.ui.Page` is created. **Note:** A page can have multiple root components.
Desktop	Framework representation: `org.zkoss.zk.ui.Desktop` It's possible for a ZUML page to include another ZUML page. This is because these pages are available under the same URL and they are grouped to a desktop. Therefore, a desktop is a container for the pages.

The following figure gives a small overview of how the three "relatives" belong together.



Identification in ZK

If we talk about identification in ZK, we have to handle three paradigms:

- Identifiers
- UUID
- ID Space

At the moment of creation of a component the **Identifier** for the component is created (each one has an identifier). It's possible to change the identifier at any time. To set the identifier for a component you can use the `id` attribute.

```
<button label="test" id="theIdOfTheButton" />
```

It's not really necessary to set an `id` for the component. If no `id` is set from the developer, the framework generates a unique ID.

In addition to the mentioned identifier the, `id` of a component, each component has another identifier, the **Universal Unique ID** (**UUID**). This identifier is created automatically at the creation phase of a component.

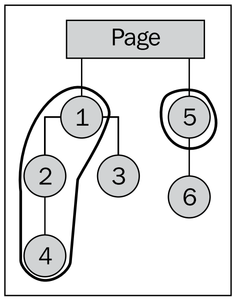
[In most cases, an application developer has no contact with the UUID.]

The identifier is used from the Client Engine to manipulate the Document Object Model at the client. For HTML components the `id`, and the UUID are the same. Therefore, if you change the `id` you change the UUID.

One attribute of an identifier should be that it is unique. If the project grows it's difficult to guarantee that an identifier is unique within the whole application. Therefore, in ZK we have the feature of **ID Spaces**. For example, an `org.zkoss.zul.Window` is an ID space. All identifiers in an ID Space must be unique instead of all identifiers in the whole application. Each component that implements the interface `org.zkoss.zk.ui.IdSpace` has its own ID Space. You could compare an ID Space with an XML Namespace.

For programmatic navigation inside and outside of the ID Spaces there is the helper class `org.zkoss.zk.ui.Path`.

A simple page hierarchy is shown in the figure below.



The class `org.zkoss.zk.ui.Path` offers two general possibilities to access a component. One is a static way, and the other is with an `org.zkoss.zk.ui.Path` instance.

The static way:

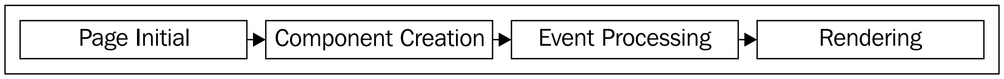
```
Path.getComponent("/5")
```

The way with an `ork.zkoss.ui.Path` instance:

```
new Path().getComponent("/1/2", "4")
```

Loading and Updating a ZK Page

The lifecycle of loading a ZK page consists of four phases. These phases are illustrated in the figure below.



Phase: Page Initial

In this phase, the page is initialized. To control this phase, it is possible to define a class or zscript for that phase.

Following is an example of defining a class for the `init` sequence:

```
<?init class="sample.InitSample" ?>
```

In the above example, the `InitSample` class from `sample` package is used. This class has to implement the `org.zkoss.zk.util.util.Initiator` interface.

The second way is to define a zscript for the `init` sequence:

```
<?init zscript="/init/init-sample.zs" ?>
```

Note: It's not necessary to define an init sequence.

Why to use an absolute class name in init?

In the description of the "Page Initial" phase the possibilities for the definition of the `init` sequence are shown. The use of `org.zkoss.zk.util.util.Initiator` interface to implement it is a good way. In most cases, the implementation class is placed in some package. To use the class you have to use the absolute name of the class.

```
<?init class="sample.InitSample" ?>
```

The reason for this is explained through the lifecycle. If you don't want to use an absolute class name in a Java program you have to import the class. In a ZUL file, you can do that with a zscript.

```
<zscript><![CDATA[
  import sample.InitSample;
  ]]>
</zscript>
```

However, if you now just use the name `InitSample` for the `init` sequence you will get a `java.lang.ClassNotFoundException`. To understand this, just add an output to the zscript.

```
<zscript><![CDATA[
  import sample.InitSample;
  System.out.println(InitSample.class);
  ]]>
</zscript>
```

Now we will implement `doInit()` and `doAfterCompose()`.

```
package sample;
import org.zkoss.zk.ui.Page;
import org.zkoss.zk.ui.util.Initiator;

public class InitSample implements Initiator
{
  public void doAfterCompose(Page page) throws
Exception
  {
    System.out.println("doAfterCompose");
  }
  public void doInit(Page page, Object[] args) throws
Exception
  {
    System.out.println("doInit");
  }
  public void doCatch(Throwable throwable)
  {
  }
  public void doFinally()
  {
  }
}
```

The result is the following output:

doInit

class sample.InitSample

doAfterCompose

The `init` sequence is called before the zscript, and therefore, your import has no consequence for that sequence. Instead of writing an init class you can do the necessary actions in a ZUL file (`init.zs`).

Phase: Component Creation

In this phase, the ZK loader is responsible for the interpretation of the pages. The components of each page are created and initialized.

The following steps are performed in this phase:

Composing Phase

Step 1: Before the creation and initialization is started, the optional values of `if` and `unless` attributes are evaluated (`<window if="expression" />` or `<window unless="expression" />`). With these attributes it's possible to control the layout depending on some conditions.

Step 2: The component is created. In the case of specification of the `use` attribute (`<window use="MyClass" />`) the specified class is used to create the component. Otherwise the component is created based on the name. After that the members are initialized.

Step 3: The nested elements are interpreted.

After the Composing Phase

Step 4: If the component implements the `org.zkoss.zk.ui.ext.AfterCompose` interface, `afterCompose()` is called.

Step 5: After the creation of all children the event `onCreate` is send to the created component.

If we use the `forEach` attribute the steps are done for each element in the collection.

```
<zscript><![CDATA[
  elements = new String[] {"ZK", "AJAX",
                          "FRAMEWORK"};
  ]]>
</zscript>
<listbox width="200px">
  <listitem label="${each}" forEach="${elements}"/>
</listbox>
```

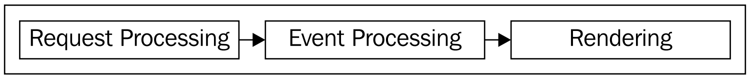
Phase: Event Processing

For each event that is queued, the event listeners are called in a thread. The processing of other events is not affected by the processing of code of a single event listener.

Phase: Rendering

The last stage is to compose from all components one single HTML page. For the rendering of a component `redraw()` is called.

The lifecycle of updating a ZK page consists of three phases. These phases are illustrated in the diagram below:



[It's worth mentioning that requests to the same desktop are processed sequentially, and requests to different desktops are processed in parallel. That's important if you are using frames, and each frame is a desktop.]

Phase: Request Processing

The ZK Asynchronous Update Engine updates the appropriate components.

Phase: Event Processing

This is the same as the event processing on creation of a page.

Phase: Rendering

ZK renders the affected components. The Client Engine updates the DOM tree on the browser on the basis of the ZK responses.

Events in ZK—Listening and Processing

In the introduction to ZK with the "Hello World" example, we actually used a way to register an event listener. We directly defined it as an attribute of a component in the page:

```
<button id="buttonToSayHello" label="Say Hello"
          onClick="alert(&quot;Hello &quot;+username.value)"
          visible="false"/>.
```

The next way to define an event listener is in the component class. Here, we have the first use of the `use` attribute of an element. When using the `use` attribute, the given class should implement the base class of the control.

```
<window use="sample.ExampleComponent" />
```

Now we can implement the event. In the example below, we implement the event `onOK`. If the event contains necessary information for the processing it's possible to give the method an argument `org.zkoss.zk.ui.event.KeyEvent`. However, the implementation of the method is correct without the argument, too.

```
package sample;
import org.zkoss.zk.ui.event.KeyEvent;
import org.zkoss.zul.Window;
public class ExampleComponent extends Window
{
  //It is possible to implement the method with an argument
  public void onOK(final KeyEvent event)
  {
  }
}
```

The `org.zkoss.zk.ui.Component` interface offers two methods: `addEventListener()` and `removeEventListener()`. With these methods it's possible to add and remove event listeners dynamically. The event listener must be from type `org.zkoss.zk.ui.event.EventListener`.

```
  package sample;
  import org.zkoss.zk.ui.event.Event;
  import org.zkoss.zk.ui.event.EventListener;
  public class SampleEventListener implements EventListener
  {
    public boolean isAsap()
    {
      return true;
    }
    public void onEvent(final Event event)
    {
      //here we process the necessary instructions
    }
  }
```

Now we can add this event listener in some initialization method.

```
  public void init(final Component component)
  {
    component.addEventListener("onClick", new SampleEventListener());
  }
```

When should it be done?—ASAP

The interface `org.zkoss.zk.ui.event.EventListener` has a method `isAsap` with the return value `boolean`.

If the method returns true, the event will be sent to the server the moment the events occurs. However, if the method returns false, the event won't be sent to the server directly. It will be sent if another event occurs that has a positive `isAsap`. In case of performance problems there is a possibility to directly affect that point.

Another way to handle events is to add an event listener to `org.zkoss.zk.ui.Page`. The consequence is that all events that belong to component from the page are send to that registered listener, too.

What is the Priority?—The Order of Event Processing

An interesting point in the processing of events is the sequence in which the events are processed. For the explanation, we assume that the `onSelect` event is sent and received.

First Step: All event listeners for the `onSelect` event for the targeting component are invoked that implement the marker interface `org.zkoss.zk.ui.event.Express`.

Second Step: Now the script for the `onSelect` attribute for the target component is executed. It's notable that if we don't have an event listener that implements `org.zkoss.zk.ui.event.Express` then the event listener in the first attribute is executed.

Third Step: Now all event listeners of the targeting component that do not implement the mentioned `org.zkoss.ui.event.Express` interface are executed.

Fourth Step: If the targeting component has an `onSelect` method it will be executed.

Fifth Step: All event listeners for the `onSelect` event that are registered to the parent page of the target component are executed.

[The only way to change the order of execution is with the help of the `org.zkoss.ui.event.Express` interface. If we have to register multiple event listeners for the same event on the same component that implements that interface then the event listener that is added first is the first to execute its event code. This interface only has a consequence if the event listener is registered on a component not on a page.]

[**Mark up your classes**

The `org.zkoss.zk.ui.event.Express` interface is a marker interface. The characteristic of such an interface is that it has an empty body. Such an interface is used only as a marker. It's useful to have such interfaces to emphasize that a class can have a special attribute.

The best known marker interface in the Java world is `java.io.Serializable`.]

[**How to send an event—Asynchronous or Synchronous ?**

It's possible to send events with the help of the `org.zkoss.zk.ui.event.Events` class. There are two methods: `postEvent` (the asynchronous way) and `sendEvent` (the synchronous way). With the `postEvent` method the event is placed at the end of the event queue. The method returns immediately. With the usage of the `sendEvent` method it can be specified that the event should be executed immediately. The method returns after the specified event is executed.]

How Can We Parallelize Event Listeners?

We mentioned before that requests to one desktop are done in a sequence. Therefore, if one event listener is processing its instructions then all other event listeners for that desktop are blocked. There is no problem if we only have short instructions, but in the case of long instructions we come across the problem that it's not acceptable to wait.

- To suspend working threads we have to use the `wait` method in `ork.zkoss.zk.ui.Executions`.
- We have to provide all necessary information to the working thread. This is because the working thread is not an event listener, and therefore, we have no direct access to the desktop.
- To proceed with the working we have to use the `notify` or `notifyAll` method of `org.zkoss.zk.ui.Executions`.
- To resume the event listener we use `org.zkoss.zul.Timer`.

For better illustration, we will provide a small example. First we have to implement a thread, which will do the work asynchronously. This thread is an implementation of the `java.lang.Thread`.

```
package sample.thread;
public class SampleWorkingThread extends Thread
{
  private static int counter;
  private Desktop desktop;
  private Label label;
  //Mutex variable for synchronization
  private final Object mutex = new Integer(0);
  /** Called by thread.zul to create a label asynchronously.
  * To create a label, it starts a thread, and waits for its
  completion.
  */
  public static final Label asyncCreate(final Desktop desktop) throws
     InterruptedException
  {
    //Create a new instance of the working thread
    final SampleWorkingThread worker = new
                           SampleWorkingThread(desktop);
    synchronized (worker.mutex)
    {
      worker.start();
      //Suspend the thread
      Executions.wait(worker.mutex);
      return worker.label;
    }
  }
  /**
  * Implementation of the standard run method from java.lang.Thread.
  */
  public void run()
  {
    ++this.counter;
    this.label = new Label("Execute, counter : " + counter);
    synchronized (this.mutex)
    {
      //Proceed working of threads which are suspend with mutex
               Executions.notify(this.desktop, this.mutex);
    }
  }
  /**
  * Constructor.
  *
  * @param desktop The reference to the acutal desktop instance.
  */
  public WorkingThread(final Desktop desktop)
  {
    this.desktop = desktop;
  }
}
```

Now we have to provide a ZUL page that starts the thread.

```
  <window id="main" title="A Sample for a working Thread">
    <button label="Start the thread">
      <attribute name="onClick">
        timer.start();
        Label label = sample.thread.SampleWorkingThread
                          .asyncCreate(desktop);
        main.appendChild(label);
        timer.stop()
      </attribute>
    </button>
    <timer id="timer" running="false" delay="1000" repeats="true"/>
  </window>
```

When the user clicks on the button **Start the thread,** the thread starts doing the work asynchronously.

Event Processing Thread—Initialize and Cleanup

One important thread in a ZK application is the event processing thread. In this thread an event listener is processed.

Sometimes there is a demand to initialize or clean up the thread. For this we have the `org.zkoss.zk.ui.event.EventThreadInit`-interface. This interface has the method `init(Component, Event)` which must be implemented (and the `prepare` method). To register the listener it's necessary to make an entry in `WEB-INF/zk.xml`.

```
<zk>
  <listener>
    <listener-class>sample.SampleEventThreadInit</listener-class>
  </listener>
</zk>
```

For cleanup there is the `org.zkoss.zk.ui.event.EventThreadCleanup` interface. Here, we have to implement the `cleanup(Component, Event, List)`method; additionally we also have to implement the `complete` method. The last parameter for the method cleanup is a list of exceptions that occur before the method is executed. To register the listener it's necessary to make an entry in `WEB-INF/zk.xml`.

```
<zk>
  <listener>
    <listener-class>sample.SampleEventThreadCleanup</listener-class>
  </listener>
</zk>
```

An example implementation of such a listener is shown in *Appendix A*.

Until now we have used ZUML, but we haven't really explained what ZUML is and how to use it. The next section should you give some idea about working with ZUML. Before we start, it's important to note that the next section is not a reference for ZUML, it's just a guide to help in the development of a ZUML page.

Event Types

The following event types exist in ZK:

- Mouse Events (e.g. `onClick`)
- Keystroke Events (e.g. `onOK`)
- Input Events (e.g. `onChange`)
- List and Tree Events (e.g. `onSelect`)
- Slider and Scroll Events (e.g. `onScroll`)
- Other Events (e.g. `onZIndex`)

It depends on the component whether an event is supported or not. It's noteworthy to say that an event is sent after the content of the component is updated.

ZUML – ZK User Interface Markup Language

The first thing a new ZK developer is confronted with is the ZUML (ZK User Interface Markup Language). It's noteworthy to say that this is not a new proprietary language; it's just an XML with namespace.

There are at least three namespaces that are important for working with ZK:

`http://www.zkoss.org/2005/zul`	The XUL component set
`http://www.w3.org/1999/xhtml`	The XHTML component set
`http://www.zkoss.org/2005/zk`	ZK-specific attributes

Inside a ZUML page, we can use for example: EL expressions, Java, JavaScript, Ruby, and Groovy.

Where is the session?

In some situations it could be faster to place some zscripts into the page, and for that we may want to access the session or some other object (for example the page itself).

In such situations there are some implicit objects that can be accessed in the pages. The objects are: `self`, `spaceOwner`, `page`, `desktop`, `session`, `componentScope`, `spaceScope`, `pageScope`, `desktopScope`, `sessionScope`, `applicationScope`, `requestScope`, `arg`, `each`, `forEachStatus`, and `event`.

You can use these objects in a zscript block, or in an element (`<button label="simple test" onClick="alert(self.label)" />`).

If you need information about the current execution you could use the `org.zkoss.zk.ui.Execution` interface. If you are in a component use `getDesktop().getExecution()`. When you don't have any component reference use the static method, `getCurrent()` from the class `org.zkoss.zk.ui.Executions`. After this introduction to ZUML, we present some important points in ZUML.

Are there some predefined Dialogs?

Each UI framework offers some predefined Dialogs, and ZK is no exception. We used the `org.zkoss.zul.Messagebox` in the "Hello World" example through the globally defined `alert` method.

Beyond the mentioned `Messagebox`, the next important predefined dialog is the `org.zkoss.zul.Fileupload`. With this dialog, it's possible to do an HTTP Upload.

Beyond the usage of the standard components such as window, textbox, and button it's possible to create components for a particular page. For that purpose there exists the `component` element.

With the `style` attribute of a component, it is possible to change the visual appearance of a component. You have the possibility to define the style each time.

For example:

```
<button label="Say Hello" style="border:2px red" />
```

However, if you want to use that button more than once, you can define a component.

For example:

```
<?component name="mybutton" extends="button" style="border:2px red"
            label="Say Hello" ?>
```

Now you can use this component as follows:

```
<mybutton />
<mybutton label="My label" />
```

It's possible to override the definitions of the predefined components:

```
<?component name="button" extends="button" style="border:2px red"
            label="Say Hello" ?>
<mybutton />
<mybutton label="My label" />
```

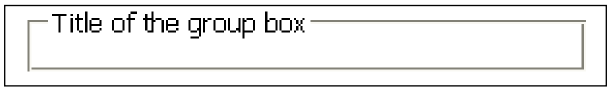
Molds

A mold is an attribute that is used to customize pages in ZK. The `org.zkoss.zk.ui.Component` interface has the `setMold` method. If there isn't a mold defined; the mold with the name `default` is used.

Some components come with more than the `default` mold. For example, `groupbox` (represented by the class `org.zkoss.zul.Groupbox`) has the `default` and `3d` molds.

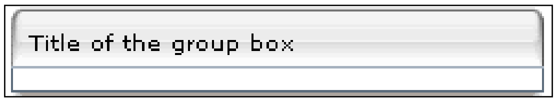
To use the `default` mold nothing special is necessary.

```
<groupbox open="true" width="250px">
   <caption label="Title of the group box"></caption>
</groupbox>
```



To use the `3d` mold we have to specify the `mold` attribute.

```
<groupbox mold="3d" open="true" width="250px">
  <caption label="Title of the group box"></caption>
</groupbox>
```



ZK Attributes

Beyond the normal attributes of elements (for example: `width`) there are some special attributes, which are called ZK Attributes.

`use`	With this attribute you can specify another class for the rendering of a concrete element instead of the default element. Example: `<window use="sample.SampleWindow" />`
`if`	The element is rendered only if the condition is true. Example: `<textbox if="${b==1}" />`
`unless`	The element is rendered only if the condition is false. Example: `<textbox unless="${c==1}" />`
`forEach`	This attribute is used in conjunction with collections. For each element in the collection, the element is rendered with the concrete value. Example: `<listitem label="${each}" forEach="${elements}" />`

`forEachBegin`	This is used in conjunction with the `forEach` attribute. Here, you could specify the index of the first element of the collection that should be used for `forEach`. Example: `<listitem label="${each}" forEach="${elements}" forEachBegin="1"/>`
`forEachEnd`	This is used in conjunction with the `forEach` attribute. Here, you could specify the index of the last element of the collection that should be used for `forEach`. Example: `<listitem label="${each}" forEach="${elements}" forEachEnd="1"/>`

ZK Elements

There are special elements that aren't responsible for creating components, the so called ZK Elements. The task of these elements is to control a ZUML page.

The following is a list of ZK Elements:

- **Element:** zk, `<zk>…</zk>`
- **Element:** zscript, `<zscript>…</zscript>`
- **Element:** attribute, `<attribute />`
- **Element:** custom-attributes, `<custom-attributes />`

The **`zk` element** is an element for grouping other components. Important at this juncture is that the `zk` element gets no member in the component tree.

Sample:

```
<window>
  <zk>
    <button id="buttonToSayHello" label="Say Hello" />
    <button id="buttonToSayHelloAgain" label="Say Hello Again" />
  </zk>
</window>
```

For layout and component tree, it's same as having no `zk` element.

A good question could be why to use the `zk` element. One important reason is that in a XML document there can be only one root element. And therefore, if we want to have more root elements (for example, more than one window) then we can use a `zk` element as root.

Another area of application is the possibility of iterating over components, or conditional evaluation. For that `zk` elements supports the following attributes:

- `forEach`
- `if`
- `unless`

A small example of usage is as follows:

```
<window>
  <zk forEach="${element}">
    <label value="${element.name}" />
      <button id="buttonToSayHello" label="Say Hello"
               if="${element.showSayHello}" />
      <button id="buttonNotToSayHello" label="Say not Hello"
               unless="${element.showSayHello}" />
  </zk>
</window>
```

The **zscript element** is used for embedding Java code into ZUML pages. At the time of page evaluation the code will be interpreted by the BeanShell (`http://www.beanshell.org`). It's possible to embed Java code directly between the opening and closing `zscript` tags. Another way is to use the `src` attribute of the `zscript` element. In this attribute, you could specify a URI to a file, which contains the Java code. The `zscript` element supports conditional evaluation with the help of the `if` and `unless` attributes.

Beyond zscript you could use **EL** (Expression Language) Expressions in your ZK pages.

The **`attribute` element** defines an attribute of the enclosing element. We have used this ZK Element in the "Hello World" example.

```
<textbox width="60%" id="username">
  <attribute name="onChanging">
  ...
  </attribute>
</textbox>
```

One area of application is event listeners. The `attribute` element supports conditional evaluation with `if` and `unless`, too.

The **`custom-attributes` element** is used to define an object in a special scope (for example, page scope).

An example for defining a `custom-attributes` element is as follows:

```
<window>
  <custom-attributes test="simple" />
</window>
```

This is the same as:

```
<window>
  <zscript>
    self.setAttribute("test", "simple");
  </zscript>
</window>
```

It's possible to use the `scope` attribute:

```
<window>
  <custom-attributes test="simple" scope="desktop" />
</window>
```

This is the same as:

```
<window>
  <zscript>
    desktop.setAttribute("test", "simple");
  </zscript>
</window>
```

The `custom-attributes` element supports conditional evaluation with `if` and `unless`, too.

Layout Techniques

If we talk about UI, we have to talk about the layout of the pages. There are some elements that help you in the layout of a ZK page.

Some important elements are:

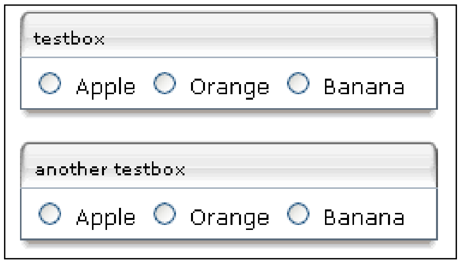
- `vbox`
- `hbox`
- `box`
- `grid` (in conjunction with: `columns`, `column`, `rows`, and `row`)

An example for using `div` and `vbox` elements is as follows:

```
<div align="center" style="vertical-align:center">
  <vbox>
```

```
        <groupbox mold="3d" open="true" width="250px">
          <caption label="testbox"></caption>
          <radiogroup  id="master">
            <radio label="Apple"/>
            <radio label="Orange"/>
            <radio label="Banana"/>
          </radiogroup>
        </groupbox>
        <groupbox mold="3d" open="true" width="250px">
          <caption label="another testbox"></caption>
          <radiogroup id="client">
            <radio label="Apple"/>
            <radio label="Orange"/>
            <radio label="Banana"/>
          </radiogroup>
        </groupbox>
      </vbox>
    </div>
```

The following screenshot shows the layout.



Prevent XHTML, where it is possible

There are some disadvantages in the usage of XHTML in ZK. We have only one identifier and not two (see the section *Identification in ZK*). Therefore, the `id` attribute has to be unique for the same desktop. There is no invalid XML element because ZK uses `org.zkoss.zhtml.Raw` for constructing any unrecognized XML element. The elements are case insensitive, and have no mold support (the attribute is ignored). Therefore, in most cases it's better to port an XHTML page to a ZK page.

Instead of using the layout control it's possible to do the layout with a XHTML, and using Cascading Stylesheets (see Chapter 6*: Creating Custom Components*).

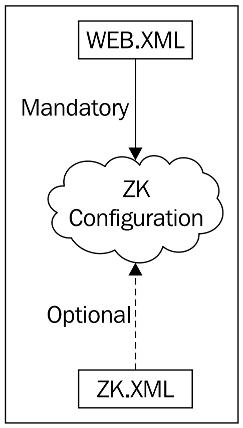
Separation of Concerns

One important paradigm in object-oriented programming is **Separation of Concerns (SoC)**. This paradigm says that an application should be broken into distinct features that overlap in functionality as little as possible (see `http://en.wikipedia.org/wiki/Separation_of_concerns`).

For large applications, it's not advisable to do the layout, and the logic together in one page. For such applications, you should follow the programming principle of separation of concerns. This separation is not only for architecture, but also for better maintenance of your application. This is because if you have your logic directly in Java classes, you can leverage the full power of your IDE, e.g. you can have a good debugger, and you can easily write unit tests.

Configuration and Deployment

The configuration for the ZK framework is separated in two files: `web.xml` for servlet and mapping definition, and `zk.xml`, which gives the possibility of overriding the default ZK system settings. The following figure shows that there are some mandatory settings, and some optional additional settings. The configuration in `web.xml` is mandatory for correct working of the ZK framework.



Configuration of web.xml

The most important place to configure a web application is the `web.xml` file. Without correct settings nothing will work. In the next paragraphs, we will have a detailed look at the settings that are necessary for the ZK framework.

First we need the `zkLoader` servlet, which loads the ZUML pages when the web container receives a request for a page.

This servlet is named `org.zkoss.zk.ui.http.DHtmlLayoutServlet`. Since this is the first entry for the ZK framework it is really necessary to load this first. Use the parameter:

```
<load-on-startup>1</load-on-startup>
```

The servlet has two init parameters:

- **Update-uri**

  This is a mandatory parameter. It specifies the URI the ZK AU engine is mapped to. The browser needs that URI to send the correct pattern of AJAX URL requests to the `DHtmlLayoutServlet`. For default, we use `/zkau`.
- **Log_level**

  This is an optional parameter. It specifies the default log level for the `org.zkoss package`. Possible values are OFF, ERROR, DEBUG, INFO, and WARNING.

This servlet supports static XHTML pages natively.

The complete settings are as follows:

```
<servlet>
  <description>ZK loader for ZUML pages</description>
  <servlet-name>zkLoader</servlet-name>
  <servlet-class>
    org.zkoss.zk.ui.http.DHtmlLayoutServlet
  </servlet-class>
  <init-param>
    <param-name>update-uri</param-name>
    <param-value>/zkau</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

The second important and mandatory servlet is `org.zkoss.zk.au.http.DHtmlUpdateServlet`. It handles AJAX requests asynchronously. The mapped URL pattern must be the same as for the `update-uri` parameter in the `DHtmlLayoutServlet`.

The last mandatory servlet is the ZK Session Cleaner. It is a listener and it cleans up the memory when a session is closed. The class is `org.zkoss.zk.ui.http.HttpSessionListener`.

The question is now what to do if any other technique like JSP or JSF should be integrated as well. ZK has a filter servlet, which post-process the output from other servlets processors.

The servlet class is `org.zkoss.zk.ui.http.DHtmlLayoutFilter` and has two optional parameters:

- **Extension**

  This describes how to process the output. The default is html.
- **Charset**

  This specifies the charset of the output. The default is UTF-8.

A sample part of `web.xml` looks as follows:

```
<listener>
  <description>
    Used to cleanup when a session is destroyed
  </description>
  <display-name>ZK Session Cleaner</display-name>
  <listener-class>
    org.zkoss.zk.ui.http.HttpSessionListener
  </listener-class>
</listener>
<servlet>
  <description>ZK loader for ZUML pages</description>
  <servlet-name>zkLoader</servlet-name>
  <servlet-class>
    org.zkoss.zk.ui.http.DHtmlLayoutServlet
  </servlet-class>
  <init-param>
    <param-name>update-uri</param-name>
    <param-value>/zkau</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>zkLoader</servlet-name>
  <url-pattern>*.zul</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>zkLoader</servlet-name>
  <url-pattern>*.zhtml</url-pattern>
</servlet-mapping>
<servlet>
  <description>The asynchronous update engine for ZK</description>
```

```
    <servlet-name>auEngine</servlet-name>
    <servlet-class>
      org.zkoss.zk.au.http.DHtmlUpdateServlet
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>auEngine</servlet-name>
    <url-pattern>/zkau/*</url-pattern>
  </servlet-mapping>
```

Configuration of zk.xml

The `zk.xml` file is an optional file, and stays in the `WEB-INF/` folder on the same level as `web.xml`. As the above overview has shown, there are many settings, which already have default values. The purpose of the `zk.xml` file is to extend, or overwrite existing settings for ZK application. A list of these settings is given in the *Appendix A*.

Deployment

The deployment via Ant or any other IDE integrated tools is rather simple. Just copy the ZK distribution under `WEB-INF/lib` and that's it. If you wish to optimize the distribution you may omit the libraries you didn't use like dojoz or timeline.

Deployment of ZK Applications with Maven

If you like to work with Maven (`http://maven.apache.org`) it takes a little bit more effort. This is because of the rapid rate of releases in ZK; sometimes the Maven repositories have a bit of a delay in publishing. The most famous Maven repositories include the ZK framework, i.e. `http://repo1.maven.org/maven2`. The structure of the ZK framework in a Maven repository looks like:

As the illustration shows the ZK framework within the Maven repository has three different groupIDs:

- `org.zkoss.common`
- `org.zkoss.zk`
- `org.zkoss.zkforge`

The artifacts zcommon and zweb are common. The second group contains the artifacts of the ZK framework itself, and the last contains the additional tools like FCKEZ. See the complete part of the `POM.XML` for the deployment of ZK in a Maven project:

```
<dependency>
  <groupId>org.zkoss.common</groupId>
  <artifactId>zcommon</artifactId>
  <version>2.3.0</version>
</dependency>
<dependency>
  <groupId>org.zkoss.common</groupId>
  <artifactId>zweb</artifactId>
  <version>2.3.0</version>
</dependency>
<dependency>
  <groupId>org.zkoss.zk</groupId>
  <artifactId>zk</artifactId>
  <version>2.3.0</version>
</dependency>
<dependency>
  <groupId>org.zkoss.zk</groupId>
  <artifactId>zul</artifactId>
  <version>2.3.0</version>
</dependency>
<dependency>
  <groupId>org.zkoss.zk</groupId>
  <artifactId>zhtml</artifactId>
  <version>2.3.0</version>
</dependency>
<dependency>
  <groupId>org.zkoss.zk</groupId>
  <artifactId>zkplus</artifactId>
  <version>2.3.0</version>
</dependency>
<dependency>
  <groupId>org.zkoss.zkforge</groupId>
  <artifactId>gmapsz</artifactId>
  <version>2.0-2</version>
```

```
</dependency>
<dependency>
  <groupId>org.zkoss.zkforge</groupId>
  <artifactId>fckez</artifactId>
  <version>2.3-2</version>
</dependency>
<dependency>
  <groupId>org.zkoss.zkforge</groupId>
  <artifactId>dojoz</artifactId>
  <version>0.2.2-2</version>
</dependency>
```

Summary

In this chapter, we give an introduction into, and behind the ZK framework. The example illustrated in the section *First Step – Say Hello to ZK* gave us a feeling of how a simple ZK application is created. It's a good point to start your own small ZK projects or do some rapid prototyping with ZK.

The section "*Inside ZK – How ZK works*" is not really necessary if you just want to do small projects with ZK. However, if you think about major projects it's wise to know a little bit about what is inside ZK, and how ZK works.

The last section of this chapter, we show some important issues from the ZK User Interface Language (ZUML). Here, it's not possible to show you the complete ZUML. We just want to pick some points to show you how these important cornerstones are working.

However, now we have talked enough about inside ZK. In the following chapter, the main focus will be on using the ZK framework, and get "infected" by the ZK framework.

2
Online Media Library

The first chapter shows the theory and deployment of the ZK framework. Now it's time to do the things that we are here for: implementing applications with ZK. The next three chapters will show the detailed implementation of a small, but rich application. It's not really possible to use all components in this application, but the application at the end of Chapter 4 has many things that you will need in your first project with the ZK framework.

With the help of this chapter, you should be able to build your own applications based on the ZK framework. In this chapter, we only build the cornerstone of the Online Media library, and improve it step by step in the next chapters.

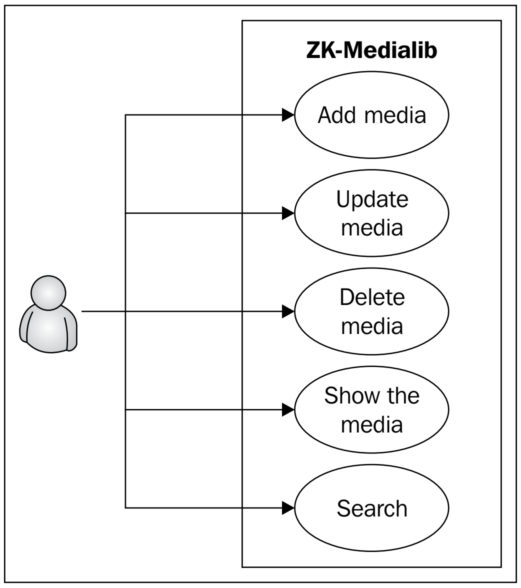
An Online Media Library

There are some traditional applications that could be used to introduce a framework. One condition for the selection is that the application should be a **CRUD (Create—Read—Update—Delete)** application. Therefore, an 'Online Media Library', which has all four operations, would be appropriate. We start with the description of requirements, which is the beginning of most IT projects.

The application will have the following features:

- Add new media
- Update existing media
- Delete media
- Search for the media (and show the results)
- User roles (administrator for maintaining the media and user accounts for browsing the media)

In the first implementation round the application should only have some basic functionality that will be extended step by step.



A media item should have the following attributes:

- A title
- A type (Song or Movie)
- An ID which could be defined by the user
- A description
- An image

The most important thing at the start of a project is to name it. We will call our project **ZK-Medialib**.

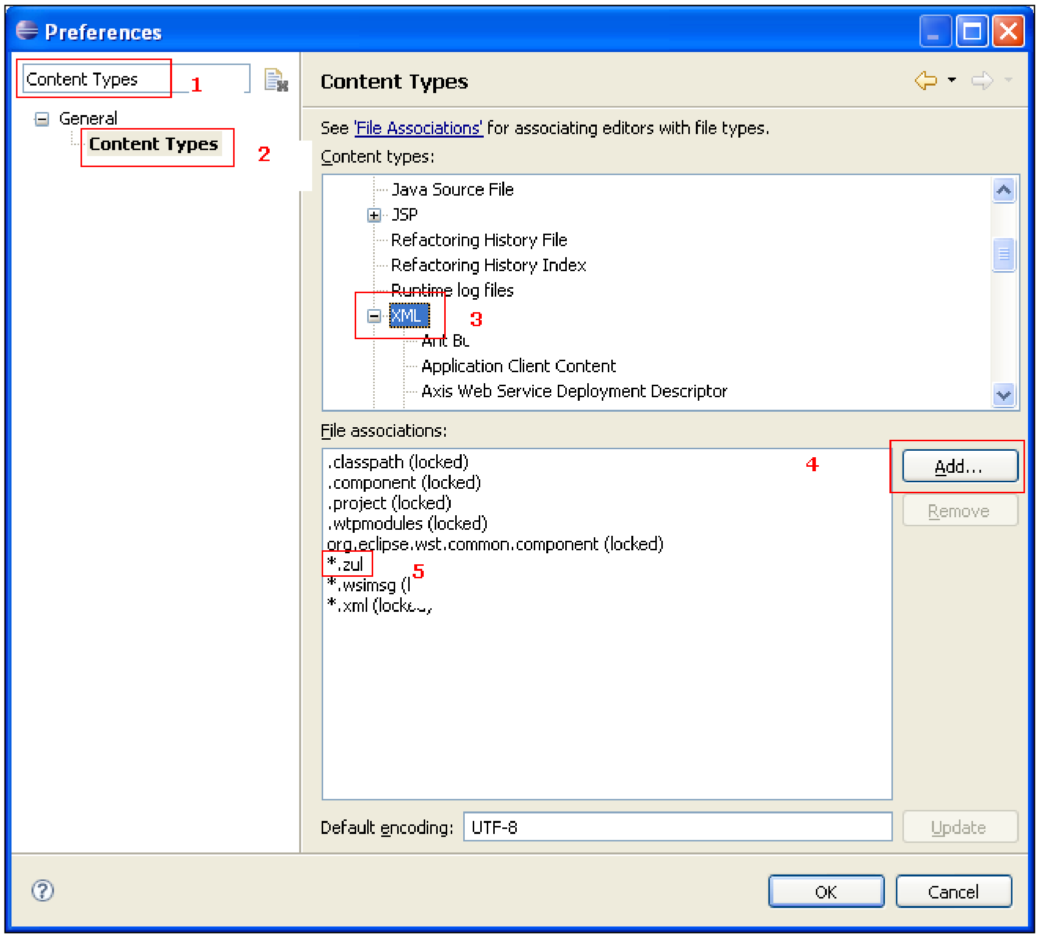
Setting up Eclipse to Develop with ZK

We use version 3.3 of Eclipse, which is also known as *Europa* release. You can download the IDE from `http://www.eclipse.org/downloads/`. We recommend using the version "Eclipse IDE for Java EE Developers".

First we have to make a file association for the `.zul` files. For that open the **Preferences** dialog with **Window | Preferences**. After that do the following steps:

1. Type **Content Types** into the search dialog.
2. Select **Content Types** in the tree.
3. Select **XML** in the tree.
4. Click **Add** and type ***.zul**.
5. See the result.

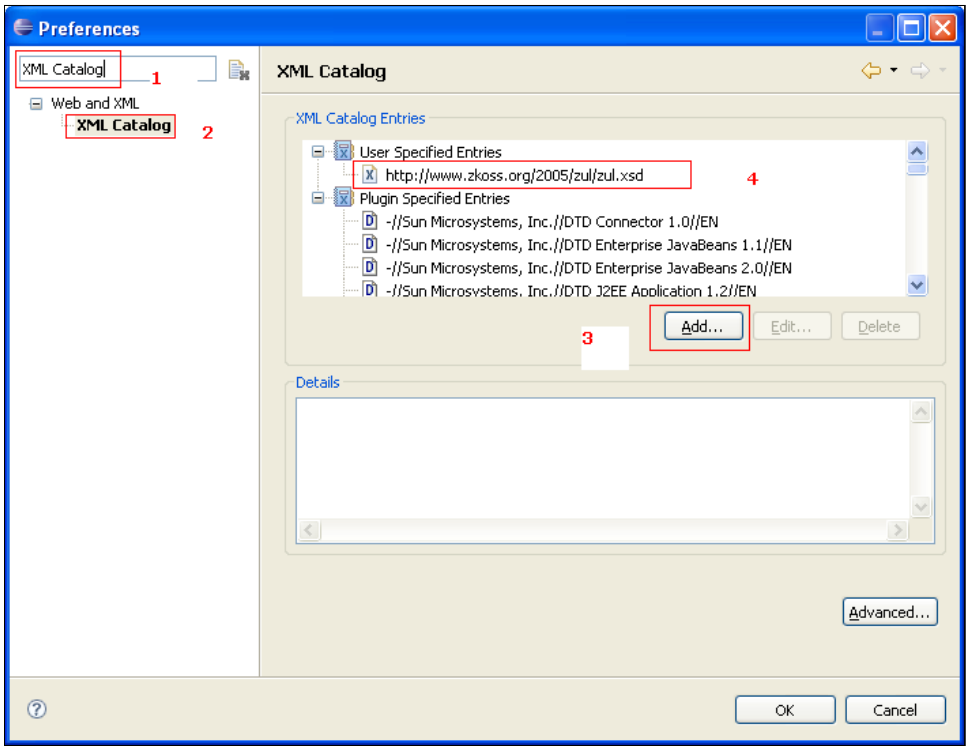
The steps are illustrated in the picture below:

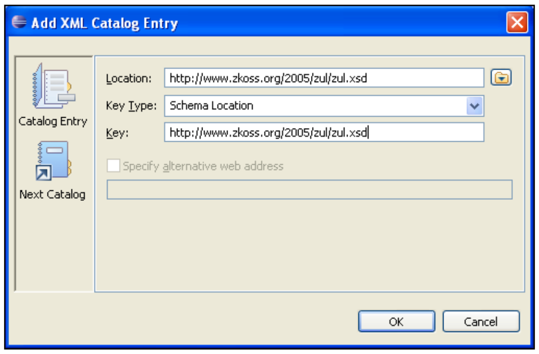


With these steps, we have syntax highlighting of our files. However, to have content assist, we have to take care about the creation of new files. The easiest way is to set up Eclipse to work with `zul.xsd`.

For that open the **Preferences** dialog with **Window | Preferences**. After that do the following steps:

1. Type **XML Catalog** into the search dialog.
2. Select **XML Catalog** in the tree.
3. Press **Add** and fill out the dialog (see the second dialog below).
4. See the result.

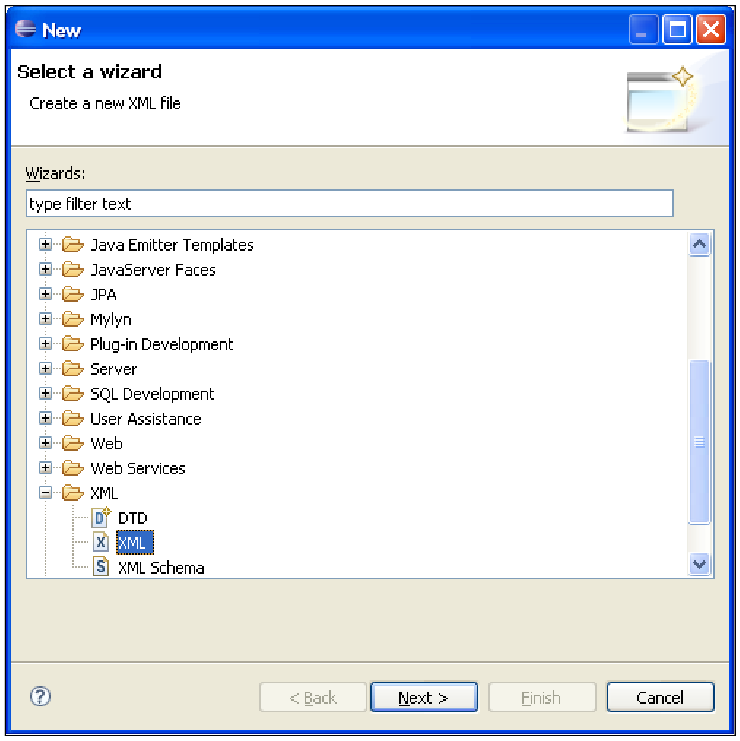

Preferences
XML Catalog
1
Web and XML
XML Catalog
2
XML Catalog
XML Catalog Entries
User Specified Entries
http://www.zkoss.org/2005/zul/zul.xsd
4
Plugin Specified Entries
-//Sun Microsystems, Inc.//DTD Connector 1.0//EN
-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN
-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN
-//Sun Microsystems, Inc.//DTD J2EE Application 1.2//EN
3
Add...
Edit...
Delete
Details
Advanced...
OK
Cancel



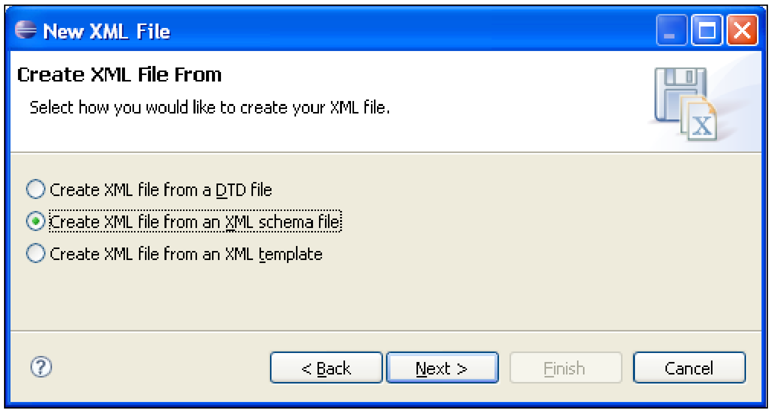
Location: http://www.zkoss.org/2005/zul/zul.xsd
Key Type: Schema Location
Key: http://www.zkoss.org/2005/zul/zul.xsd

Now we can easily create new ZUL files with the following steps:

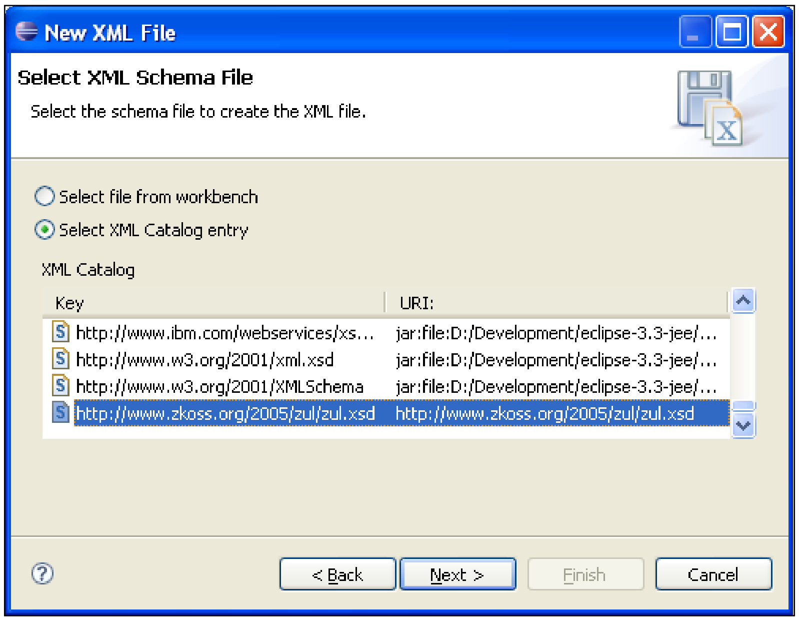
1. **File** | **New** | **Other**, and select **XML**:



2. Type in the name of the file (for example `hello.zul`).
3. Press **Next**.
4. Choose **Create XML file from an XML schema file**:


New XML File
Create XML File From
Select how you would like to create your XML file.
Create XML file from a DTD file
Create XML file from an XML schema file
Create XML file from an XML template
< Back
Next >
Finish
Cancel

5. Press **Next**.
6. Select **Select XML Catalog entry**.
7. Now select **zul.xsd**:

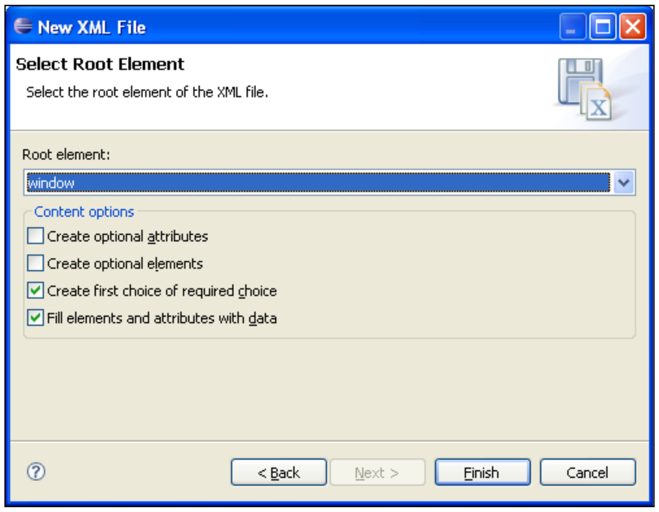


Select file from workbench [✓ Select XML Catalog entry]

XML Catalog

Key	URI:
http://www.ibm.com/webservices/xs...	jar:file:D:/Development/eclipse-3.3-jee/...
http://www.w3.org/2001/xml.xsd	jar:file:D:/Development/eclipse-3.3-jee/...
http://www.w3.org/2001/XMLSchema	jar:file:D:/Development/eclipse-3.3-jee/...
http://www.zkoss.org/2005/zul/zul.xsd	http://www.zkoss.org/2005/zul/zul.xsd

8. Now select the **Root Element** of the page (e.g. window).

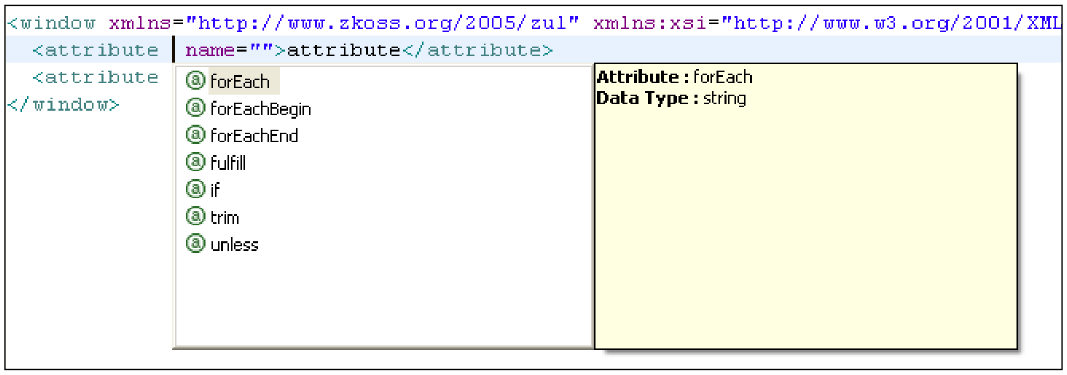


Root element: window

Content options
- ☐ Create optional attributes
- ☐ Create optional elements
- ✓ Create first choice of required choice
- ✓ Fill elements and attributes with data

9. Select **Finish**.

Now you have a new ZUL file with content assist. Go into the generated attribute element and press *Alt+Space*.

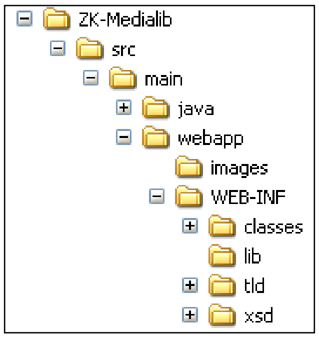


```
<window xmlns="http://www.zkoss.org/2005/zul" xmlns:xsi="http://www.w3.org/2001/XML
  <attribute | name="">attribute</attribute>
  <attribute
</window>
```

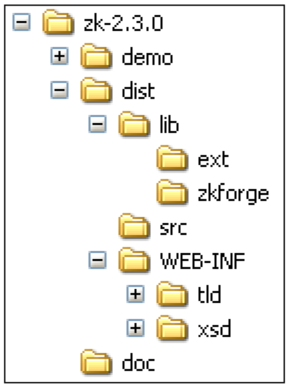
Attribute : forEach
Data Type : string

Setting up a New Project

The first thing we will need for the project is the framework itself. You can download the ZK framework from `http://www.zkoss.org`. At the time of writing, the latest version of ZK is 2.3.0. Therefore it's recommended to use that version for the examples that are provided in this book. After downloading and unzipping the ZK framework we should define a project structure. A good structure for the project is the directory layout from the **Maven project** (`http://maven.apache.org/`). The structure is shown in the figure below.



The directory **lib** contains the libraries of the ZK framework. For the first time it's wise to copy all JAR files from the ZK framework distribution. If you unzip the distribution of the version 2.3.0 the structure should look like the figure below. The structure below shows the structure of the ZK distribution. Here you can get the files you need for your own application.



```
zk-2.3.0
  demo
  dist
    lib
      ext
      zkforge
    src
    WEB-INF
      tld
      xsd
  doc
```

For our example, you should copy all JAR files from **lib**, **ext**, and **zkforge** to the **WEB-INF/lib** directory of your application. It's important that the libraries from **ext** and **zkforge** are copied direct to **WEB-INF/lib**. Additionally copy the directories **tld** and **xsd** to the **WEB-INF** directory of your application.

Now after the copy process, we have to create the deployment descriptor (`web.xml`) for the web application. Here you can use `web.xml` from the demo application, which is provided from the ZK framework. For our first steps, we need no `zk.xml` (that configuration file is optional in a ZK application).

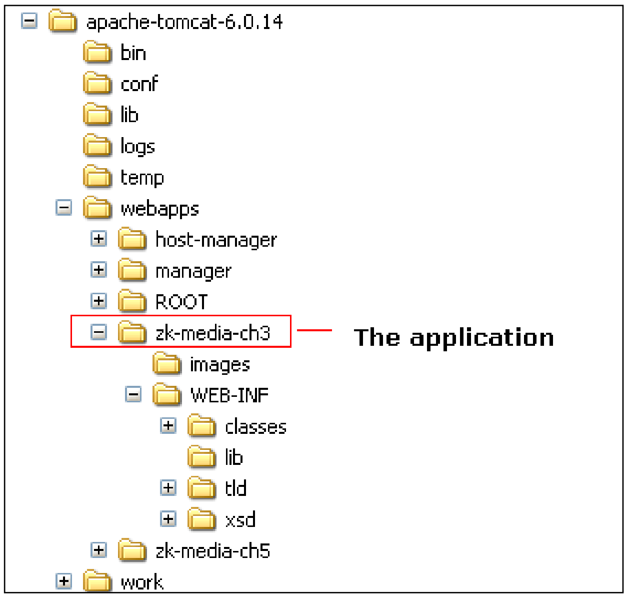
The application itself must be run inside a **JEE** (Java Enterprise Edition) **Webcontainer**. For our example, we used the Tomcat container from the Apache project (`http://tomcat.apache.org`). However, you can run the application in each JEE container that follows the Java Servlet Specification 2.4 (or higher) and runs under a Java Virtual Machine 1.4 (or higher). We create the `zk-media.xml` file for Tomcat, which is placed in `conf/Catalina/localhost` of the Tomcat directory.

```
<Context path="/zk-media" docBase="D:/Development/workspaces/
  workspace-zk-medialib/ZK-Medialib/src/main/webapp" debug="0"
  privileged="true" reloadable="true" crossContext="false">
  <Logger className="org.apache.catalina.logger.FileLogger"
    directory="D:/Development/workspaces/workspace-zk-medialib/logs/
    ZK-Medialib" prefix="zkmedia-" suffix=".txt" timestamp="true"/>
</Context>
```

With the help of this context file, we can directly see the changes of our development, since, we set the root of the web application to the development directory.

Deployment of the demo application

The folder structure of the demo application is the Maven structure (`http://maven.apache.org/`). To deploy the application, create a new subdirectory (e.g. `zk-media-lib`) in the **webapps** directory of Tomcat and copy everything from the webapp directory of the application to this new subdirectory. Don't forget to copy the libraries, too. After that, you can start your Tomcat and run the application with `http://localhost:8080/zk-media-lib` (if your Tomcat is running on port 8080).



Creating the Pages

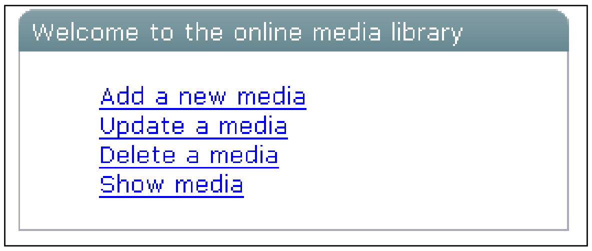
After the implementation of the model classes (which is not in the scope of this book) it's time to start with implementing the ZUL pages. The first page is `index.zul` which should be a simple navigation page for the individual pages. A simple way to execute the individual pages is to offer links for each page on the starting page. In a simple HTML page, we would use a `href` tag. In a ZUL page the preferred way is to use `toolbarbutton`. With the following line we add a link to `add-media.zul`:

```
<toolbarbutton label="Add a new media" href="add-media.zul"/>
```

We need four navigation points, and therefore, four pages to fulfill the requirements of the CRUD feature. These four pages are:

- Add a new media item: `add-media.zul`
- Update an existing media item: `update-media.zul`
- Delete an existing media item: `delete-media.zul`
- Show media: `show-media.zul`

The navigation page is depicted in the following figure:

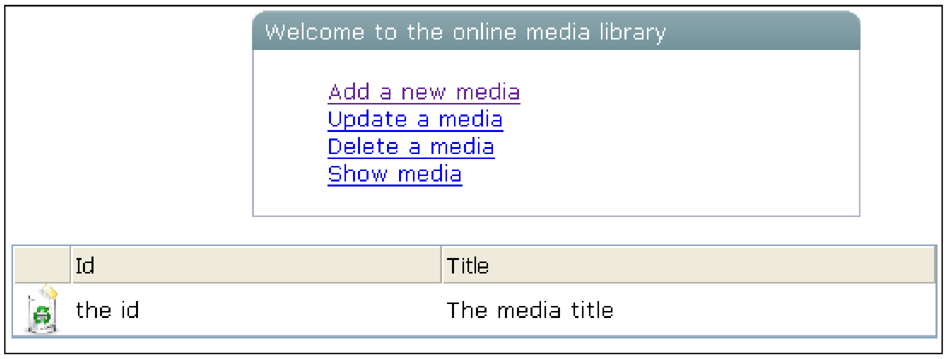


To have a border with a title, the `window` component of ZK is used. With the help of the `div` component, the `window` component is centered. The next problem we have to solve is that each link should appear on a new line. This is achieved by using the `br` tag from HMTL. Hence, we have to define the XHTML namespace.

The complete implementation of the page is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<zk xmlns:html="http://www.w3.org/1999/xhtml">
 <div align="center">
 <window title="Welcome to the online media library" border=
                                          "normal" width="300px">
  <div align="left" style="margin-left:30pt;margin-top:10pt;
                                              margin-bottom:10pt">
  <toolbarbutton label="Add a new media" href="add-media.zul"/>
  <html:br />
  <toolbarbutton label="Update a media" href="update-media.zul"/>
  <html:br />
  <toolbarbutton label="Delete a media" href="delete-media.zul"/>
  <html:br />
  <toolbarbutton label="Show media" href="show-media.zul"/>
  </div>
 </window>
 </div>
</zk>
```

A better solution for the scenario is to combine the result page and the menu on one page. Therefore, we need no extra page for showing the media. The redesigned first page (`index.zul`) is shown in the figure below:


Welcome to the online media library

Add a new media
Update a media
Delete a media
Show media

	Id	Title
	the id	The media title

For displaying the result in the first step, we use the `listbox` component. The first column is an image with a link to remove a media item. Here, we can use the `toolbarbutton` component with the `image` attribute.

```
<toolbarbutton image="images/trashcan.gif" />
```

To define the header of the grid, we can use the `listhead` component in combination with the `listheader` component.

```
<listhead>
 <listheader label="" />
 <listheader label="Id" />
 <listheader label="Title" />
</listhead>
```

To represent a list of data you can use the `listbox` component or the `grid` component. To decide whether to use the `grid` component or the `listbox` component you have to define what you want to do with the data. A `grid` component is designed for showing data. The main use of `listbox` is to show data that should be selected by the user.

The data for the `listbox` component is provided from the underlying model classes. The model classes are simple **POJO** (Plain Ordinary/Old Java Objects).

How to use the ${each} variable inside a zscript block

The `${each}` is an Expression Language variable, and therefore, it cannot be used directly inside a `zscript` block. But in some cases (especially in event handlers) it's very useful to use `${each}`. It's possible to define a custom attribute, and place the variable in the custom-attributes map for the component. With `<custom-attributes thename="${each}" />` the `${each}` is stored with key `thename` in the mentioned map. In the zscript block, it's now possible to access this object with `componentScope.get("thename")`.

The data in the `listbox` component is shown with the help for the `forEach` attribute in combination with the `${each}` EL (Expression Language) variable.

```
<listitem forEach="${list}">
  <listcell style="width:30px">
    <toolbarbutton image="images/trashcan.gif"
                   tooltip="tooltip.remove">
      <custom-attributes myMedia="${each}"/>
        <attribute name="onClick"><![CDATA[
          dao.removeMedia(componentScope.get("myMedia"));
          Executions.sendRedirect("index.zul");
        ]]>
        </attribute>
    </toolbarbutton>
  </listcell>
  <listcell label="${each.id}"/>
  <listcell label="${each.title}"/>
</listitem>
```

The full implementation of the redesigned page is given below. Here, we want to mix elements from XHTML with elements from ZUL. For that we need a clear separation of the two worlds. In an XML file, there is an element that helps us to do this separation, the namespace. If we use a namespace we have to add a prefix to each tag from that namespace (e.g. `<html:br />`). For further information about namespaces visit `http://www.w3.org/TR/REC-xml-names/`.

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

<zk xmlns:html="http://www.w3.org/1999/xhtml">
  <div align="center">
    <window title="Welcome to the online media library"
            border="normal" width="300px">
      <div align="left" style="margin-left:30pt;margin-top:10pt;
          margin-bottom:10pt">
        <toolbarbutton label="Add a new media"
```

```
                href="add-media.zul"/>
          <html:br />
          <toolbarbutton label="Update a media"
                href="update-media.zul"/>
        </div>
      </window>
    </div>
    <zscript><![CDATA[
      import java.util.Collection;
      import com.packtpub.zk.media.dao.*;
    ]]>
    </zscript>
    <html:br/>
    <listbox mold="paging">
      <zscript>
        MediaDAO dao = MediaDAOFactory.getDAO();
        Collection list = dao.getMedia();
      </zscript>
      <listhead>
        <listheader label="" />
        <listheader label="Id" />
        <listheader label="Title" />
      </listhead>
      <listitem forEach="${list}">
        <listcell style="width:30px">
          <toolbarbutton image="images/trashcan.gif"
              tooltip="tooltip.remove">
            <custom-attributes myMedia="${each}"/>
              <attribute name="onClick"><![CDATA[
                dao.removeMedia(componentScope.get("myMedia"));
                Executions.sendRedirect("index.zul");
              ]]>
              </attribute>
          </toolbarbutton>
        </listcell>
        <listcell label="${each.id}"/>
        <listcell label="${each.title}"/>
      </listitem>
    </listbox>
    <popup id="tooltip.remove">
       Remove the media.
    </popup>
  </zk>
```

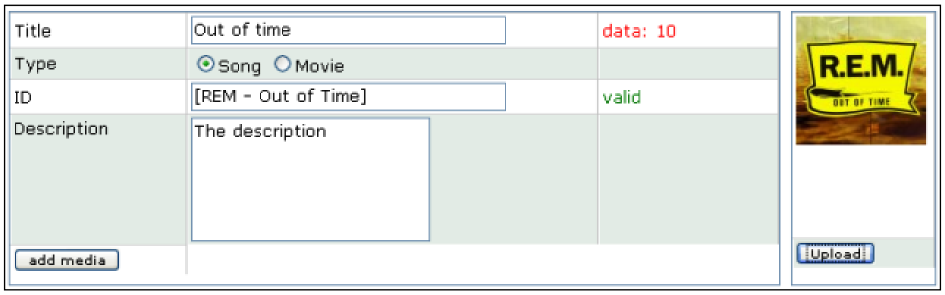
How to implement a tooltip

One aim in the design of user interfaces should be 'as easy as possible'. However, the user should also be guided with some help by the application itself on his or her first touch. One help could be a tooltip for the elements. For this, most of the ZK elements have the `tooltip` attribute (e.g: `<button label="Upload" tooltip="tooltip.upload">`). The tooltip itself is realized with the help of the `popup` component. That means, beyond the tooltip attribute, you have to provide a popup implementation for the tooltip (e.g. `<popup id="tooltip.upload">Upload a image for the media.</popup>`).

Sometimes we may want to provide some error handling for the page, especially, if we have to add a warning dialog before we really remove a media item from the underlying persistent storage.

The last screen that we want to implement in the first round is the page for adding new media.

The following figure shows the page:



Title	Out of time	data: 10
Type	[✓ Song] Movie	
ID	[REM - Out of Time]	valid
Description	The description	
add media		

How to store data in the model classes

On one side, we have the ZUL page, and on the other side the model classes. In the case of entering data, the programmer would want to use the model class directly to store the data. For this, we need some "glue" code. The ZK framework solves that by binding of data with annotations in the ZUL pages. To use such annotations we have to define a namespace for them: `xmlns:a=http://www.zkoss.org/2005/zk/annotation`. Additionally, we also have to initiate the mechanism of annotation in the page. This is done with a special init class (`org.zkoss.zkplus.databind.AnnotateDataBinderInit`). Now the page is ready to bind the input directly to the model classes. To bind data, we have to place an annotation over a normal element (e.g. `<a:bind value="object.name" />` binds the following input to the `name` property to `object`; note: we have to create an instance object in the `zscript` block).

The uploading of an image is done with the `Fileupload` dialog from the ZK framework.

[If you want to upload more than one file within one dialog you should use the `Fileupload.get(int)` method. For example `Fileupload.get(5)` presents a dialog with five fields for five uploads.]

First we have to define a `button` component where we connect the `onClick` event (`<attribute name="onClick">`) with the opening of the dialog.

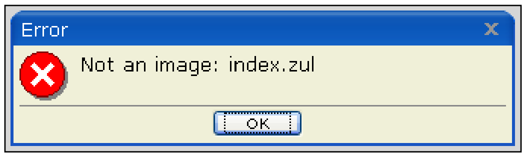
```
<button label="Upload" tooltip="tooltip.upload">
  <attribute name="onClick">
  {
    Object media = Fileupload.get();
    if (media instanceof org.zkoss.image.Image)
    {
      org.zkoss.image.Image img = (org.zkoss.image.Image) media;
      image.setContent(img);
      imagegrid.setWidth(""+(img.getWidth()+10)+"px");
    }
    else if (media != null)
        Messagebox.show("Not an image: "+media, "Error",
                               Messagebox.OK, Messagebox.ERROR);
  }</attribute>
</button>
```

If the uploaded file is an unacceptable media we would like to provide an error message to the user. For that, we use `Messagebox` from the ZK framework.

```
Messagebox.show("Not an image: "+media, "Error",

                    Messagebox.OK, Messagebox.ERROR);
```

The mentioned `Messagebox` is shown as a modal dialog. An example of the appearance in the case of an error is shown in the following figure.


Error

Not an image: index.zul

The gray background comes from the ZK framework. You have to acknowledge the message by clicking the **OK** button.

After the upload of a correct image it is shown directly, and without reloading the page. To set the image on the correct component, we address the component with `id`. For that we defined an `image` component with `id="image"`.

```
<image id="image" />
```

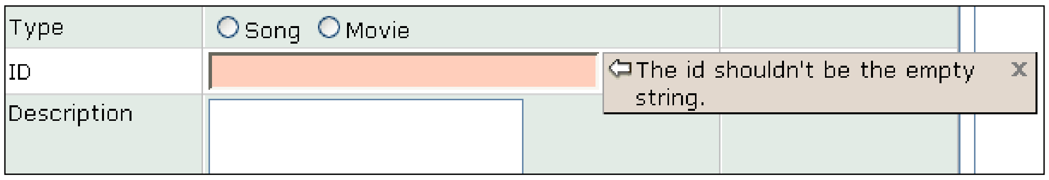
The field `id` is extended with two validations. The first validation is done while the user inserts some input into the field. For that we provided some code for the `onChanging` handler. If the user types only white spaces there is the warning **not valid** on the right side. Otherwise the green note **valid** is presented.

```
<attribute name="onChanging">
  if (event.getValue().trim().length() > 0)
  {
    idstatus.setValue("valid");
    idstatus.setStyle("color:green");
  }
  else
  {
    idstatus.setValue("not valid");
    idstatus.setStyle("color:red");
  }
</attribute>
```

The next validation is done in the `onChange` handler.

```
<attribute name="onChange">
  if (self.value != null &amp;&amp; self.value.trim().length() == 0)
  {
    self.value = "";
    throw new WrongValueException(self, "The id shouldn't be the
                                 empty string.");
  }
</attribute>
```

We use the possibility to throw a `org.zkoss.zk.ui.WrongValueException`. If this exception occurs the framework shows an error message linked to the concerned field. The following figure shows such an error message.



The id shouldn't be the empty string.

It's important to say that the code in an event handler or a zscript must be valid XML, because a ZUL page is an XML page. Here, we are confronted with the problem of using some special characters (e.g. & or <). We have two possibilities to use such characters. The first is to use valid XML entities (e.g. `&amp;&amp;` for `&&`) or we can embed the code in a `CDATA` block (`<![CDATA[ … ]]>`). For more readable code it's recommended to use the `CDATA` block variant.

The complete implementation of `add-media.zul` is presented below:

```
<?xml version="1.0" encoding="UTF-8"?>
<?init class="org.zkoss.zkplus.databind.AnnotateDataBinderInit" ?>
<window id="mainwin"
   xmlns:a="http://www.zkoss.org/2005/zk/annotation">
<zscript>
  import com.packtpub.zk.media.model.MediaType;
  import com.packtpub.zk.media.dao.*;
</zscript>
<!-- Component to describe the media -->
<hbox>
<grid width="700px" height="320px">
  <rows>
    <row>
      <label value="Title"/>
      <a:bind value="media.title"/>
      <textbox width="80%" id="titlecontent" tooltip="tooltip.title">
        <attribute name="onChanging">
          if (event.getValue().length() == 0)
          {
            titlecount.setValue("");
          }
          else
          {
            //titlecount.setValue(""+event.getValue().length());
            titlecount.setValue(com.packtpub.zk.media.
                                           MyDataProvider.next());
          }
        </attribute>
      </textbox>
      <label id="titlecount" style="color:red" />
    </row>
    <row>
      <label value="Type" />
        <radiogroup>
          <zscript>
```

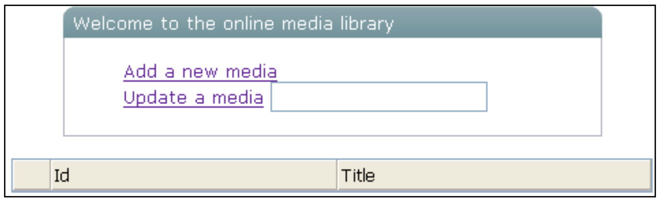
```
                MediaType[] types ={MediaType.Song,MediaType.Movie};
              </zscript>
              <a:bind value="media.type" />
              <radio label="${each.screenText}"
                     tooltip="tooltip.media.${each.screenText}"
                     forEach="${types}" value="${each}"/>
            </radiogroup>
          <label />
        </row>
        <row>
          <label value="ID" />
          <a:bind value="media.id"/>
            <textbox width="80%" tooltip="tooltip.id">
              <attribute name="onChanging">
                if (event.getValue().trim().length() > 0)
                {
                  idstatus.setValue("valid");
                  idstatus.setStyle("color:green");
                }
                else
                {
                  idstatus.setValue("not valid");
                  idstatus.setStyle("color:red");
                }
              </attribute>
              <attribute name="onChange">
                if (self.value != null &amp;&amp;
                      self.value.trim().length() == 0)
                {
                  self.value = "";
                  throw new WrongValueException(self, "The id shouldn't
                                              be the empty string.");
                }
              </attribute>
            </textbox>
          <label id="idstatus" style="color:red" value="not valid"/>
        </row>
        <row valign="top">
          <label value="Description"/>
            <textbox height="200px" width="50%"
                     tooltip="tooltip.description"/>
           <label />
        </row>
        <row valign="top">
          <button label="add media">
            <attribute name="onClick"><![CDATA[
              dao.addMedia(media);
              Executions.sendRedirect("index.zul");
              ]]>
            </attribute>
          </button>
        </row>
      </rows>
    </grid>
    <grid height="320px" id="imagegrid">
      <rows>
        <row height="284px" valign="top">
          <image id="image"/>
        </row>
        <row>
          <button label="Upload" tooltip="tooltip.upload">
            <attribute name="onClick">
              {
                Object media = Fileupload.get();
                if (media instanceof org.zkoss.image.Image)
                {
                  org.zkoss.image.Image img = (org.zkoss.image.Image)
                        media;
                  image.setContent(img);
                  imagegrid.setWidth(""+(img.getWidth()+10)+"px");
                }
                else if (media != null)
                  Messagebox.show("Not an image: "+media, "Error",
                              Messagebox.OK, Messagebox.ERROR);
              }
            </attribute>
          </button>
        </row>
      </rows>
    </grid>
    </hbox>
    <!-- Initialize the model objects for using with anotations -->
    <zscript>
      //prepare the media instance
        com.packtpub.zk.media.model.Media media =
                com.packtpub.zk.media.model.MediaFactory.create();
        MediaDAO dao = MediaDAOFactory.getDAO();
    </zscript>
    <!-- tooltip definition -->
    <popup id="tooltip.title">
      The title of the media.
    </popup>
```

```
<popup id="tooltip.media.Song">
  A Song.
</popup>
<popup id="tooltip.media.Movie">
  A Movie.
</popup>
<popup id="tooltip.id">
  The id for the media. That is a user defined id. The id
                                          must be unique.
</popup>
<popup id="tooltip.description">
  The description for the media.
</popup>
<popup id="tooltip.upload">
  Upload a image for the media.
</popup>
</window>
```

How to redirect to other pages

It is often necessary to redirect to another page after an action is executed. In that case, the ZK framework offers the general utility class `org.zkoss.zk.ui.Executions`. This method offers the `sendRedirect` method. In a ZUL page, you could directly access the `Executions` instance and redirect to another page (e.g. `Exectutions.sendRedirect('index.zul')`). A redirect sends a response to the client giving it a new URL. The client then requests the new URL. A forward happens internally in the servlet container. The target is given the chance to respond to the original request. If you use the browser's back button the same request is sent again.

For the first version of the update, we extend `index.zul` with a `textbox` component beside a `toolbarbutton` of **Update a media**.



The extended `index.zul` is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<zk xmlns:html="http://www.w3.org/1999/xhtml">
  <div align="center">
```

```
    <window title="Welcome to the online media library" border="normal"
            width="500px">
      <div align="left" style="margin-left:30pt;margin-top:10pt;
            margin-bottom:10pt">
        <toolbarbutton label="Add a new media" href="add-media.zul"/>
        <html:br />
        <toolbarbutton label="Update a media">
          <attribute name="onClick"><![CDATA[
                  sessionScope.put("id", mediaid.getValue());
                  Executions.sendRedirect("update-media.zul");
          ]]>
          </attribute>
        </toolbarbutton>
        <textbox id="mediaid"></textbox>
      </div>
    </window>
    </div>
  <zscript><![CDATA[
    import java.util.Collection;
    import com.packtpub.zk.media.dao.*;
  ]]>
  </zscript>
  <html:br/>
    <listbox mold="paging">
      <zscript>
        MediaDAO dao = MediaDAOFactory.getDAO();
        Collection list = dao.getMedia();
      </zscript>
      <listhead>
        <listheader label="" />
        <listheader label="Id" />
        <listheader label="Title" />
      </listhead>
      <listitem forEach="${list}">
        <listcell style="width:30px">
          <toolbarbutton image="images/trashcan.gif"
                         tooltip="tooltip.remove">
            <custom-attributes myMedia="${each}"/>
              <attribute name="onClick"><![CDATA[
                dao.removeMedia(componentScope.get("myMedia"));
                Executions.sendRedirect("index.zul");
              ]]>
            </attribute>
          </toolbarbutton>
        </listcell>
        <listcell label="${each.id}"/>
        <listcell label="${each.title}"/>
      </listitem>
    </listbox>
    <popup id="tooltip.remove">
      Remove the media.
    </popup>
  </zk>
```

The interesting point is the extended `toolbarbutton` component. The data is transferred with the `sessionScope` instance.

```
<toolbarbutton label="Update a media">
  <attribute name="onClick"><![CDATA[
    sessionScope.put("id", mediaid.getValue());
    Executions.sendRedirect("update-media.zul");
  ]]>
  </attribute>
</toolbarbutton>
```

The page `update-media.zul` is similar to `add-media.zul` except that the creation of a media instance is replaced with retrieving of the media from `sessionScope`.

For completeness the full implementation of `update-media.zul` is shown next.

```
<?xml version="1.0" encoding="UTF-8"?>
<?init class="org.zkoss.zkplus.databind.AnnotateDataBinderInit" ?>
<window id="mainwin" xmlns:a="http://www.zkoss.org/2005/zk/
annotation">
<zscript>
  import com.packtpub.zk.media.model.MediaType;
  import com.packtpub.zk.media.dao.*;
</zscript>
<!-- Component to describe the media -->
<hbox>
<grid width="700px" height="320px">
  <rows>
    <row>
      <label value="Title"/>
        <a:bind value="media.title"/>
        <textbox width="80%" id="titlecontent"
                 tooltip="tooltip.title">
          <attribute name="onChanging">
            if (event.getValue().length() == 0)
            {
              titlecount.setValue("");
            }
            else
            {
```

```
          //titlecount.setValue(""+event.getValue().length());
          titlecount.setValue(com.packtpub.zk.media.
                       MyDataProvider.next());
        }
      </attribute>
    </textbox>
  <label id="titlecount" style="color:red" />
</row>
<row>
  <label value="Type" />
    <radiogroup>
      <zscript>
        MediaType[] types = {MediaType.Song,MediaType.Movie};
      </zscript>
        <a:bind value="media.type" />
        <radio label="${each.screenText}"
               tooltip="tooltip.media.${each.screenText}"
               forEach="${types}" value="${each}"/>
    </radiogroup>
  <label />
</row>

<row>
  <label value="ID" />
    <a:bind value="media.id"/>
    <textbox width="80%" tooltip="tooltip.id">
      <attribute name="onChanging">
        if (event.getValue().trim().length() > 0)
        {
          idstatus.setValue("valid");
          idstatus.setStyle("color:green");
        }
        else
        {
          idstatus.setValue("not valid");
          idstatus.setStyle("color:red");
        }
      </attribute>
      <attribute name="onChange">
        if (self.value != null &amp;&amp;
               self.value.trim().length() == 0)
        {
          self.value = "";
          throw new WrongValueException(self, "The id shouldn't
                    be the empty string.");
        }
      </attribute>
```

```
          </textbox>
        <label id="idstatus" style="color:red" value="not valid"/>
      </row>
      <row valign="top">
        <label value="Description"/>
          <textbox height="200px" width="50%"
                   tooltip="tooltip.description"/>
        <label />
      </row>
      <row valign="top">
        <button label="add media">
          <attribute name="onClick"><![CDATA[
            dao.addMedia(media);
            Executions.sendRedirect("index.zul");
          ]]>
          </attribute>
        </button>
      </row>
    </rows>
  </grid>
  <grid height="320px" id="imagegrid">
    <rows>
      <row height="284px" valign="top">
        <image id="image"/>
      </row>
      <row>
        <button label="Upload" tooltip="tooltip.upload">
          <attribute name="onClick">
            {
              Object media = Fileupload.get();
              if (media instanceof org.zkoss.image.Image)
              {
                org.zkoss.image.Image img = (org.zkoss.image.Image)
                     media;
                image.setContent(img);
                imagegrid.setWidth(""+(img.getWidth()+10)+"px");
              }
              else if (media != null)
                Messagebox.show("Not an image: "+media, "Error",
                            Messagebox.OK, Messagebox.ERROR);
            }
          </attribute>
        </button>
      </row>
    </rows>
  </grid>
  </hbox>
  <!-- Initialize the model objects for using with anotations -->
  <zscript><![CDATA[
    //prepare the media instance
    MediaDAO dao = MediaDAOFactory.getDAO();
    com.packtpub.zk.media.model.Media media =
          dao.getMediaById(sessionScope.get("id"));
    ]]>
  </zscript>
  <!-- tooltip definition -->
  <popup id="tooltip.title">
    The title of the media.
  </popup>
  <popup id="tooltip.media.Song">
    A Song.
  </popup>
  <popup id="tooltip.media.Movie">
    A Movie.
  </popup>
  <popup id="tooltip.id">
    The id for the media. That is a user defined id. The id must not be
    unique.
  </popup>
  <popup id="tooltip.description">
    The description for the media.
  </popup>
  <popup id="tooltip.upload">
    Upload a image for the media.
  </popup>
  </window>
```

Now we have a small, but running CRUD application with the ZK framework. In the coming chapters, we will improve this application step by step.

Summary

In this chapter, we started implementing a CRUD (Create—Read—Update—Delete) application. Before starting the implementation of the application, we created the setup of the ZK application in the web container. After the preparation of the project, we defined some model classes to store data.

After these cornerstones, we designed and implemented the pages with the ZK framework. We have not used the full functionality of the framework here, but showed some solutions for daily work with it.

The application has not finished with this chapter, but we have some base functionality, which we will further expand in the coming chapters.

3
Extending the Online Media Library

In the last chapter, we started the implementation of a CRUD application. We implemented the application with some basics of ZK application. Now it's time to extend and optimize the application from the last chapter.

AJAX—Live Data

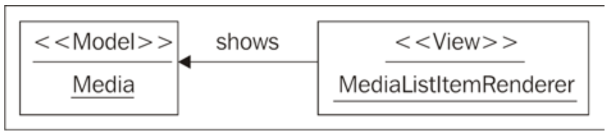
There are many discussions around AJAX. Everybody understands different things if we are talking about richness and responsiveness of an application. However, one thing everybody would agree is that one feature of a user interface should be to show actual data. In the start page (`index.zul`) in Chapter 2, we implemented a `listbox`, which shows the media instances from the underlying persistent storage. The `listbox` component has a header (realized with the `listheader` component) and the content. In addition to the attributes of a media class there is one column (symbolized with the recycling bin image) to delete a media item. If we click on the image the media is removed, and with `Executions.sendRedirect("index.zul")` the page is reloaded. The following code recapitulates the implementation of this section.

```
<listbox mold="paging">
  <zscript>
    MediaDAO dao = MediaDAOFactory.getDAO();
    Collection list = dao.getMedia();
  </zscript>
  <listhead>
    <listheader label="" />
    <listheader label="Id" />
    <listheader label="Title" />
```

```
        </listhead>
        <listitem forEach="${list}">
          <listcell style="width:30px">
            <toolbarbutton image="images/trashcan.gif"
                           tooltip="tooltip.remove">
              <custom-attributes myMedia="${each}"/>
              <attribute name="onClick"><![CDATA[
                dao.removeMedia(componentScope.get("myMedia"));
                Executions.sendRedirect("index.zul");
                ]]>
              </attribute>
            </toolbarbutton>
          </listcell>
          <listcell label="${each.id}"/>
          <listcell label="${each.title}"/>
        </listitem>
      </listbox>
```

This implementation works, but it's not really AJAX. This is because with `Executions.sendRedirect("index.zul")` we reload the page. However, one feature of ZK is to do such things without loading the whole page again. Therefore, it's now time to use the feature of the ZK framework that will load data behind the scenes. Instead of providing data for the `listbox` with the help of `forEach` attribute, we have to provide the data with the help of the `org.zkoss.zul.ListModel` interface. ZK offers an implementation of the interface, which is called `org.zkoss.zul.SimpleListModel`. With this class, it's possible to provide data for one column, but in our case, we have more than one column. Therefore, we have to implement our own `ListModel`. Additionally we have to implement the interface, `org.zkoss.zul.ListItemRenderer`. Without such an implementation, `listbox` cannot render the cells. The default renderer supports a string array model, and a single column list. The mentioned interface has only one method (`render(Listitem, Object)`), which must be implemented. In this method, instances of type `org.zkoss.zul.Listcell` are created. It's important to set the parent to a newly created instance. The following code demonstrates the implementation of the renderer class in our example.

The model-view-controller pattern is used, but without a controller.



```
public class MediaListItemRenderer implements ListitemRenderer
{
  public void render(Listitem item, Object data) throws Exception
  {
    if (data instanceof Media)
    {
      Media media = (Media) data;
      Listcell cell = null;
      cell = new Listcell();
      cell.setParent(item);
      cell.appendChild(this.createRemoveButton(media, cell));
      // The id of the media
      cell = new Listcell();
      cell.setParent(item);
      cell.setLabel(media.getId());
      // The title of the Media
      cell = new Listcell();
      cell.setParent(item);
      cell.setLabel(media.getTitle());
    }
  }
  private Toolbarbutton createRemoveButton(final Media media,
                        final Component parent)
  {
    Toolbarbutton button = new Toolbarbutton();
    button.addEventListener("onClick", new EventListener()
    {
      public boolean isAsap()
      {
        return false;
      }
      public void onEvent(Event event)
      {
        MediaDAOFactory.getDAO().removeMedia(media);
      }
    });
    button.setSrc("images/trashcan.gif");
    button.setTooltip("tooltip.remove");
    button.setParent(parent);
    return button;
  }
}
```

In the next piece, we will need an implementation of `org.zkoss.zul.ListModel`. The framework provides an abstract implementation (`org.zkoss.zul.AbstractListModel`) of that interface.

```
public class MediaListModel extends AbstractListModel
{
  /**
  * @see org.zkoss.zul.ListModel#getElementAt(int)
  */
  public Object getElementAt(final int index)
  {
    return MediaDAOFactory.getDAO().getMedia().toArray()[index];
  }
  /**
  * @see org.zkoss.zul.ListModel#getSize()
  */
  public int getSize()
  {
    return MediaDAOFactory.getDAO().getMedia().toArray().length;
  }
  /**
  * Remove a media item.
  * For the removal a ListDataEvent is fired.
  * @param media The media which should be removed.
  */
  public void removeMedia(final Media media)
  {
    this.fireEvent(ListDataEvent.CONTENTS_CHANGED, 0,
                              this.getSize()-1);
  }
}
```

One remarkable thing in the implementation above is the method `removeMedia(Media)`. In the case that a media item is removed, an event of type `org.zkoss.zul.event.ListDataEvent` is fired. With this event, the `listbox` is actualized. To have more control about registering listeners, we provide a custom implementation of `org.zkoss.zul.Listbox`. The implementation is as follows:

```
public class MyListbox extends Listbox  implements MediaDAOListener
{
  /**
  * Constructor.
  *
  */
```

```
    public MyListbox()
    {
    }
    public void removeMedia(final Media media)
    {
      ((MediaListModel)this.getModel()).removeMedia(media);
    }
    /**
    * Setting the id and register the Listbox as listener for the
      MediaDAOFactory.
    */
    @Override
    public void setId(final String id)
    {
      super.setId(id);
      MediaDAOFactory.registerListener(this);
    }
  }
```

This implementation is used through the `use` attribute of the `listbox` component.

```
<listbox id="myList" use="com.packtpub.zk.media.view.MyListbox"
         itemRenderer="com.packtpub.zk.media.view.
         MediaListItemRenderer" model="${model}">
```

To complete the description, the complete implementation of the changed `index.zul` page is shown next:

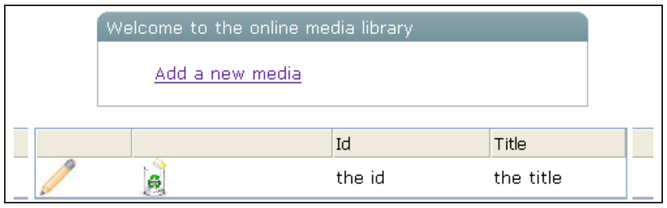
```
<?xml version="1.0" encoding="UTF-8"?>
<zk xmlns:html="http://www.w3.org/1999/xhtml">
  <window>
    <div align="center">
      <window title="Welcome to the online media library"
              border="normal" width="500px">
        <div align="left" style="margin-left:30pt;margin-top:10pt;
             margin-bottom:10pt">
          <toolbarbutton label="Add a new media"
                         href="add-media.zul"/>
          <html:br />
          <toolbarbutton label="Update a media">
            <attribute name="onClick"><![CDATA[
              sessionScope.put("id", mediaid.getValue());
```

```
                  Executions.sendRedirect("update-media.zul");
                  ]]>
                </attribute>
              </toolbarbutton>
              <textbox id="mediaid"></textbox>
            </div>
          </window>
        </div>
        <zscript><![CDATA[
          import java.util.Collection;
          import com.packtpub.zk.media.dao.*;
          import com.packtpub.zk.media.model.*;
          import com.packtpub.zk.media.view.*;
          ]]>
        </zscript>
        <html:br/>
        <zscript><![CDATA[
          ListModel model = new MediaListModel();
          ]]>
        </zscript>
        <listbox id="myList" use="com.packtpub.zk.media.view.MyListbox"
                 itemRenderer="com.packtpub.zk.media.view.
                 MediaListItemRenderer" model="${model}">
          <listhead>
            <listheader label="" />
            <listheader label="Id" />
            <listheader label="Title" />
          </listhead>
        </listbox>
        <popup id="tooltip.remove">
          Remove the media.
        </popup>
      </window>
    </zk>
```

The `grid` component supports live data, too. However, to implement the `ListItemRenderer` it's necessary to implement `org.zkoss.zul.RowRenderer`. For providing data, a `ListModel` is used.

Updating at the Right Place

In the last chapter, we implemented a minimal update. However, the functionality is not really the normal way an update should work. Therefore, it's better to move the update to the result table. The following screenshot depicts the way an update should appear in the table.



If the user clicks on the pencil, the selected media should be opened in the update mask. For that case, we have to update the `com.packtpub.zk.media.view.MediaListItemRenderer` class. We append an additional column for the update.

How to access the sessionScope in a Java class

In a ZUL page, we can access a session directly with the variable `sessionScope`. However, in a Java class we haven't used that variable, because there are no global variables in Java. To work with the current session in a class, we can use the utility class `org.zkoss.zk.ui.Sessions`. With the `getCurrent()` method, we get access to the current session. Now it's possible to set (method: `setAttribute`) and get (method: `getAttribute`) attributes from the session.

The class is extended with the `createEditButton` method for creating a `toolbarbutton` for the update.

```
public class MediaListItemRenderer implements ListitemRenderer
{
  public void render(Listitem item, Object data) throws Exception
  {
    if (data instanceof Media)
    {
      Media media = (Media) data;
      Listcell cell = null;
      //The update button
      cell = new Listcell();
      cell.setParent(item);
      cell.appendChild(this.createEditButton(media, cell));
      //The remove button
      cell = new Listcell();
```

```
        cell.setParent(item);
        cell.appendChild(this.createRemoveButton(media, cell));
        //The id of the media
        cell = new Listcell();
        cell.setParent(item);
        cell.setLabel(media.getId());
        //The title of the Media
        cell = new Listcell();
        cell.setParent(item);
        cell.setLabel(media.getTitle());
      }
    }
    private Toolbarbutton createEditButton(final Media media,
                               final Component parent)
    {
      Toolbarbutton button = new Toolbarbutton();
      button.addEventListener("onClick", new EventListener()
      {
        public boolean isAsap()
        {
          return false;
        }
        public void onEvent(Event event)
        {
          Sessions.getCurrent().setAttribute("id", media.getId());
          Executions.sendRedirect("update-media.zul");
        }
      });
      button.setSrc("images/pencil.jpg");
      button.setTooltip("tooltip.update");
      button.setParent(parent);
      return button;
    }
    private Toolbarbutton createRemoveButton(final Media media,
                               final Component parent)
    {
      Toolbarbutton button = new Toolbarbutton();
      button.addEventListener("onClick", new EventListener()
      {
        public boolean isAsap()
        {
          return false;
        }
        public void onEvent(Event event)
        {
          MediaDAOFactory.getDAO().removeMedia(media);
        }
      });
      button.setSrc("images/trashcan.gif");
      button.setTooltip("tooltip.remove");
```

```
        button.setParent(parent);
        return button;
      }
    }
```

On the `index.zul` page, we have to extend the `listbox` component with `listheader`.

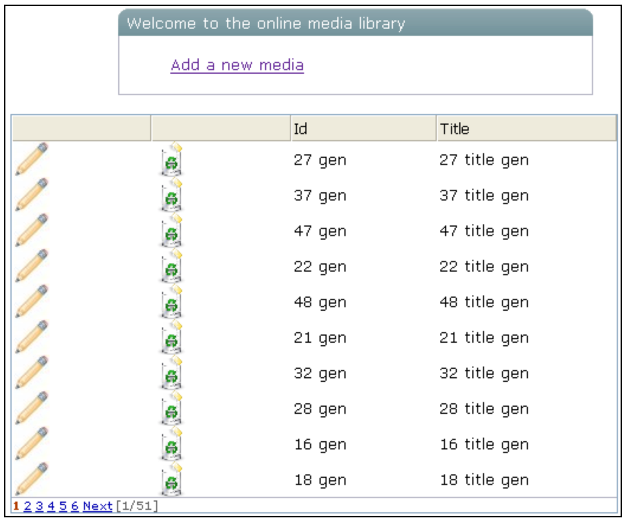
```
<listbox id="myList" use="com.packtpub.zk.media.view.MyListbox"
         itemRenderer="com.packtpub.zk.media.view.
         MediaListItemRenderer" model="${model}">
  <listhead>
    <listheader label="" />
    <listheader label="" />
    <listheader label="Id" />
    <listheader label="Title" />
  </listhead>
</listbox>
```

Additionally, we have to add a popup with the `tooltip.update` `id` to provide the tooltip for the pencil symbol.

```
<popup id="tooltip.update">
  Update the media.
</popup>
```

Optimize the Result Presentation

If we have many results in the list, we need paging. This feature is provided through a mold. In practice, even with a mold, the lists in ZK have proven to be too slow when getting bigger (10K elements). We just have to change the `listbox` component on `index.zul`, and the following presentation of the result.



Welcome to the online media library

Add a new media

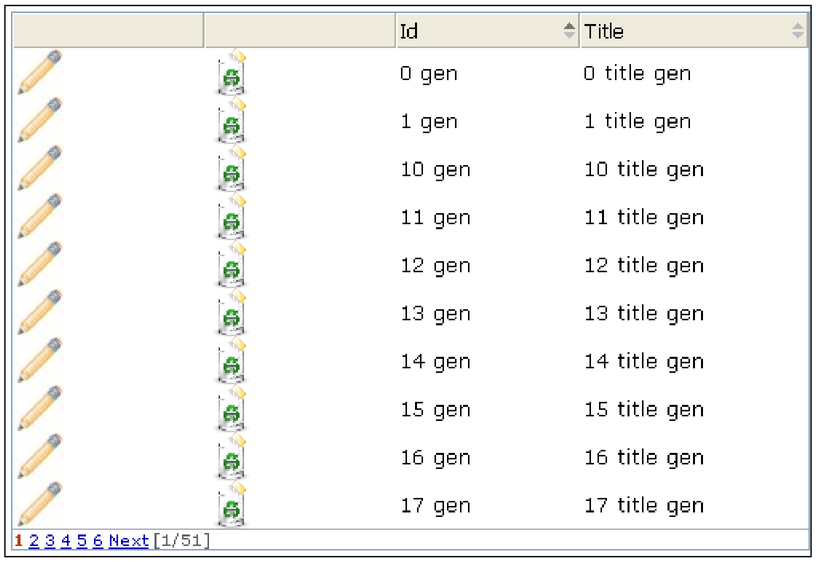
		Id	Title
✎	🗑	27 gen	27 title gen
✎	🗑	37 gen	37 title gen
✎	🗑	47 gen	47 title gen
✎	🗑	22 gen	22 title gen
✎	🗑	48 gen	48 title gen
✎	🗑	21 gen	21 title gen
✎	🗑	32 gen	32 title gen
✎	🗑	28 gen	28 title gen
✎	🗑	16 gen	16 title gen
✎	🗑	18 gen	18 title gen

1 2 3 4 5 6 Next [1/51]

The changed `listbox` is shown in the following code:

```
<listbox mold="paging" pageSize="10" id="myList"
         use="com.packtpub.zk.media.view.MyListbox"
         itemRenderer="com.packtpub.zk.media.view.
         MediaListItemRenderer" model="${model}">
  <listhead>
    <listheader label="" />
    <listheader label="" />
    <listheader label="Id" />
    <listheader label="Title" />
  </listhead>
</listbox>
```

For lists with many entries, it makes sense to have a sorting facility. This feature is supported directly, too. If we enable the feature, the result looks like the figure that follows. The columns that are now extended with sorting capability have two arrows.



		Id	Title
✏	🗑	0 gen	0 title gen
✏	🗑	1 gen	1 title gen
✏	🗑	10 gen	10 title gen
✏	🗑	11 gen	11 title gen
✏	🗑	12 gen	12 title gen
✏	🗑	13 gen	13 title gen
✏	🗑	14 gen	14 title gen
✏	🗑	15 gen	15 title gen
✏	🗑	16 gen	16 title gen
✏	🗑	17 gen	17 title gen

1 2 3 4 5 6 Next [1/51]

The simplest way to extend a column with the sort feature is to set the `sort` attribute with a value `auto`.

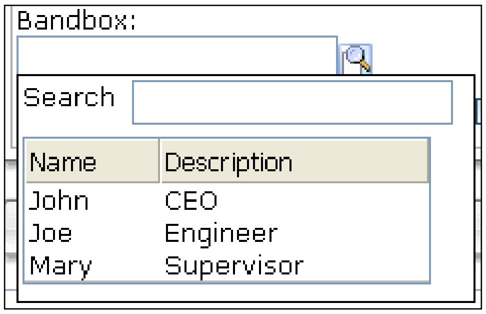
```
<listbox mold="paging" pageSize="10" id="myList"
         use="com.packtpub.zk.media.view.MyListbox"
         itemRenderer="com.packtpub.zk.media.view.
         MediaListItemRenderer" model="${model}">
  <listhead>
```

```
        <listheader label="" />
        <listheader label="" />
        <listheader label="Id" sort="auto"/>
        <listheader label="Title" sort="auto"/>
      </listhead>
    </listbox>
```

If some special sort order is necessary, it's possible to provide our own implementation of `java.util.Comparator`. To provide such a comparator to your `listheader`, you could specify the `sortAscending` and `sortDescending` attributes. This means you could provide different orders for ascending and descending sorts.

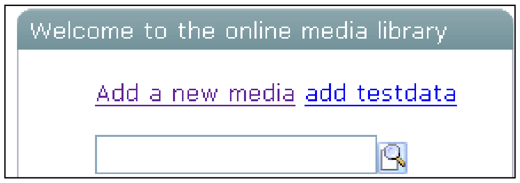
Improve Navigation Inside the Data

Actually, we only have the possibility of seeing all data in the result list, and navigating with the help of paging. It's not really satisfying. To improve the application, we should add a search facility. Here, we don't just provide a simple input field; we want to offer some exciting functionality. The idea is to use the `bandbox` component. A bandbox suggests possible choices for the input field. An example **Bandbox** is shown in the figure below.

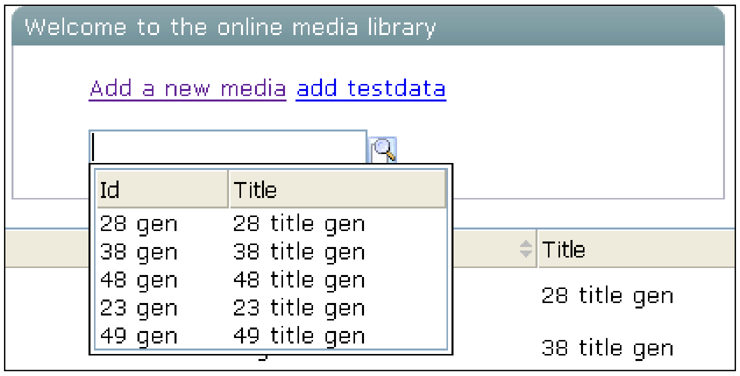


Name	Description
John	CEO
Joe	Engineer
Mary	Supervisor

The special property of a bandbox component is that the popup is customizable. Now we will try to implement such a bandbox with a live data result grid.



Additionally for large data, it's not wise to present the user with a complete list of data. We present them only the first five records. Additionally, if we select a record from the pop-up list, the title of that record should appear in the input field, and the popup should close. The result should look like the following figure. For faster generation of test data, we have added a small functionality to generate such data. This feature is available with the link **add testdata**.



Id	Title
28 gen	28 title gen
38 gen	38 title gen
48 gen	48 title gen
23 gen	23 title gen
49 gen	49 title gen

Inside a bandbox, we can use the same components that are used outside a bandbox. Therefore, we use the `listbox` component for the result list. To fill the `listbox` with live data, we have to provide the data in `ListModel` (an instance of `org.zkoss.zul.ListModel`). Since we have more than one column, we have to implement an instance of `org.zkoss.zul.ListitemRenderer`. The implementation of this renderer is shown in the following listing.

```
public class MediaListItemRenderer implements ListitemRenderer
{
  public void render(Listitem item, Object data) throws Exception
  {
    if (data instanceof Media)
    {
      Media media = (Media) data;
      Listcell cell = null;
      // The id of the media
      cell = new Listcell();
      cell.setParent(item);
      cell.setLabel(media.getId());
      // The title of the Media
      cell = new Listcell();
      cell.setParent(item);
      cell.setLabel(media.getTitle());
      //Set the value of the item for the selection
```

```
            item.setValue(media.getTitle());
          }
        }
      }
```

We manually set the value of `listitem` with `item.setValue(media.getTitle())` to have the possibility of accessing that value in the `onSelect` listener of the `listbox`. Most of the ZK controls implement the `setValue` and `getValue` methods, as a way of exchanging data over the application.

After seeing the implementation of the `ListItemRenderer`, we should look at the implementation of the `ListModel`.

```
public class SearchMediaListModel extends AbstractListModel
{
  /**
  * @see org.zkoss.zul.ListModel#getElementAt(int)
  */
  public Object getElementAt(final int index)
  {
    String searchattribute = (String)
               Sessions.getCurrent().getAttribute(„searchattribute");
    if (searchattribute == null)
    {
      searchattribute = „";
    }
    return MediaDAOFactory.getDAO().searchByTitle(searchattribute, 5)
               .toArray()[index];
  }
  /**
  * @see org.zkoss.zul.ListModel#getSize()
  */
  public int getSize()
  {
    String searchattribute = (String)
               Sessions.getCurrent().getAttribute(„searchattribute");
    if (searchattribute == null)
    {
      searchattribute = „";
    }
    return MediaDAOFactory.getDAO().searchByTitle(searchattribute, 5)
               size();
  }
  public void update()
  {
    this.fireEvent(ListDataEvent.CONTENTS_CHANGED, 0,
               this.getSize() - 1);
  }
}
```

The `update` method is used to update the data in the listbox with the help of firing events from the type `ListDataEvent`. We want to have the feature that on each character the user types into the input field of the bandbox, the data of the result list is actualized. To transfer the data from the input field to the model, we use the session.

To finish the implementation, we have to add the bandbox to `index.zul`.

```
<?xml version="1.0" encoding="UTF-8"?>
<zk xmlns:html="http://www.w3.org/1999/xhtml">
  <zscript><![CDATA[
    import java.util.Collection;
    import com.packtpub.zk.media.dao.*;
    import com.packtpub.zk.media.model.*;
    import com.packtpub.zk.media.view.*;
    import com.packtpub.zk.media.view.search.*;
    MediaDAO dao = MediaDAOFactory.getDAO();
    ListModel searchmodel = new SearchMediaListModel();
    ]]>
  </zscript>
  <window>
    <div align="center">
      <window title="Welcome to the online media library"
              border="normal" width="500px">
        <div align="left" style="margin-left:30pt;
             margin-top:10pt; margin-bottom:10pt">
          <toolbarbutton label="Add a new media"
                         href="add-media.zul"/>
          <toolbarbutton label="add testdata">
            <attribute name="onClick">
              MediaDAOFactory.create(50);
              Executions.sendRedirect("index.zul");
            </attribute>
          </toolbarbutton>
          <html:br />
          <html:br />
          <bandbox id="bd" autodrop="true">
            <attribute name="onChanging">
              sessionScope.put("searchattribute", event.getValue());
              searchlistbox.getModel().update();
            </attribute>
            <attribute name="onOpen">
              searchlistbox.getModel().update();
            </attribute>
            <bandpopup>
              <listbox id="searchlistbox" width="200px"
                       itemRenderer="com.packtpub.zk.media.view.
                       search.MediaListItemRenderer"
                       model="${searchmodel}">
                <attribute name="onSelect">
                  bd.value=self.selectedItem.value;
                  bd.closeDropdown();
                </attribute>
                <listhead>
                  <listheader label="Id"/>
                  <listheader label="Title"/>
                </listhead>
              </listbox>
            </bandpopup>
          </bandbox>
        </div>
      </window>
    </div>
    <html:br/>
    <zscript><![CDATA[
      ListModel model = new MediaListModel();
      ]]>
    </zscript>
    <listbox mold="paging" pageSize="10" id="myList"
             use="com.packtpub.zk.media.view.MyListbox"
             itemRenderer="com.packtpub.zk.media.view.
             MediaListItemRenderer" model="${model}">
      <listhead>
        <listheader label="" />
        <listheader label="" />
        <listheader label="Id" sort="auto"/>
        <listheader label="Title" sort="auto"/>
      </listhead>
    </listbox>
    <popup id="tooltip.remove">
      Remove the media.
    </popup>
    <popup id="tooltip.update">
      Update the media.
    </popup>
  </window>
</zk>
```

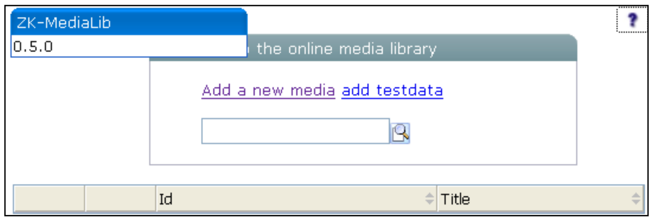
If the user selects a line of the list, the title of the media is shown in the input field, and the bandbox is shown.

```
<attribute name="onSelect">
  bd.value=self.selectedItem.value;
  bd.closeDropdown();
</attribute>
```

The title of the media is the value of the selected item. We have set that value in our implementation of the `ListItemRenderer`.

Some Candies for the User

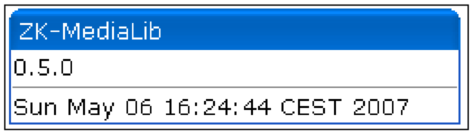
Now that we have added some really necessary functionalities, it's time to add some features from the category "nice to have". The first one that each application should have is version information. For that, we add a query symbol (realized in a `toolbarbutton` component), which opens a `window` component as a popup. The following figure shows the result.



The implementation of this feature is not really difficult, and is shown next:

```
<zk>
  <window id="version" border="normal" width="200px" visible="false">
    <zscript>
      String appVersion = com.packtpub.zk.media.
                              ZKMediaLibVersion.VERSION;
      String appName = com.packtpub.zk.media.
                              ZKMediaLibVersion.NAME;
    </zscript>
    <caption label="${appName}"/>
    <label value="${appVersion}" />
  </window>
</zk>
<div align="right">
  <toolbarbutton image="/images/query-symbol.jpg">
    <attribute name="onClick">
      version.doPopup();
    </attribute>
  </toolbarbutton>
</div>
```

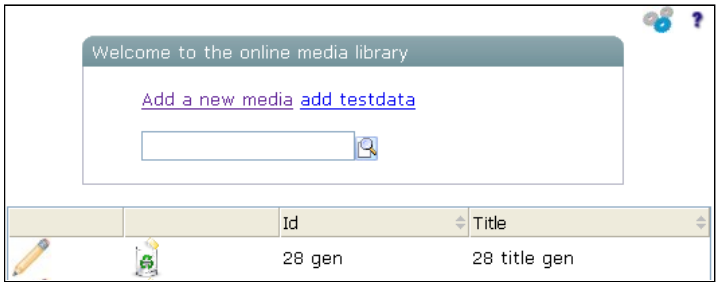
Now the version information should be extended with the timer component from ZK. This component sends an onTimer event at regular intervals. The intervals could be controlled with the help of the delay attribute of the component. The appearance of the changed version information is shown in the following figure:


```
ZK-MediaLib
0.5.0
Sun May 06 16:24:44 CEST 2007
```

Additionally, with the timer, we use the separator component for a visual separation of the version information and the running watch. The implementation of the improved version information is as follows:

```
<zk>
  <window id="version" border="normal" width="250px" visible="false">
    <zscript>
      String appVersion = com.packtpub.zk.media.
                             ZKMediaLibVersion.VERSION;
      String appName = com.packtpub.zk.media.
                             ZKMediaLibVersion.NAME;
    </zscript>
    <caption label="${appName}"/>
    <label value="${appVersion}" />
    <separator bar="true" />
    <label id="time"/>
    <timer id="timer" delay="1000" repeats="true"
           onTimer="time.setValue(new Date().toString())"/>
  </window>
</zk>
```

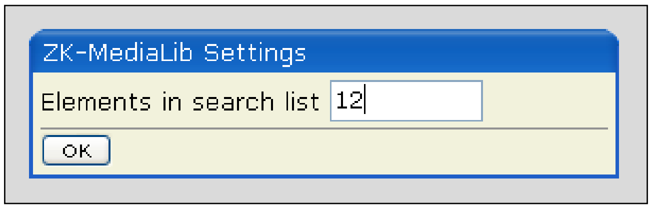
One feature that should also be available is the feature of configuration. For the first step, we add a feature to configure the number of elements that are displayed in the search preview list that is presented with the help of the bandbox component. To access the configuration window, we add a toolbarbutton component with a cogwheel image.


Welcome to the online media library

Add a new media add testdata

		Id	Title
		28 gen	28 title gen

When the `toolbarbutton` is clicked, a modal dialog with an input field for the number of elements should appear. The appearance of the dialog is shown in the following figure:


ZK-MediaLib Settings

Elements in search list 12

The default value of the number of elements should be initialized at the start of the page. For that, we have to provide an implementation of the `org.zkoss.zk.ui.util.Initiator`. interface.

```
public class SettingsInitializer implements Initiator
{
  public void doAfterCompose(final Page page) throws Exception
  {
  }
  public void doCatch(final Throwable ex)
  {
  }
  public void doFinally()
  {
  }
  public void doInit(final Page page, final Object[] args) throws
                 Exception
  {
    InputStream in = SettingsInitializer.class.getClassLoader().
                 getResourceAsStream("settings.properties");
    Properties properties = new Properties();
    properties.load(in);
    SettingsFactory.get().setNoOfElementsInResult(Integer.parseInt
                 (properties.getProperty("noOfElementsInResult")));
  }
}
```

To use this implementation, we have to define the `init` class at the beginning of the page.

```
<?init class="com.packtpub.zk.media.bootstrap.SettingsInitializer" ?>
```

The implementation of the modal dialog is the same as the implementation of a normal dialog. We lay out the components in a normal `window` component. The implementation is as follows:.

```
<zk>
  <window id="settingsWindow" border="normal" width="450px"
          visible="false">
    <caption label="${appName} Settings"/>
    <label value="Elements in search list" />
    <zscript>
    Settings settingsStorage = SettingsFactory.get();
    </zscript>
    <a:bind value="settingsStorage.noOfElementsInResult"/>
    <textbox id="noOfElementsInResult"/>
    <separator bar="true" />
    <button label="OK">
      <attribute name="onClick">
        settingsWindow.doModal();
      </attribute>
    </button>
  </window>
</zk>
```

The value is set with the help of an annotation. To use this annotation, we have to add an `init` sequence for the `org.zkoss.zkplus.databind.AnnotateDataBinderInit` class.

> It's important to know that a page can have more than one class for the `init` sequence.

To use that implementation we have to define that `init` class at the beginning of the page.

```
<?init class="org.zkoss.zkplus.databind.AnnotateDataBinderInit" ?>
```

To complete the picture the complete implementation of the new `index.zul` is shown below.

```
<?xml version="1.0" encoding="UTF-8"?>
<zk xmlns:html="http://www.w3.org/1999/xhtml"
            xmlns:a="http://www.zkoss.org/2005/zk/annotation">
  <?init class="org.zkoss.zkplus.databind.AnnotateDataBinderInit" ?>
  <?init class="com.packtpub.zk.media.bootstrap.SettingsInitializer"
            ?>
  <zscript><![CDATA[
    import java.util.Collection;
    import com.packtpub.zk.media.dao.*;
    import com.packtpub.zk.media.model.*;
    import com.packtpub.zk.media.view.*;
    import com.packtpub.zk.media.view.search.*;
    import com.packtpub.zk.media.bootstrap.*;
    import com.packtpub.zk.media.*;
    MediaDAO dao = MediaDAOFactory.getDAO();
    ListModel searchmodel = new SearchMediaListModel();
    ]]>
  </zscript>
  <!-- the version information -->
  <zk>
    <window id="version" border="normal" width="250px"
            visible="false">
      <zscript>
        String appVersion = com.packtpub.zk.media.
                              ZKMediaLibVersion.VERSION;
        String appName = com.packtpub.zk.media.
                              ZKMediaLibVersion.NAME;
      </zscript>
      <caption label="${appName}"/>
      <label value="${appVersion}" />
      <separator bar="true" />
      <label id="time"/>
      <timer id="timer" delay="1000" repeats="true"
             onTimer="time.setValue(new Date().toString())"/>
    </window>
  </zk>
  <div align="right">
    <toolbarbutton image="/images/cogwheel.jpg">
      <attribute name="onClick">
        settingsWindow.doModal();
      </attribute>
    </toolbarbutton>
    <toolbarbutton image="/images/query-symbol.jpg">
      <attribute name="onClick">
        version.doPopup();
      </attribute>
    </toolbarbutton>
  </div>
  <!-- settings -->
  <zk>
    <window id="settingsWindow" border="normal" width="450px"
            visible="false">
      <caption label="${appName} Settings"/>
      <label value="Elements in search list" />
      <zscript>
        Settings settingsStorage = SettingsFactory.get();
      </zscript>
      <a:bind value="settingsStorage.noOfElementsInResult"/>
      <textbox id="noOfElementsInResult"/>
      <separator bar="true" />
      <button label="OK">
        <attribute name="onClick">
          settingsWindow.doPopup();
        </attribute>
      </button>
    </window>
  </zk>
  <window>
    <div align="center">
      <window title="Welcome to the online media library"
              border="normal" width="500px">
        <div align="left" style="margin-left:30pt;margin-top:10pt;
             margin-bottom:10pt">
          <toolbarbutton label="Add a new media"
                         href="add-media.zul"/>
          <toolbarbutton label="add testdata">
            <attribute name="onClick">
              MediaDAOFactory.create(50);
              Executions.sendRedirect("index.zul");
            </attribute>
          </toolbarbutton>
          <html:br />
          <html:br />
          <bandbox id="bd" autodrop="true">
            <attribute name="onChanging">
              sessionScope.put("searchattribute", event.getValue());
              searchlistbox.getModel().update();
            </attribute>
            <attribute name="onOpen">
              searchlistbox.getModel().update();
            </attribute>
            <bandpopup>
              <listbox id="searchlistbox" width="200px"
                       itemRenderer="com.packtpub.zk.media.
                       view.search.MediaListItemRenderer"
                       model="${searchmodel}">
                <attribute name="onSelect">
                  System.out.println(self.selectedItem);
                  bd.value=self.selectedItem.value;
```

```
                  bd.closeDropdown();
                </attribute>
                <listhead>
                  <listheader label="Id"/>
                  <listheader label="Title"/>
                </listhead>
              </listbox>
            </bandpopup>
          </bandbox>
        </div>
      </window>
    </div>
    <html:br/>
    <zscript><![CDATA[
      ListModel model = new MediaListModel();
      ]]>
    </zscript>
    <listbox mold="paging" pageSize="10" id="myList"
             use="com.packtpub.zk.media.view.MyListbox"
             itemRenderer="com.packtpub.zk.media.view.
             MediaListItemRenderer" model="${model}">
      <listhead>
        <listheader label="" />
        <listheader label="" />
        <listheader label="Id" sort="auto"/>
        <listheader label="Title" sort="auto"/>
      </listhead>
    </listbox>
    <popup id="tooltip.remove">
      Remove the media.
    </popup>
    <popup id="tooltip.update">
      Update the media.
    </popup>
    <popup id="tooltip.query">
      About the ZK-Medialib
    </popup>
  </window>
</zk>
```

Summary

Our second iteration of the application is now complete. Before we proceed with the last iteration, and give finishing touches in the next chapter, we should look back at what we have done, and learned in this chapter.

At the beginning of this chapter, we started with a simple CRUD application. The first thing we have done is to add some AJAX features to the application (live data). Here, we learned that many cornerstones are provided from the ZK framework, and that we only have to implement some interfaces to use these features. We moved the application from a mixed code approach to a Model-View-Controller Architecture. The use of the "Live Data" concept is the only way in ZK to use this design pattern. It's advantages, against the flat approach, are code reusability (model and controller could be inherited from other projects addressing the same business logic; the view is the only part dependent on ZK.

With the help of the live data it's possible to give the user the feeling of a desktop application, and that's one of the reasons why we are using ZK.

The next steps in this chapter extended the application, in particular the `index.zul` page. In the last steps, we added version information, and a settings dialog.

Where to get more help

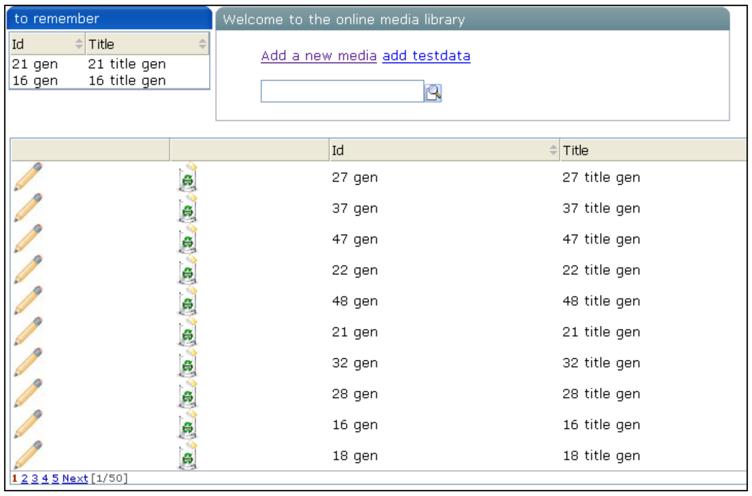
In some situations it's not really clear how to choose the right user interface controls with the ZK framework. In such cases, it's wise to look into the provided `zkdemo` application from the ZK framework. In this application each component is used and you could get a feeling of how to use that component.

4
Is it on the Desktop or on the Web?

In the last two chapters, we implemented the basic functionality of an online media library. There we started with a simple page (`index.zul`), which we extended with some additional features. Now in this chapter, we run the last cycle of our implementation. The goal of the chapter is to have an application that will give the user a feeling of a desktop application.

Adding Drag-and-Drop

One feature of many desktop applications is drag-and-drop. Therefore, we should implement such a feature, too. We want to have the ability to drag a list item from the result list of media to a separate list to remember the items later. The modified application screen is as shown in the figure:



to remember

Id	Title
21 gen	21 title gen
16 gen	16 title gen

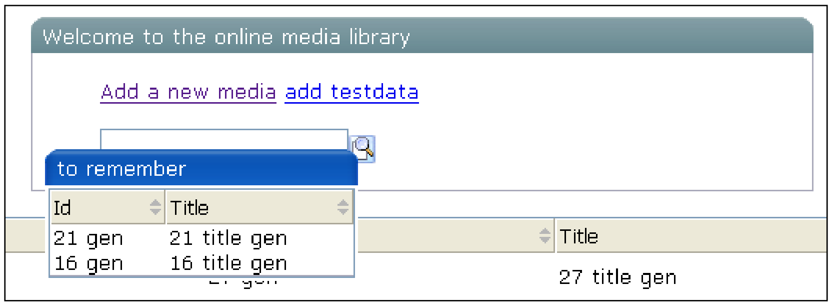
Welcome to the online media library

Add a new media add testdata

		Id	Title
✏	🗑	27 gen	27 title gen
✏	🗑	37 gen	37 title gen
✏	🗑	47 gen	47 title gen
✏	🗑	22 gen	22 title gen
✏	🗑	48 gen	48 title gen
✏	🗑	21 gen	21 title gen
✏	🗑	32 gen	32 title gen
✏	🗑	28 gen	28 title gen
✏	🗑	16 gen	16 title gen
✏	🗑	18 gen	18 title gen

1 2 3 4 5 Next [1/50]

We added a `window` component with the title **to remember** to our application. This component should be moveable around the browser, and should also be able to overlap other components. The following screenshot shows the overlapping feature:



To activate the overlap feature for a window, you simply have to call one method, `doOverlapped()`.

```
<zscript>
  rememberWindow.doOverlapped();
</zscript>
```

This additional window should have a `listbox` component with live data. For that we have to provide the data from a list model.

```
<window title="to remember" width="200px" firstLeft="10px"
firstTop="10px"
        id="rememberWindow" use="com.packtpub.zk.media.view.remember.
        RememberWindow">
  <listbox mold="paging" pageSize="10" itemRenderer="com.packtpub.
zk.media.view.
           remember.DropableMediaListItemRenderer"
model="${testmodel}">
    <listhead>
      <listheader label="Id" sort="auto"/>
      <listheader label="Title" sort="auto"/>
    </listhead>
  </listbox>
  <zscript>
    rememberWindow.doOverlapped();
  </zscript>
</window>
```

With the `doOverlapped` method, we get the ability to move the window around. However, when we click on **add testdata** to add some data to the main listbox, we enforce a reload of the page because of `Executions.sendRedirect("index.zul")` at the end of the action of **add testdata**. After reload, the position of `rememberWindow` is lost. To solve this problem, we have to store the position of the `window` component in the session, and restore that position in the construction phase of the window. To implement this feature, we have to extend `org.zkoss.zul.Window`.

```
public class RememberWindow extends Window implements EventListener
{
  /*******************************************************************
********
  * Internal Constants
  *******************************************************************
*******/
  private final static String LEFT = "LEFT";
  private final static String TOP = "TOP";
  /*******************************************************************
********
  * Instance variables
  *******************************************************************
*******/
  private boolean setBySession = false;
  private String firstTop;
  private String firstLeft;
  /*******************************************************************
********/
  /**
  * Constructor.
  * A event listener for the event <code>onMove</code> is registered.
  */
  public RememberWindow()
  {
    this.addEventListener("onMove", this);
    // Restore the left position from the session if avaiable
    if (Sessions.getCurrent().getAttribute(LEFT) != null)
    {
      this.setLeft((String) Sessions.getCurrent().getAttribute(LEFT));
      this.setBySession = true;
    }
    // Restore the top position from the session if avaiable
    if (Sessions.getCurrent().getAttribute(TOP) != null)
    {
      this.setTop((String) Sessions.getCurrent().getAttribute(TOP));
    }
  }
  /**
  * Set the top position which is used on the first time. First time 
means at
  * initialization. The position is only evaluated if the position is 
not set
  * in the constructor from a session variable.
  *
  * @param firstTop The top position.
  */
  public void setFirstTop(final String firstTop)
  {
    this.firstTop = firstTop;
    if (!this.setBySession)
    {
      this.setTop(this.firstTop);
    }
  }
  /**
  * Set the left position which is used on the first time. First time 
means at
  * initialization. The position is only evaluated if the position is 
not set
  * in the constructor from a session variable.
  *
  * @param firstLeft The left position.
  */
  public void setFirstLeft(final String firstLeft)
  {
    this.firstLeft = firstLeft;
    if (!this.setBySession)
    {
      this.setLeft(this.firstLeft);
    }
  }
  /**
  * @see EventListener#isAsap()
  */
  public boolean isAsap()
  {
    return false;
  }
  /**
  * Eventlistener method.
  */
```

```
      public void onEvent(final Event event)
      {
        // store the left position
        Sessions.getCurrent().setAttribute(RememberWindow.LEFT,
                           ((MoveEvent) event).getLeft());
        // store the right position
        Sessions.getCurrent().setAttribute(RememberWindow.TOP,
                           ((MoveEvent) event).getTop());
        System.out.println(((MoveEvent) event).getTop());
      }
    }
```

For storing the actual position of the window, we implement an event listener for the `onMove` event. In this method, we store the top and the left position of the window into the session of the user. For the first time, we want to set a position for the window, too. Here, we have to add two new attributes `firstTop` and `firstLeft`. To extend a component with some attributes, we just have to implement simple setter methods. The small window for the remembered media instances should not display the same list as the result list, therefore, we have to implement our own `ListitemRenderer`.

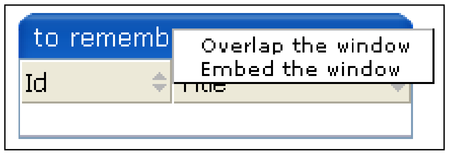
```
    public class DropableMediaListItemRenderer implements ListitemRenderer
    {
      public void render(final Listitem item, Object data) throws
    Exception
      {
        if (data instanceof Media)
        {
          final Media media = (Media) data;
          Listcell cell = null;
          // The id of the media
          cell = new Listcell();
          cell.setParent(item);
          cell.setLabel(media.getId());
          // The title of the Media
          cell = new Listcell();
          cell.setParent(item);
          cell.setLabel(media.getTitle());
          item.setDroppable("true");
          item.addEventListener("onDrop", new EventListener()
          {
            public boolean isAsap()
            {
              return false;
            }
            @SuppressWarnings("unchecked")
            public void onEvent(Event event)
            {
              Listitem it = (Listitem) ((DropEvent)event).getDragged();
              Object[] childs = it.getChildren().toArray();
              RememberList remember = (RememberList) Sessions.getCurrent()
                    .getAttribute(RememberListInitializer.REMEMBER_LIST);
              Media media = MediaFactory.create(((Listcell)
                                                childs[4]).getLabel());
              media.setTitle(((Listcell)childs[3]).getLabel());
              media.setId(((Listcell)childs[2]).getLabel());
              remember.add(media);
              ((RememberMediaListModel)((Listbox)item.getParent())
                                               .getModel()).update();
            }
          });
        }
      }
    }
```

One notable line in the code is `item.setDroppable("true")`. With this line, we activate the `listbox` as a possible destination of the drag-and-drop process. To activate the main `listbox` as possible source of a drag-and-drop process, we have to add the line `item.setDraggable("true")` to `com.packtpub.zk.media.view.MediaListItemRenderer`. Additionally we implement our own `ListModel` for that window.

```
public class RememberMediaListModel extends AbstractListModel
implements ListModel
{
  /**
  * @see org.zkoss.zul.ListModel#getElementAt(int)
  */
  public Object getElementAt(final int index)
  {
    return this.getMedia().toArray()[index];
  }
  /**
  * @see org.zkoss.zul.ListModel#getSize()
  */
  public int getSize()
  {
    return this.getMedia().size();
  }
```

```
    @SuppressWarnings("unchecked")
    private RememberList getMedia()
    {
      return (RememberList) Sessions.getCurrent().getAttribute(
                     RememberListInitializer.REMEMBER_LIST);
    }
    public void update()
    {
      this.fireEvent(ListDataEvent.CONTENTS_CHANGED, 0,
                                            this.getSize() - 1);
    }
  }
```

With these pieces, we have a working drag-and-drop, as well as a window that we can move around. Now we want to have the option to embed the window. To choose this option, we add a context menu to the `remember` window. The result should appear like the following figure:



The context menu is realized as `menupopup` component. The elements in that menu are `menuitem` components. With the help of the `autocheck` attribute, the check symbol for the element in the menu is changed every time an entry is selected.

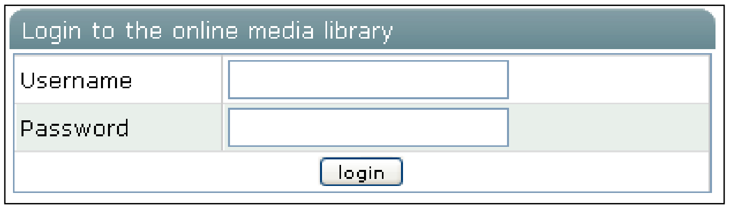
```
  <menupopup id="rememberOptions">
    <menuitem label="Overlap the window" autocheck="true"
  id="rememberOverlap">
      <attribute name="onClick">
        rememberWindow.doOverlapped();
        rememberEmbed.setChecked(false);
      </attribute>
    </menuitem>
    <menuitem label="Embed the window" autocheck="true"
  id="rememberEmbed">
      <attribute name="onClick">
        rememberWindow.doEmbedded();
        rememberOverlap.setChecked(false);
      </attribute>
    </menuitem>
  </menupopup>
```

If we click on **Embed the window** the window is embedded by calling `doEmbedded()`. To mark the menu as context menu to the `rememberWindow` we use the `context` attribute of the window component.

```
<window title="to remember" width="200px" firstLeft="10px"
firstTop="10px"
        id="rememberWindow" use="com.packtpub.zk.media.view.remember.
        RememberWindow" context="rememberOptions" border="true">
  <listbox mold="paging" pageSize="10" itemRenderer=
                "com.packtpub.zk.media.view.remember
                .DropableMediaListItemRenderer" model="${testmodel}">
    <listhead>
      <listheader label="Id" sort="auto"/>
      <listheader label="Title" sort="auto"/>
    </listhead>
  </listbox>
  <zscript>
    rememberWindow.doOverlapped();
  </zscript>
</window>
```

Adding a Login Page

Now it's time to add a login page to our application. The screen itself is very simple, and appears like the following figure:



The two fields **Username** and **Password** should have validation. The validation is enabled with the help of the `constraint` attribute. This attribute can contain a combination of the following options:

- no positive
- no negative
- no zero
- no empty

- no future
- no past
- no today
- a regular expression

In our example, we use `no empty`.

How to validate a form

In the earlier chapters, we used `WrongValueException` to validate a single input field. However, when we click on the button and submit the data of form, validation doesn't happen automatically. To validate we have to call `getValue()` on an input field. If the validation is broken a popup occurs on the concerned field. If we want to prevent such popup, we can call the `isValid()` function and create our own error message.

The complete login page (`login.zul`) is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<zk xmlns:html="http://www.w3.org/1999/xhtml"
    xmlns:a="http://www.zkoss.org/2005/zk/annotation">
  <?init class="org.zkoss.zkplus.databind.AnnotateDataBinderInit" ?>
  <zscript>
    import com.packtpub.zk.media.model.*;
  </zscript>
  <div align="center">
    <window width="400px" title="Login to the online media library">
      <grid>
        <rows>
          <row>
            <label value="Username" />
            <a:bind value="user.name"/>
            <textbox id="username"/>
          </row>
          <row>
            <label value="Password" />
              <a:bind value="user.password" />
              <textbox type="password" id="password" constraint="no
empty"/>
          </row>
          <row spans="2" align="center">
            <button label="login">
              <attribute name="onClick">
                //validate the username
```

```
                    username.getValue();
                    //validate the password
                    password.getValue();
                    UserFactory.login(user);
                    Executions.sendRedirect("index.zul");
                  </attribute>
                </button>
              </row>
            </rows>
          </grid>
        </window>
      </div>
      <!-- Initialize the model objects for using with anotations -->
      <zscript>
        com.packtpub.zk.media.model.User user = com.packtpub.zk.media.
                          model.UserFactory.create();
      </zscript>
    </zk>
```

The user instance is stored into the session. For minimal security, which isn't enough for production environments, we use an implementation of `javax.servlet.Filter` and register it in `web.xml`.

```
<filter>
  <filter-name>SecurityFilter</filter-name>
  <filter-class>com.packtpub.zk.media.controller.SecurityFilter</
filter-class>
  <init-param>
    <param-name>loginPage</param-name>
    <param-value>login.zul</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>SecurityFilter</filter-name>
  <url-pattern>*.zul</url-pattern>
</filter-mapping>
```

To complete the example, implementation of the filter method `doFilter` is as follows:

```
public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException
{
  HttpSession session = ((HttpServletRequest) request).getSession();
```

```
    if (session != null && session.getAttribute
                                    (SESSIONKEY_USER) != null)
    {
      User user = (User) session.getAttribute(SESSIONKEY_USER);
      if (user.isLoggedIn())
      {
        chain.doFilter(request, response);
      }
    }
    if (loginPage != null)
    {
      // forward to the loginpage
      request.getRequestDispatcher(loginPage).forward
                                  (request, response);
    }
    else
    {
      chain.doFilter(request, response);
    }
  }
```

[For a production environment it's recommended to embed the Acegi framework (`http://www.acegisecurity.org`) for security. An example for integration is shown in the next chapter.]

One minimal problem of the login mask is that the user has to press the **login** button and the enter key is not recognized. If the user presses the enter key the `onOK` event is sent. Therefore, we adapt the `window` component in `login.zul`.

```
<window width="400px" title="Login to the online media library"

        onOK="login()">
```

The method `login` is the extracted login method of the button. For completeness, the new `login.zul` is as shown:

```
<?xml version="1.0" encoding="UTF-8"?>
<zk xmlns:html="http://www.w3.org/1999/xhtml"
    xmlns:a="http://www.zkoss.org/2005/zk/annotation">
  <?init class="org.zkoss.zkplus.databind.AnnotateDataBinderInit" ?>
  <zscript>
    import com.packtpub.zk.media.model.*;
  </zscript>
  <div align="center">
    <window width="400px" title="Login to the online media library"
```

```
              onOK="login()">
        <grid>
          <rows>
            <row>
              <label value="Username" />
              <a:bind value="user.name"/>
              <textbox id="username" constraint="no empty"/>
            </row>
            <row>
              <label value="Password" />
                <a:bind value="user.password" />
                <textbox type="password" id="password"
                                         constraint="no empty"/>
            </row>
            <row spans="2" align="center">
              <button label="login">
                <attribute name="onClick">
                  login();
                </attribute>
              </button>
            </row>
          </rows>
        </grid>
      </window>
    </div>
    <zscript>
      void login()
      {
        //validate the username
        username.getValue();
        //validate the password
        password.getValue();
        UserFactory.login(user);
        Executions.sendRedirect("index.zul");
      }
    </zscript>
    <!-- Initialize the model objects for using with anotations -->
      <zscript>
        com.packtpub.zk.media.model.User user =
                    com.packtpub.zk.media.model.UserFactory.create();
      </zscript>
  </zk>
```

Before we proceed, it's time to show the complete `index.zul` page.

```
<?xml version="1.0" encoding="UTF-8"?>
<zk xmlns:html="http://www.w3.org/1999/xhtml"
    xmlns:a="http://www.zkoss.org/2005/zk/annotation">
  <?init class="org.zkoss.zkplus.databind.AnnotateDataBinderInit" ?>
  <?init class="com.packtpub.zk.media.bootstrap.SettingsInitializer"
?>
  <?init class="com.packtpub.zk.media.bootstrap.
RememberListInitializer" ?>
  <zscript><![CDATA[
    import java.util.Collection;
    import com.packtpub.zk.media.dao.*;
    import com.packtpub.zk.media.model.*;
    import com.packtpub.zk.media.view.*;
    import com.packtpub.zk.media.view.search.*;
    import com.packtpub.zk.media.view.remember.*;
    import com.packtpub.zk.media.bootstrap.*;
    import com.packtpub.zk.media.*;
    MediaDAO dao = MediaDAOFactory.getDAO();
    ListModel searchmodel = new SearchMediaListModel();
    User user = Sessions.getCurrent().getAttribute(com.packtpub.
                zk.media.controller.SecurityFilter.SESSIONKEY_USER);
    ]]>
  </zscript>
  <!-- the version information -->
  <zk>
    <window id="version" border="normal" width="250px"
            visible="false">
      <zscript>
        String appVersion = com.packtpub.zk.media.
                                    ZKMediaLibVersion.VERSION;
        String appName = com.packtpub.zk.media.ZKMediaLibVersion.NAME;
      </zscript>
      <caption label="${appName}"/>
      <label value="${appVersion}" />
      <separator bar="true" />
      <label id="time"/>
      <timer id="timer" delay="1000" repeats="true"
             onTimer="time.setValue(new Date().toString())"/>
    </window>
  </zk>
  <div align="right">
    <toolbarbutton image="/images/cogwheel.jpg">
      <attribute name="onClick">
```

```
          settingsWindow.doModal();
        </attribute>
      </toolbarbutton>
      <toolbarbutton image="/images/query-symbol.jpg">
        <attribute name="onClick">
          version.doPopup();
        </attribute>
      </toolbarbutton>
      <toolbarbutton label="logout">
        <attribute name="onClick">
          UserFactory.logout();
          Executions.sendRedirect("index.zul");
        </attribute>
      </toolbarbutton>
    </div>
    <!-- settings -->
    <zk>
      <window id="settingsWindow" border="normal" width="450px"
              visible="false">
        <caption label="${appName} Settings"/>
          <label value="Elements in search list" />
          <zscript>
            Settings settingsStorage = SettingsFactory.get();
          </zscript>
          <a:bind value="settingsStorage.noOfElementsInResult"/>
          <textbox id="noOfElementsInResult"/>
          <separator bar="true" />
          <button label="OK">
            <attribute name="onClick">
              settingsWindow.doPopup();
            </attribute>
          </button>
        </window>
      </zk>
    <window>
    <div align="center">
      <window title="Hello ${user.name}, Welcome to the
                                           online media library"
              border="normal" width="500px">
        <div align="left" style="margin-left:30pt;margin-top:10pt;
             margin-bottom:10pt">
          <toolbarbutton label="Add a new media">
            <attribute name="onClick">
              Sessions.getCurrent().setAttribute("media.mode", "add");
              Executions.sendRedirect("add-update-media.zul");
            </attribute>
          </toolbarbutton>
          <toolbarbutton label="add testdata">
            <attribute name="onClick">
              MediaDAOFactory.create(50);
              Executions.sendRedirect("index.zul");
            </attribute>
          </toolbarbutton>
          <html:br />
          <html:br />
          <bandbox id="bd" autodrop="true">
            <attribute name="onChanging">
              sessionScope.put("searchattribute", event.getValue());
              searchlistbox.getModel().update();
            </attribute>
            <attribute name="onOpen">
              searchlistbox.getModel().update();
            </attribute>
            <bandpopup>
              <listbox id="searchlistbox" width="200px"
                                          itemRenderer="com.packtpub.
                       zk.media.view.search.MediaListItemRenderer"
                       model="${searchmodel}">
                <attribute name="onSelect">
                  bd.value=self.selectedItem.value;
                  bd.closeDropdown();
                </attribute>
                <listhead>
                  <listheader label="Id"/>
                  <listheader label="Title"/>
                </listhead>
              </listbox>
            </bandpopup>
          </bandbox>
        </div>
      </window>
    </div>
    <html:br/>
    <zscript><![CDATA[
      ListModel model = new MediaListModel();
      ListModel testmodel = new RememberMediaListModel();
      ]]>
    </zscript>
```

```
     <listbox mold="paging" pageSize="10" id="myList"
                                         use="com.packtpub.zk.
              media.view.MyListbox" itemRenderer="com.packtpub
                                                 .zk.media.view.
              MediaListItemRenderer" model="${model}">
       <listhead>
         <listheader label="" />
         <listheader label="" />
         <listheader label="Id" sort="auto"/>
         <listheader label="Title" sort="auto"/>
       </listhead>
     </listbox>
     <popup id="tooltip.remove">
       Remove the media.
     </popup>
     <popup id="tooltip.update">
       Update the media.
     </popup>
     <popup id="tooltip.query">
       About the ZK-Medialib
     </popup>
   </window>
   <html:br/>
     <window title="to remember" width="200px"
                                 firstLeft="10px" firstTop="10px"
             id="rememberWindow" use="com.packtpub.zk.media.view
                                .remember.RememberWindow" context=
                                "rememberOptions" border="true">
       <listbox mold="paging" pageSize="10" itemRenderer="com.packtpub
              .zk.media.view.remember.DropableMediaListItemRenderer
              "model="${testmodel}">
         <listhead>
           <listheader label="Id" sort="auto"/>
           <listheader label="Title" sort="auto"/>
         </listhead>
       </listbox>
       <zscript>
         rememberWindow.doOverlapped();
       </zscript>
     </window>

     <menupopup id="rememberOptions">
       <menuitem label="Overlap the window" autocheck="true"
                 id="rememberOverlap">
         <attribute name="onClick">
           rememberWindow.doOverlapped();
           rememberEmbed.setChecked(false);
```

```
          </attribute>
        </menuitem>
        <menuitem label="Embed the window" autocheck="true"
                  id="rememberEmbed">
          <attribute name="onClick">
            rememberWindow.doEmbedded();
            rememberOverlap.setChecked(false);
          </attribute>
        </menuitem>
      </menupopup>
    </zk>
```

Extending Add and Update

The add media (`add-media.zul`) and update media (`update-media,zul`) features have so far been implemented as two different pages. These pages are very similar, and therefore, we can merge them. The only difference is that in the case of an update we load the data, and for adding we create a new instance. The simplest way is to extend the page with a mode parameter. The new page can have the name `add-update-media.zul`.

The first change is in the `index.zul` page:

```
<toolbarbutton label="Add a new media">
  <attribute name="onClick">
    Sessions.getCurrent().setAttribute("media.mode", "add");
    Executions.sendRedirect("add-update-media.zul");
  </attribute>
</toolbarbutton>
```

Additionally we have to extend `com.packtpub.zk.media.view.MediaListItemRenderer`.

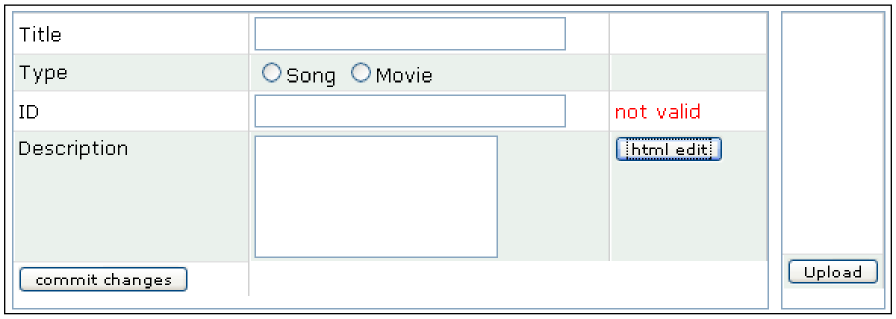
```
public void onEvent(Event event)
{
  Sessions.getCurrent().setAttribute("id", media.getId());
  Sessions.getCurrent().setAttribute("media.mode", "update");
  Executions.sendRedirect("add-media.zul");
}
```

In the new page `add-update-media.zul`, we now have to evaluate the session parameter `media.mode`.

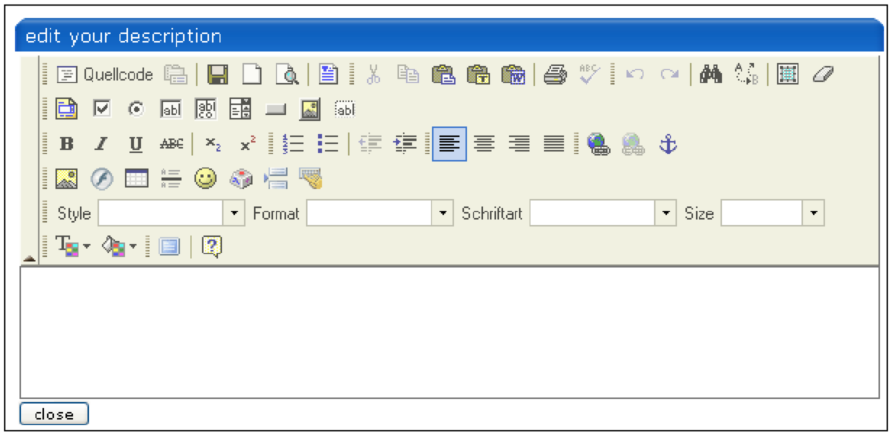
```
  <zscript>
    MediaDAO dao = MediaDAOFactory.getDAO();
    //prepare the media instance
```

```
        com.packtpub.zk.media.model.Media media = null;
        String mode = sessionScope.get("media.mode");
        if ("update".equals(mode))
        {
          media = dao.getMediaById(sessionScope.get("id"));
        }
        else
        {
          media = com.packtpub.zk.media.model.MediaFactory.create();
        }
      </zscript>
```

Now we have a single page, and we can proceed with improvements for that page. The first extension should be an HTML editor for the description. For that we can use the FCKeditor (`http://www.fckeditor.net`). This editor can be embedded into the ZK framework. For integration, we extend the add/update screen with a button (**html edit**) for opening a new window with that editor component. The following screenshot shows the adapted page.



When we click on the **html edit** button, a window with the editor appears (see the following screenshot).



The integration of the editor is very simple, because FCKeditor comes along with the normal distribution of the ZK framework. The window with the editor has the following source code:

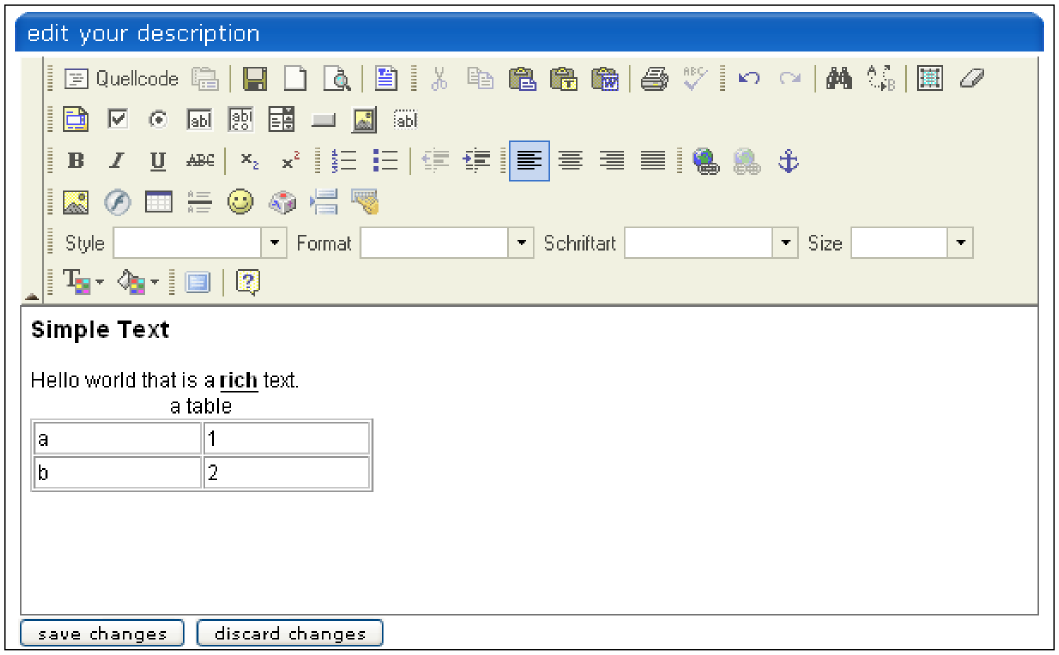
```
<window title="edit your description" width="600px" id="htmlEdit"
visible="false">
  <fckeditor height="600px"/>
  <button label="close">
    <attribute name="onClick">
      htmlEdit.setVisible(false);
    </attribute>
  </button>
</window>
```

The window is opened in the embedded mode:

```
<button label="html edit">
  <attribute name="onClick">
    htmlEdit.doOverlapped();
  </attribute>
</button>
```

With the shown implementation, we only have a window with an FCKeditor, but we haven't implemented the ability to show the edited text. For that we want to extend the add/update screen. The first extension is for the edit window itself. We add a button for discarding changes (**discard changes**) and rename the other button to **save changes**.

```
<window title="edit your description" width="600px" id="htmlEdit"
visible="false">
  <fckeditor id="fck" height="600px" />
  <button label="save changes">
    <attribute name="onClick">
      descriptionFrame.setContent(fck.value);
      htmlEdit.setVisible(false);
    </attribute>
  </button>
  <button label="discard changes">
    <attribute name="onClick">
      fck.value = descriptionFrame.getContent();
      htmlEdit.setVisible(false);
    </attribute>
  </button>
</window>
```



Simple Text

Hello world that is a **rich** text.

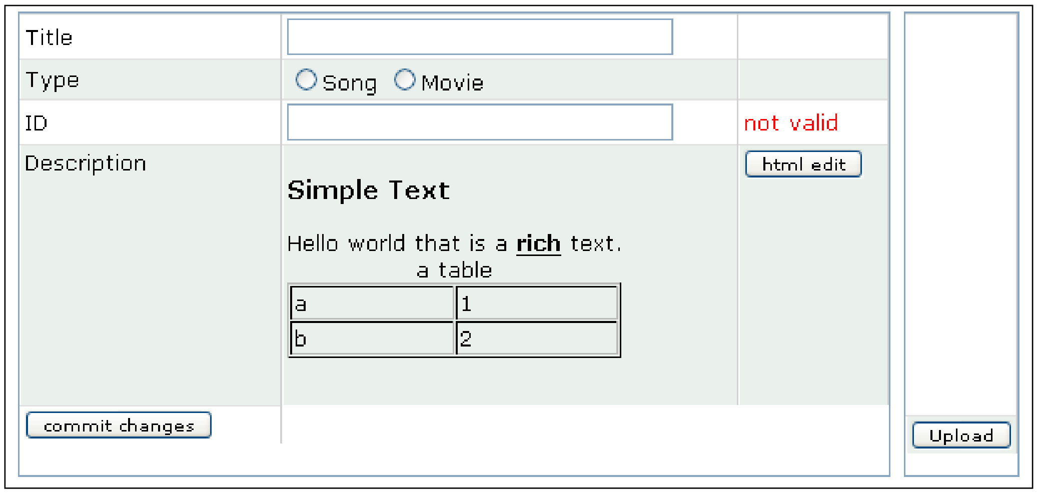
a table

a	1
b	2

To show the content, we use the `html` component of the ZK framework.

```
<html id="descriptionFrame" width="100%"/>
```

With the help of this component it's possible to show formatted HTML text.



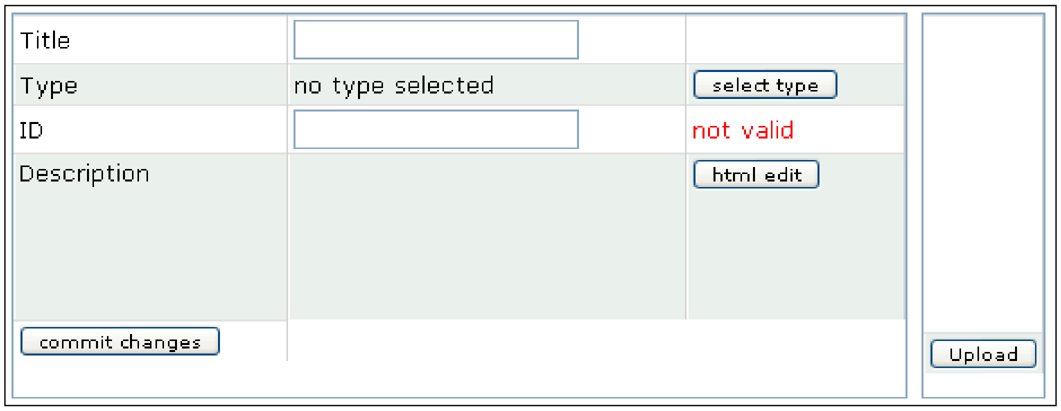
Title		
Type	○ Song ○ Movie	
ID		not valid
Description	**Simple Text** Hello world that is a **rich** text. a table	html edit

a	1
b	2

The next thing we can change is the selection of media type. We can remove the radio button group, and insert a popup with a tree with media types. The popup is opened with a **select type** button.

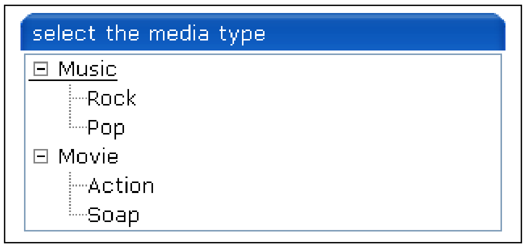
```
<button label="select type">
  <attribute name="onClick">
    mediaType.doOverlapped();
  </attribute>
</button>
```

The adapted screen looks as follows:



Title		
Type	no type selected	select type
ID		not valid
Description		html edit
commit changes		

The figure below shows the window that is opened on a click on **select type**.



The tree is build with the `tree` component. The selection of a media type should work with drag-and-drop. Therefore, we add the `draggable` attribute to the `treecell` components.

```
<window title="select the media type" id="mediaType" width="300px"
        visible="false">
  <tree id="tree">
```

```
      <treechildren>
        <treeitem>
          <treerow>
            <treecell label="Music"/>
          </treerow>
          <treechildren>
            <treeitem>
              <treerow>
                <treecell label="Rock" draggable="true"/>
              </treerow>
            </treeitem>
            <treeitem>
              <treerow>
                <treecell label="Pop" draggable="true"/>
              </treerow>
            </treeitem>
          </treechildren>
        </treeitem>
        <treeitem>
          <treerow>
            <treecell label="Movie"/>
          </treerow>
          <treechildren>
            <treeitem>
              <treerow>
                <treecell label="Action" draggable="true"/>
              </treerow>
            </treeitem>
            <treeitem>
              <treerow>
                <treecell label="Soap" draggable="true"/>
              </treerow>
            </treeitem>
          </treechildren>
        </treeitem>
      </treechildren>
    </tree>
  </window>
```

A tree could have a header. The header of a tree has the same appearance as the header of `listbox` component. A header is implemented with `treecols` and `treecol`.

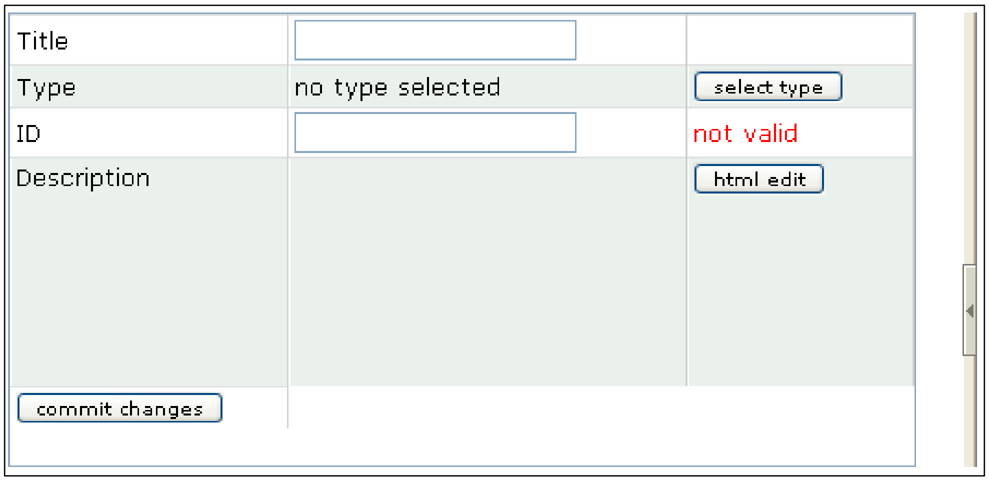
Now we have to implement a destination for the drag-and-drop. Here, we use a simple label.

```
<row>
  <label value="Type" />
  <label droppable="true" value="no type selected">
    <attribute name="onDrop"><![CDATA[
      self.value = ((Treecell)((DropEvent)event).getDragged())
                 .getTreeitem().getLabel();
      mediaType.setVisible(false);
      ]]>
    </attribute>
  </label>
  <button label="select type">
    <attribute name="onClick">
      mediaType.doOverlapped();
    </attribute>
  </button>
</row>
```

One interesting layout element is the `splitter` component. With one simple statement it's possible to add a splitter in ZK. Therefore, we just try adding a splitter between the input box, and the image upload box.

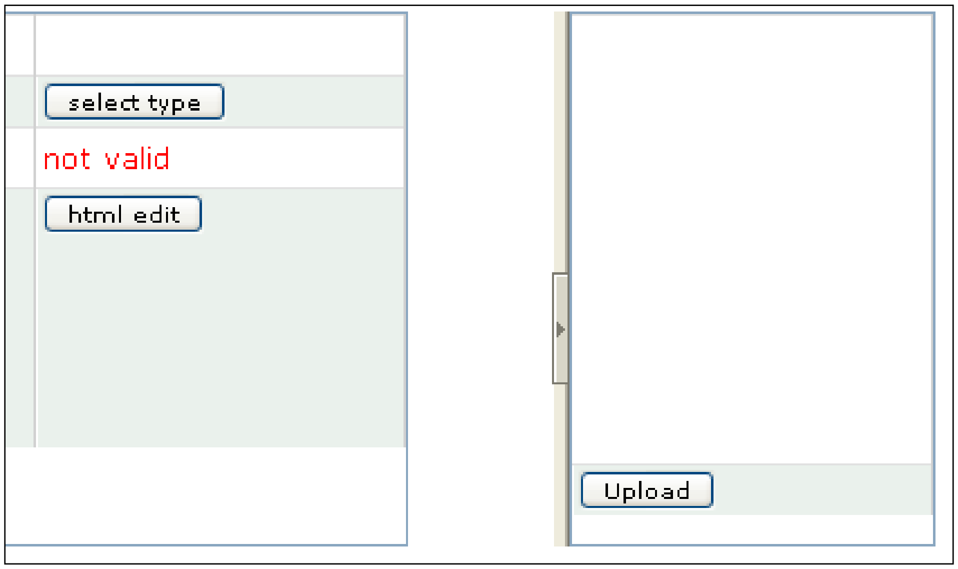
```
<splitter collapse="after" open="false"/>
```

The screen with the closed splitter (attribute `open="false"`) is as shown :



Title		
Type	no type selected	select type
ID		not valid
Description		html edit
commit changes		

When we click on the arrow, the image upload box is shown.



Now, we have a rich add/update page. To complete the example we show `add-update-media.zul`.

```
<?xml version="1.0" encoding="UTF-8"?>
  <?init class="org.zkoss.zkplus.databind.AnnotateDataBinderInit" ?>
  <window id="mainwin"
          xmlns:a="http://www.zkoss.org/2005/zk/annotation">
  <zscript>
    import com.packtpub.zk.media.model.MediaType;
    import com.packtpub.zk.media.dao.*;
    String tmpvalue = null;
  </zscript>
  <!-- htmledit for the description of a media -->
  <window title="edit your description" width="600px" id="htmlEdit"
          visible="false">
    <fckeditor id="fck" height="600px" />
    <button label="save changes">
      <attribute name="onClick">
        descriptionFrame.setContent(fck.value);
        htmlEdit.setVisible(false);
      </attribute>
    </button>
    <button label="discard changes">
```

```
      <attribute name="onClick">
        fck.value = descriptionFrame.getContent();
        htmlEdit.setVisible(false);
      </attribute>
    </button>
  </window>
  <!-- Component to describe the media -->
  <hbox>
    <grid width="700px" height="320px">
      <rows>
        <row>
          <label value="Title"/>
          <a:bind value="media.title"/>
          <textbox width="80%" id="titlecontent"
                                    tooltip="tooltip.title">
            <attribute name="onChanging">
              if (event.getValue().length() == 0)
              {
                titlecount.setValue("");
              }
              else
              {
                //titlecount.setValue(""+event.getValue().length());
                titlecount.setValue(com.packtpub.zk.media.
                                    MyDataProvider.next());
              }
            </attribute>
          </textbox>
          <label id="titlecount" style="color:red" />
        </row>
        <row>
          <label value="Type" />
          <label droppable="true" value="no type selected">
            <attribute name="onDrop"><![CDATA[
              self.value = ((Treecell)((DropEvent)event).getDragged())
                          .getTreeitem().getLabel();
              mediaType.setVisible(false);
              ]]>
            </attribute>
          </label>
          <button label="select type">
            <attribute name="onClick">
              mediaType.doOverlapped();
            </attribute>
          </button>
        </row>
        <row>
          <label value="ID" />
          <a:bind value="media.id"/>
          <textbox width="80%" tooltip="tooltip.id">
            <attribute name="onChanging">
              if (event.getValue().trim().length() > 0)
              {
                idstatus.setValue("valid");
                idstatus.setStyle("color:green");
              }
              else
              {
                idstatus.setValue("not valid");
                idstatus.setStyle("color:red");
              }
            </attribute>
            <attribute name="onChange">
              if (self.value != null &amp;&amp; self.value.trim().
                  length() == 0)
              {
                self.value = "";
                throw new WrongValueException(self, "The id shouldn't
                be the empty string.");
              }
            </attribute>
          </textbox>
          <label id="idstatus" style="color:red" value="not valid"/>
        </row>
        <row valign="top"  height="200px">
          <label value="Description" />
          <html id="descriptionFrame" width="100%"/>
          <button label="html edit">
            <attribute name="onClick">
              htmlEdit.doOverlapped();
            </attribute>
          </button>
        </row>
        <row valign="bottom">
          <button label="commit changes">
            <attribute name="onClick"><![CDATA[
              dao.addMedia(media);
              Executions.sendRedirect("index.zul");
            ]]>
          </attribute>
        </button>
      </row>
    </rows>
  </grid>
  <splitter collapse="after" open="false"/>
  <grid height="320px" id="imagegrid" width="300px">
    <rows>
      <row height="284px" valign="top">
        <image id="image"/>
      </row>
      <row>
        <button label="Upload" tooltip="tooltip.upload">
          <attribute name="onClick">
            {
              Object media = Fileupload.get();
              if (media instanceof org.zkoss.image.Image)
              {
                org.zkoss.image.Image img =
                                    (org.zkoss.image.Image) media;
                image.setContent(img);
                imagegrid.setWidth(""+(img.getWidth()+10)+"px");
              }
              else if (media != null)
                Messagebox.show("Not an image: "+media, "Error",
                                 Messagebox.OK, Messagebox.ERROR);
            }
          </attribute>
        </button>
      </row>
    </rows>
  </grid>
</hbox>
<!-- media type -->
<window title="select the media type" id="mediaType" width="300px"
        visible="false">
  <tree id="tree">
    <treechildren>
      <treeitem>
        <treerow>
          <treecell label="Music"/>
        </treerow>
        <treechildren>
```

```
            <treeitem>
              <treerow>
                <treecell label="Rock" draggable="true"/>
              </treerow>
            </treeitem>
            <treeitem>
              <treerow>
                <treecell label="Pop" draggable="true"/>
              </treerow>
            </treeitem>
          </treechildren>
        </treeitem>
        <treeitem>
          <treerow>
            <treecell label="Movie"/>
          </treerow>
          <treechildren>
            <treeitem>
              <treerow>
                <treecell label="Action" draggable="true"/>
              </treerow>
            </treeitem>
            <treeitem>
              <treerow>
                <treecell label="Soap" draggable="true"/>
              </treerow>
            </treeitem>
          </treechildren>
        </treeitem>
      </treechildren>
    </tree>
  </window>
  <!-- Initialize the model objects for using with anotations -->
  <zscript>
    MediaDAO dao = MediaDAOFactory.getDAO();
    //prepare the media instance
    com.packtpub.zk.media.model.Media media = null;
    String mode = sessionScope.get("media.mode");
    if ("update".equals(mode))
    {
      media = dao.getMediaById(sessionScope.get("id"));
    }
    else
    {
      media = com.packtpub.zk.media.model.MediaFactory.create();
    }
  </zscript>
  <!-- tooltip definition -->
  <popup id="tooltip.title">
    The title of the media.
  </popup>
  <popup id="tooltip.media.Song">
    A Song.
  </popup>
  <popup id="tooltip.media.Movie">
    A Movie.
  </popup>
  <popup id="tooltip.id">
    The id for the media. That is a user defined id.
                                             The id must be unique.
  </popup>
  <popup id="tooltip.description">
    The description for the media.
  </popup>
  <popup id="tooltip.upload">
    Upload a image for the media.
  </popup>
</window>
```

Internationalization with the ZK Framework

Before we end the chapter, and the tutorial part of this book, we should look at how internationalization is done with ZK. For that we will use `login.zul` page from the previous chapter.

The ZK framework comes with a built-in functionality to separate the name of the labels in a property file. The name of the property file is `i3-label_lang_CNTY.properties`. `lang` is the language (for example: en) and `CNTY` is the country (for example US). The file must be located in the `WEB-INF` directory of the web application. If the file is not found, the file `i3-label_lang.properties` is loaded, and if that fails the framework tries to load `i3-label.properties`. To access the properties, you can use `org.zkoss.util.resource.Labels`.

For completeness, the new implementation of `login.zul` is as shown:

```
<?xml version="1.0" encoding="UTF-8"?>
<zk xmlns:html="http://www.w3.org/1999/xhtml"
    xmlns:a="http://www.zkoss.org/2005/zk/annotation">
```

```
<?init class="org.zkoss.zkplus.databind.AnnotateDataBinderInit" ?>
<zscript>
  import com.packtpub.zk.media.model.*;
</zscript>
<div align="center">
  <window width="400px" title="Login to the online media
          library" onOK="login()">
    <grid>
      <rows>
        <zscript>
          String username = org.zkoss.util.resource.Labels
                            .getLabel("login.username");
          String password = org.zkoss.util.resource.Labels
                            .getLabel("login.password");
          String login = org.zkoss.util.resource.Labels
                            .getLabel("login.button");
        </zscript>
        <row>
          <label value="${username}" />
          <a:bind value="user.name"/>
          <textbox id="username" constraint="no empty"/>
        </row>
        <row>
          <label value="${password}" />
          <a:bind value="user.password" />
          <textbox type="password" id="password"
                                        constraint="no empty"/>
        </row>
        <row spans="2" align="center">
          <button label="${login}">
            <attribute name="onClick">
              login();
            </attribute>
          </button>
        </row>
      </rows>
    </grid>
  </window>
</div>
<zscript>
  void login()
  {
    //validate the username
    username.getValue();
```

```
        //validate the password
        password.getValue();
        UserFactory.login(user);
        Executions.sendRedirect("index.zul");
      }
    </zscript>
    <!-- Initialize the model objects for using with anotations -->
    <zscript>
      com.packtpub.zk.media.model.User user =
                      com.packtpub.zk.media.model.UserFactory.create();
    </zscript>
  </zk>
```

Internationalization

Besides using the `org.zkoss.utiil.resource.Labels` class, in the ZUL file you can use the supplied `taglib` from the ZK framework:

```
<%@ taglib uri="/WEB-INF/tld/web/core.dsp.tld"
prefix="c" %>
...
<label value="${c:l('login.password')}">
...
</window>
```

In a Java class, the only possibility is to use the `Label` class. For the ZUL file, it's easier to use the `taglib` as shown.

Summary

Now we are at the end of our third phase in the development of a CRUD application with the AJAX ZK framework. We started with a simple application, and extended it step by step. The application now has many features, which you will need in other applications as well. Besides the components used there are many other interesting components, which can be used to enrich your application and give user the feeling of a desktop application.

5
Integration with Other Frameworks

ZK's biggest strength is its productivity. But it does have other advantages:

- ZK decouples the rendering logic from the presentation logic. That means with ZK, applications can be retargetted to devices other than the browser, such as the desktop or Swing or even a mobile device.
- Async update has always been an issue. With ZK the whole async update business can be abstracted into a timer component and it is very easy to do. However, you still have to pay attention to your server-side logic.
- ZK's ZUML makes it easy to build complex UI component trees. It also shortens the edit—compile—test cycle for developers.
- ZK allows developers to keep their business logic on the server side.

ZK is a great framework but the focus is on the GUI so other frameworks may be needed. This chapter discusses the integration of ZK with other frameworks so as to overcome these weaknesses. One of the most important frameworks is **Spring**. This is very often used together with **Hibernate**. As we will see, it is very easy to integrate ZK with other frameworks. Once it was difficult to use captchas but in the latest ZK versions, ZK supports captchas natively.

Integration with the Spring Framework

Spring is one of the most complete lightweight containers, which provides centralized, automated configuration, and wiring of your application objects. It improves your application's testability and scalability by allowing software components to be first developed and tested in isolation, then scaled up for deployment in any environment. This approach is called the POJO (Plain Old Java Object) approach and is gaining popularity because of its flexibility.

So, with all these advantages, it's no wonder that Spring is one of the most used frameworks. Spring provides many nice features: however, it works mainly in the back end. Here ZK may provide support in the view layer. The benefit from this pairing is the flexible and maturity of Spring together with the easy and speed of ZK. Specify a Java class in the use attribute of a window ZUL page and the world of Spring will be yours. Remember as discussed in previous chapters what a sample ZUL looks I like:

```
<?xml version="1.0" encoding="UTF-8"?>
<p:window xmlns:p="http://www.zkoss.org/2005/zul"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.zkoss.org/2005/zul
      http://www.zkoss.org/2005/zul "
    border="normal" title="Hello!"
      use="com.myfoo.myapp.HelloController">

Thank you for using our Hello World Application.

</p:window>
```

The `HelloController` points directly to a Java class where you can use Spring features easily.

Normally, if a Java Controller is used for a ZUL page it becomes necessary sooner or later to call a Spring bean. Usually in Spring you would use the `applicationContext` like:

```
ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
UserDAO userDAO = (UserDAO) ctx.getBean("userDAO");
```

Then the `userDAO` is usable for any further access. In ZK there is a helper class `SpringUtil`. It wrapps the `applicationContext` and simplifies the code to:

```
UserDAO  userDAO = (UserDAO) SpringUtil.getBean("userDAO");
```

Pretty easy, isn't it? Let us examine an example.

Assume we have a small web application that gets flight data from a flight table. The `web.xml` file looks like:

```
<?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">
```

```
<display-name>IRT-FLIGHTSAMPLE</display-name>

<filter>
    <filter-name>hibernateFilter</filter-name>
    <filter-class>org.springframework.orm.hibernate3.support.
                 OpenSessionInViewFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>hibernateFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext-jdbc.xml
                ,classpath:applicationContext-dao.xml
                ,classpath:applicationContext-service.xml
                ,classpath:applicationContext.xml
    </param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.
                   ContextLoaderListener</listener-class>
</listener>

<session-config>
    <!-- Default to 30 minute session timeouts -->
    <session-timeout>30</session-timeout>
</session-config>

<mime-mapping>
    <extension>xsd</extension>
    <mime-type>text/xml</mime-type>
</mime-mapping>

<servlet>
    <description>
        <![CDATA[
The servlet loads the DSP pages.
        ]]>
    </description>
    <servlet-name>dspLoader</servlet-name>
    <servlet-class>
        org.zkoss.web.servlet.dsp.InterpreterServlet
    </servlet-class>
```

```
</servlet>
<servlet-mapping>
    <servlet-name>dspLoader</servlet-name>
    <url-pattern>*.dsp</url-pattern>
</servlet-mapping>

<!-- ZK -->
<listener>
    <description>
        Used to cleanup when a session is destroyed
    </description>
    <display-name>ZK Session Cleaner</display-name>
    <listener-class>
        org.zkoss.zk.ui.http.HttpSessionListener
    </listener-class>
</listener>
<servlet>
    <description>ZK loader for ZUML pages</description>
    <servlet-name>zkLoader</servlet-name>
    <servlet-class>
        org.zkoss.zk.ui.http.DHtmlLayoutServlet
    </servlet-class>
    <!-- Must. Specifies URI of the update engine
      (DHtmlUpdateServlet).
        It must be the same as <url-pattern> for the update
        engine.
    -->
    <init-param>
        <param-name>update-uri</param-name>
        <param-value>/zkau</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>zkLoader</servlet-name>
    <url-pattern>*.zul</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>zkLoader</servlet-name>
    <url-pattern>*.zhtml</url-pattern>
</servlet-mapping>
<servlet>
    <description>The asynchronous update engine for
                ZK</description>
        <servlet-name>auEngine</servlet-name>
        <servlet-class>
            org.zkoss.zk.au.http.DHtmlUpdateServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>auEngine</servlet-name>
        <url-pattern>/zkau/*</url-pattern>
    </servlet-mapping>

    <welcome-file-list id="WelcomeFileList">
        <welcome-file>index.zul</welcome-file>
    </welcome-file-list>

</web-app>
```

Furthermore let's have a small ZUL page that has the interface to retrieve and show flight data:

```
<?xml version="1.0" encoding="UTF-8"?>
<p:window xmlns:p="http://www.zkoss.org/2005/zul"
          xmlns:h="http://www.w3.org/1999/xhtml"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.zkoss.org/2005/zul
            http://www.zkoss.org/2005/zul "
          id="query" use="com.myfoo.controller.SampleController">

    <p:grid>
        <p:rows>
            <p:row>
                Airline Code:
                <p:textbox id="airlinecode"/>
            </p:row>
            <p:row>
                Flightnumber:
                <p:textbox id="flightnumber"/>
            </p:row>
            <p:row>
                Flightdate:
                <p:datebox id="flightdate"/>
            </p:row>
            <p:row>
                <p:button label="Search" id="search"/>
                <p:separator width="5px"/>
            </p:row>
        </p:rows>
    </p:grid>

    <p:listbox width="100%" id="resultlist" mold="paging" rows="21"
               style="font-size: x-small;">
         <p:listhead sizable="true">
                <p:listheader label="Airline Code" sort="auto"
                              style="font-size: x-small;"/>
                <p:listheader label="Flightnumber" sort="auto"
                              style="font-size: x-small;"/>
                <p:listheader label="Flightdate" sort="auto"
                              style="font-size: x-small;"/>
                <p:listheader label="Destination" sort="auto"
                              style="font-size: x-small;"/>

            </p:listhead>

    </p:listbox>
</p:window>
```

As you can see, the `use` attribute of the ZUL page is the link to the `SampleController`. The `SampleController` handles and controls the objects. Let's have a short look at the `SampleController` sample code:

```
    public class SampleController extends Window {

        private Listbox resultlist;
        private Textbox airlinecode;
        private Textbox flightnumber;
        private Datebox flightdate;
        private Button search;

        /**
         * Initialize the page
         */
        public void onCreate() {

            // Components
            resultlist = (Listbox)
             this.getPage().getFellow("query").getFellow("resultlist");
            airlinecode = (Textbox)
             this.getPage().getFellow("query").getFellow("airlinecode");
            flightnumber = (Textbox)
             this.getPage().getFellow("query").getFellow("flightnumber");
            flightdate = (Datebox)
             this.getPage().getFellow("query").getFellow("flightdate");
            search = (Button)
             this.getPage().getFellow("query").getFellow("search");
            search.addEventListener("onClick", new EventListener() {

                public void onEvent(Event event) throws Exception {
                    performSearch();
                }
            });

        }

        /**
         * Execute the search and fill the list
         */
        private void performSearch() {

    //(1)      List<Flight> flightlist = ((FlightService)
                                SpringUtil.getBean("flightService")).
                                getFlightBySearch(airlinecode.getValue(),
                                flightnumber.getValue(),
                                flightdate.getValue(),"");
            resultlist.getItems().clear();

            for (Flight aFlightlist : flightlist) {
                // add flights to list
            }
        }
    }
    /* (1)-shows the integration of the Spring Bean*/
```

Just for completion the context file for Spring is listed here with the bean that is called.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
      "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

    <bean id="txProxyTemplate" abstract="true"
          class="org.springframework.transaction.
                            interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager">
            <ref bean="transactionManager"/>
        </property>
        <property name="transactionAttributes">
            <props>
                <prop key="save*">PROPAGATION_REQUIRED</prop>
                <prop key="add*">PROPAGATION_REQUIRED</prop>
                <prop key="remove*">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>

    <bean id="flightService" parent="txProxyTemplate">
        <property name="target">
            <bean class="com.myfoo.services.impl.FlightServiceImpl">
                <property name="flightDAO">
                    <ref bean="flightDao"/>
                </property>
            </bean>
        </property>
    </bean>

</beans>
```

In short we have learned how to use Spring with ZK and about the configurations. We have seen that the integration is quite smooth and also powerful.

Hibernate

Well, Hibernate has really nothing to do with a view layer so why are we talking here about this issue?

Normally, Hibernate is used along with Spring to take care of the database transactions. In the Spring sample the section above in we have also shown the integration of Hibernate to do the mapping and DAO handling. However, unfortunately there is a downer. Hibernate has a different session management. Unlike the ZK multi-threaded environment, Hibernate handles all data accessing operations in one session. To solve this problem the creators of ZK developed their own ZK listener for Hibernate.

This listener has to be configured in the `zk.xml`:

```
<listener>
    <description>Hibernate thread session context
              handler</description>
    <listener-class>
```

```
        org.zkoss.zkplus.hibernate.
HibernateSessionContextListener
    </listener-class>
</listener>
```

The `<listener>` copies the `sessionMap` to the event thread's `ThreadLocal` variable whenever a new event thread is initiated. So whenever you call `getCurrentSession()` in Hibernate the same `sessionMap` is referenced.

Pay attention of this issue and then Hibernate can be used as described in the Hibernate documentation.

JasperReport

JasperReport is an open-source reporting engine. It can generate print-quality output like PDF, HTML, and RTF. Because there is no communication between the ZK layer and the JasperReport module the integration is quite easy. Just generate the reports (or graphs or PDFs) with JasperReport and show them with ZK tools. A graphic may be loaded dynamically or a PDF can be shown by opening the link.

To use JasperReport use the Java Controller for a ZUL page and use statements like:

```
jasperReport = JasperCompileManager.compileReport(
          "reports/zkbook_demo.jrxml");
jasperPrint = JasperFillManager.fillReport(
          jasperReport, new HashMap(), new
JREmptyDataSource());
JasperExportManager.exportReportToPdfFile(
          jasperPrint, "reports/zkbook.pdf");
```

There is nothing more to configure.

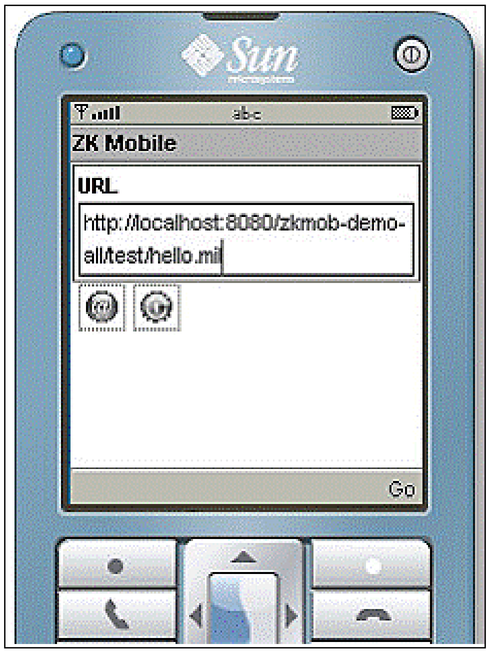
ZK Mobile

Mobile devices have become more and more popular. There are some tools in the market to create mobile websites. ZK also supports the mobile device market with a special ZK Mobile solution. This ZK Mobile client has to be installed into the mobile device and should be made to run as a thin client to interact with a ZK server. However, the solution isn't perfect because the user has to install software on his or her device even if he or she wishes to access only a single website. But the installation process on mobile devices can be automated so it's not that big an issue.

ZK Mobile features 10 out-of-the-box components:

- listbox
- listitem
- textbox
- image
- label
- command
- datebox
- decimalbox
- intbox
- frame

Also, the way of developing ZK applications is similar in ZK Mobile. After the installation of the ZK Mobile client (easy with `zkmob.jad`) any URL pointing to a ZK application may be called. See the following sample picture:



```
http://localhost:8080/zkmob-demo-all/test/hello.mil
```

Instead of using the extension `*.zul` the extension `*.mil` is used for pages. MIL stands for Mobile Interactive Language. In other chapters we have learned that we need a Loader to interpret the ZUL pages and one will also be needed for MIL pages.

The `web.xml` modifications are:

```
<servlet-mapping>
   <servlet-name>zkLoader</servlet-name>
   <url-pattern>*.mil</url-pattern>
</servlet-mapping>

<mime-mapping>
   <extension>jad</extension>
   <mime-type>text/vnd.sun.j2me.app-descriptor</mime-type>
</mime-mapping>

<welcome-file-list>
   <welcome-file>index.mil</welcome-file>
</welcome-file-list>
```

To use ZK Mobile it is necessary to use at least version 3.0.0 of the ZK framework.

ZK JSP Tags Library

ZK as an AJAX framework has its own interpreting and rendering technology. In some cases, however, the developer would like to use JSP pages and add some ZK features. Maybe there is a large application with hundreds of JSP pages that can't migrate so easily. For this purpose ZK provides the ZK JSP tags library.

Actually the ZK JSP tags library consists of two files:

- `zuljsp.jar`
- `zuljsp.tld`

Declare the namespace in the JSP files and start using ZK via tags. Check this `sample.jsp`:

```
<%@ taglib uri="http://www.zkoss.org/2005/zul/jsp" prefix="j" %>
<j:page zscriptLanguage="java">
    <h3>JSP Tag Demo</h3>
   <j:window title="jsp demo" border="normal" width="650px">
       <j:button label="Click Me">
```

```
            <j:attribute name="onClick">{
                      Messagebox.show("It works.", "Information",
                        Messagebox.OK, Messagebox.INFORMATION);
             }
            </j:attribute>
        </j:button>

         <j:vbox id="demo"/>

     </j:window>

   </j:page>
```

ZK JSF Components

To enable features from JSF like Validator and ValueBinding the ZK JSF Components wrap ZK components.

Similarly to with the JSP components, just declare the name space and start using the components, that is:

```
<%@ taglib uri="http://www.zkoss.org/jsf/zul" prefix="z"%>
```

The sample code from the JSP section will look like:

```
<HTML>
<HEAD>
<title>JSF Component Demo</title>
</HEAD>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%@ taglib uri="http://www.zkoss.org/jsf/zul" prefix="z"%>
<f:view>
   <z:page>
      <z:window title="jsf demo" border="normal">
          <z:button label="Click Me">
      <z:attribute name="onClick">{
Messagebox.show("It works.", "Information", Messagebox.OK, Messagebox.
INFORMATION);
          }</z:attribute>

          </z:button>
          <z:vbox id="demo"/>
      </z:window>
   </z:page>
</f:view>
</body>
</HTML>
```

Binding to a Backing Bean

With the EditableValueHolder implementation a component can bind values to and from a backing bean. The input ZK components implement the EditableValueHolder interface and the following table shows the list of components with the, class of binding target and the attribute of the ZK component that will be set in wrapping.

ZK JSF Component	Class of Binding	Attribute of ZK Component
Bandbox	String	value
Calendar	Date	value
Checkbox	Boolean	checked
Combobox	String	value
Datebox	Date	text
Decimalbox	BigDecimal	text
Doublebox	Double	text
Intbox	Integer	text
Listbox	String String[] (multiple selection)	Binding on selection
Radiogroup	String	Binding on selection
Slider	Integer	curpos
Timebox	Date	text
Textbox	String	value
Tree	String String[] (multiple selection)	Binding on selection

ValueBinding of a component

The binding follows typical JSF style. See the following code snippet:

```
<z:window z:title="ValueBinding Demo" width="650px" >
   <z:textbox id="demotextbox"
              f:value="#{ValueBindingBean.text}" /><br/>
   <z:intbox id="demointbox"
             f:value="#{ValueBindingBean.number}" /><br/>
   <z:datebox id="demodatebox" format="dd/MM/yyyy"
              f:value="#{ValueBindingBean.date}" />
   <br/>
   <z:listbox id="demolistbox"
              f:value="#{ValueBindingBean.selection}">
      <z:listitem value="1" label="Text 1" />
```

```
         <z:listitem value="2" label="Text 2" />
         <z:listitem value="3" label="Text 3" />
         <z:listitem value="4" label="Text 4" />
         <z:listitem value="5" label="Text 5" />
         <z:listitem value="6" label="Text 6" />
      </z:listbox>
      <h:commandButton id="submit" action="#{ValueBindingBean.doSubmit}"
         value="Submit" />
   </z:window>
```

Summary

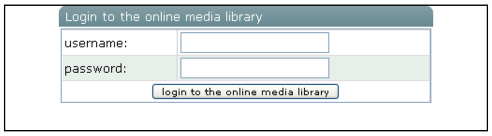
We started this chapter with advantages and disadvantages. Then we saw how to integrate ZK with the Spring Framework and also why it is useful to do so. We then moved on to Hibernate and JasperReport. Then we had a quick brush with ZK Mobile, ZK JSP tags library, and ZK JSF components.

6
Creating Custom Components

After the tutorial, and the integration of some interesting framework into the ZK framework, it's interesting to see how to customize existing components, and to create our own components for the framework, which you could spread around the community.

Cascading Style Sheets (CSS)

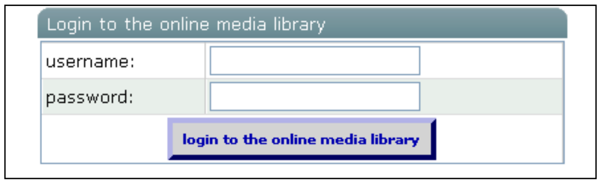
One way to customize components is by the use of **Cascading Style Sheets**. This technique is familiar from HTML, and therefore, it's normal that Potix (the manufacturer of ZK framework) decided to offer that possibility as well.


Login to the online media library
username:
password:
login to the online media library

A possible way to customize a component with a style is by using the `style` attribute of an element. We start by changing the layout of the login button.

```
<button label="${login}" style="border: 4px outset;background-color:
lightgray;padding:5px;color: #0000AF; font-weight:bold">
  <attribute name="onClick">
    login();
  </attribute>
</button>
```

The result is shown as follows:.


Login to the online media library
username:
password:
login to the online media library

Another solution for defining the layout of a control is to externalize the style definition (instead of using the `style` attribute). With this solution, we have the benefit of being able to reuse the style definition. The definitions from the external stylesheet could be used with the `sclass` attribute.

```
<label value="${labelPassword}" sclass="simplelabel" />
```

To provide the definition, we can either embed the style directly into the ZUL page, or include an external CSS file.

External variant:

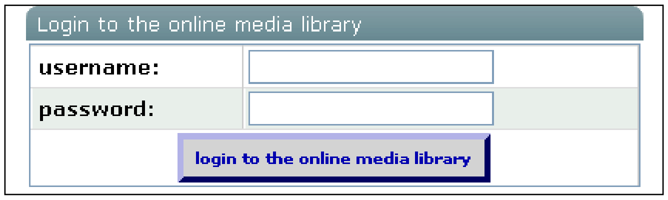
```
<style src="/css/sample-styles.css"/>
.simplelabel
{
  padding:2px;
  color: black;
  font-weight:bold;
}
```

The mentioned path (`/css/sample-styles.css`) is relative to the web application.

Embedded variant:

```
<style>
  mylabel
  {
    padding:2px;
    color: black;
    font-weight:bold;
  }
</style>
```

The changed mask is shown as follows:


Login to the online media library
username:
password:
login to the online media library

It's good to externalize the stylesheets into a file. With this mechanism it's possible to reuse the styles across pages. The disadvantage is that we have to specify the `sclass` attribute every time we want to use it. A more comfortable way is to define our own component, and use it.

```
<?component name="loginLabel" extends="label" sclass="mylabel"?>
<loginLabel value="${labelUsername}" />
```

To use an extended component in the whole application, we have to define a `language-addon`. A `language-addon` allows the usage of a component in the whole application. Such an addon contains a set of components. First we have to provide a file that defines the addons. An example is as follows.

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

<language-addon>
  <addon-name>my-addon</addon-name>

  <language-name>xul/html</language-name>
  <component>
    <component-name>specialLabel</component-name>
    <extends>label</extends>
    <property>
      <property-name>sclass</property-name>
      <property-value>simplelabel</property-value>
    </property>
  </component>
</language-addon>
```

`language-name`	The language of the addon. The possible languages are defined in the `lang.xml` file which is provided from the ZK framework.
`addon-name`	The name of the addon. The addon name is the identifier for the addon. That name must be unique for one web application.
`component-name`	The name of the component. This name is used in the ZUL files to integrate the component.
`extends`	This is same as the `extends` in a Java class. Here, we define the base component.
`property`	Inside property, we define the predefined attributes for the component.

To register a `language-addon`, we have two possibilities. The first one is to place the file into the `metainfo/zk` directory, and name it `language-addon.xml`. Another way is to register the file inside the optional configuration `zk.xml` file. This file has to be located inside the `WEB-INF` directory. The `zk.xml` file to register a `language-addon` is depicted.

```
<?xml version="1.0" encoding="UTF-8"?>
<zk>
  <language-config>
    <addon-uri>/WEB-INF/my-addon.xml</addon-uri>
  </language-config>
</zk>
```

Now we can use the component inside the pages.

```
<specialLabel value="${labelPassword}" />
```

A `language-addon` can do many more things for you. For example, with the `component-class` attribute you can provide your own class for the component. Just look into `lang.xml` file of the `zul.jar` file of the ZK framework.

Macro Components

Beyond the possibility of customizing existing components with the help of CSS files, it's also possible to create new components based on existing ones. In our sample login page, we used a label followed by a textbox component. Now we will replace that component with a component that has a label, and a textbox.

```
<specialLabel value="${labelPassword}" />
<a:bind value="user.password" />
<textbox type="password" id="password" constraint="no empty"/>
```

Now we start creating a macro component, and use that component in the page. The interesting thing about the macro component itself is the usage of annotation-based data binding in the macro component.

The first step is to create the macro ZUL file.

```
<zk xmlns="http://www.zkoss.org/2005/zul"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.zkoss.org/2005/zul ../WEB-
         INF/xsd/zul/zul.xsd"
    xmlns:a="http://www.zkoss.org/2005/zk/annotation">

<?init class="org.zkoss.zkplus.databind.AnnotateDataBinderInit" ?>

  <label value="${arg.labelvalue}" />
  <a:bind value="user.password" />
  <textbox type="password" id="password" constraint="no empty"/>

  <zscript>
    import com.packtpub.zk.media.model.*;
    User user = self.getDynamicProperty(„user");
  </zscript>
</zk>
```

To use data binding based on annotation, we have to declare the `org.zkoss.zkplus.databind.AnnotateDataBinderInit` initialization class. In our example, the component we created is of type `org.zkoss.zk.ui.HtmlMacroComponent`. To initialize the user for the databinding, we have to access the property with the `getDynamicProperty` method.

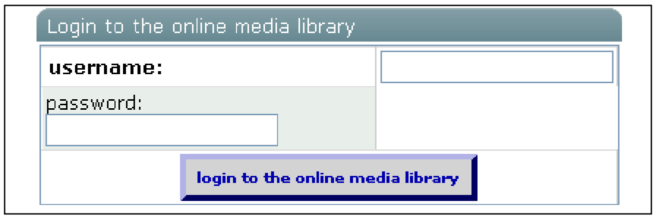
In the ZUL page, we have to define the new component as:

```
<?component name="passwordLine" macro-uri="/WEB-INF/macros/
passwordline.zul"?>
```

Now we can use this component as follows:

```
<passwordLine id="passwordMacro" user="${user}" labelvalue="${labelPa
ssword}"/>
```

If we use the new component in the login page, the screen looks like the following screenshot.


Login to the online media library
username:
password:
login to the online media library

The resulting screen is not as expected. This is because of the row component used. This component creates each component inside a cell (in HTML a `<td>` tag). For the row, our components appear like one component. To solve the problem, we can change the layout of the login page, or extend the row component. However, that's not within the scope of the chapter. Before we start with the implementation of our component, we should look at the actual `login.zul`. The `sample-style.css` is shown as follows.

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

<zk xmlns:html="http://www.w3.org/1999/xhtml"
    xmlns:a="http://www.zkoss.org/2005/zk/annotation">
  <?init class="org.zkoss.zkplus.databind.AnnotateDataBinderInit" ?>
  <style src="/css/sample-styles.css"/>
  <style>
    .mylabel
    {
      padding:2px;
      color: black;
      font-weight:bold;
    }
  </style>

  <zscript>
    import com.packtpub.zk.media.model.*;
  </zscript>

  <?component name="loginLabel" extends="label" sclass="mylabel"?>
  <?component name="passwordLine" macro-uri="/WEB-
      INF/macros/passwordline.zul"?>

  <div align="center">
    <window width="400px" title="Login to the online media library"
```

```
        onOK="login()">
      <grid>
        <rows>
          <zscript>
            String labelUsername = org.zkoss.util.resource.Labels
                       .getLabel("login.username");
            String labelPassword = org.zkoss.util.resource.Labels
                       .getLabel("login.password");
            String login = org.zkoss.util.resource.Labels
                       .getLabel("login.button");
          </zscript>

          <row>
            <loginLabel value="${labelUsername}" />
            <a:bind value="user.name"/>
            <textbox id="username" constraint="no empty"/>
          </row>

          <row>
            <passwordLine id="passwordMacro" user="${user}"
                      labelvalue="${labelPassword}"/>

          </row>

          <row spans="2" align="center">
            <button label="${login}" style="border: 4px outset;
                    background-color:lightgray;padding:5px;
                    color: #0000AF; font-weight:bold">
              <attribute name="onClick">
                login();
              </attribute>
            </button>
          </row>
        </rows>
      </grid>
    </window>
  </div>

  <zscript>
    void login()
    {
      //validate the username
      username.getValue();
      //validate the password
```

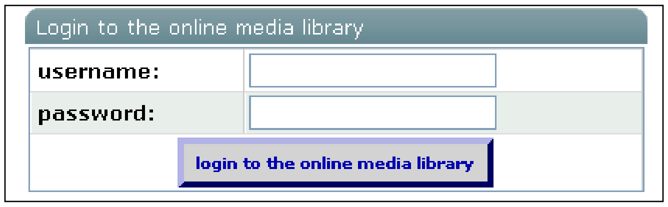
```
            password.getValue();
            UserFactory.login(user);
            Executions.sendRedirect("index.zul");
          }
        </zscript>

        <!-- Initialize the model objects for using with anotations -->
        <zscript>
          com.packtpub.zk.media.model.User user = com.packtpub.zk.media
                       .model.UserFactory.create();
        </zscript>
      </zk>
```

Creating Our Own Component

After we have customized the layout and the styles, and created our own components based on existing components it's time to create a custom component. In application development, it should be rare to create a custom component, but to have a complete understanding of the framework it could be important to know how a component is created.

In the example `login.zul`, we use a window component with a title ('**Login to the online media library**').


Login to the online media library
username:
password:
login to the online media library

Now, we try to create our own window component. The first step is to define a `language-addon`.

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

  <language-addon>
    <addon-name>my-window</addon-name>
    <language-name>xul/html</language-name>
    <component>
      <component-name>mywindow</component-name>
      <component-class>org.zkoss.zul.Window</component-class>
      <mold>
```

```
          <mold-name>default</mold-name>
          <mold-uri>my-window.dsp</mold-uri>
        </mold>
      </component>
    </language-addon>
```

In the `language-addon`, we use the original class `org.zkoss.zul.Window`. An exciting part of the `language-addon` definition is `mold`. Here, we define our own `.dsp` file which is responsible for rendering the component as HTML.

This is the original `window.dsp` in the directory `web/zul/html` of the archive `zul.jar`:

```
<%--
window.dsp
{
  {
    IS_NOTE
    Purpose:
    Description:
    z.autoz:
    Automatically adjust z-index onmousedown (au.js)
    History:
    Tue May 31 19:37:23     2005, Created by tomyeh
  }
}
IS_NOTE
Copyright (C) 2005 Potix Corporation. All Rights Reserved.
{
  {
    IS_RIGHT
    This program is distributed under GPL Version 2.0 in the hope that
    it will be useful, but WITHOUT ANY WARRANTY.
  }
}
IS_RIGHT
--%>
<%@ taglib uri="/WEB-INF/tld/web/core.dsp.tld" prefix="c" %>
<%@ taglib uri="/WEB-INF/tld/zk/core.dsp.tld" prefix="z" %>
<c:set var="self" value="${requestScope.arg.self}"/>
<c:set var="titlesc" value="${self.titleSclass}"/>
<div id="${self.uuid}" z.type="zul.wnd.Wnd"
     z.autoz="true"${self.outerAttrs}${self.innerAttrs}>
```

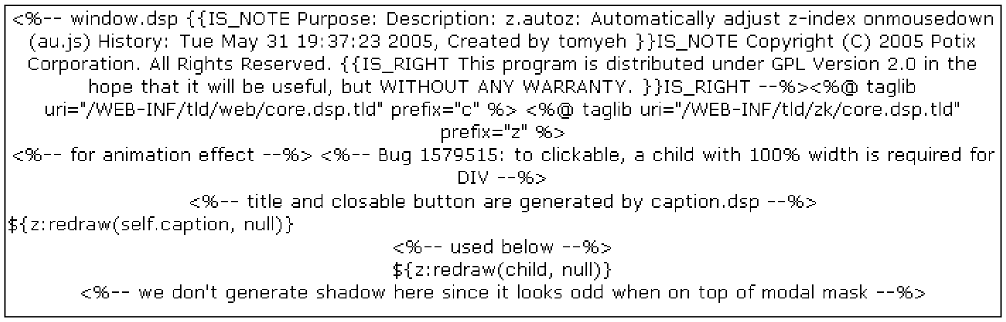
```
<div>
<%-- for animation effect --%>
  <c:choose>
    <c:when test="${empty self.caption and empty self.title}">
      <c:if test="${c:isExplorer() and !c:isExplorer7()}">
      <%-- Bug 1579515: to clickable, a child with 100% width is
                 required for DIV --%>
        <table width="100%" border="0" cellpadding="0"
                                       cellspacing="0">
          <tr height="1px"><td></td></tr>
        </table>
      </c:if>
    </c:when>
    <c:otherwise>
      <table width="100%" border="0" cellpadding="0"
                                     cellspacing="0">
        <c:choose>
          <c:when test="${empty self.caption}">
            <tr id="${self.uuid}!caption" class="title">
              <td class="l${titlesc}"></td>
              <td class="m${titlesc}">
                <c:out value="${self.title}"/>
              </td>
              <c:if test="${self.closable}">
                <td width="16" class="m${titlesc}">
                  <img id="${self.uuid}!close" src="${c:encodeURL
                         ('~./zul/img/close-off.gif')}"/>
                </td>
              </c:if>
              <td class="r${titlesc}"></td>
            </tr>
          </c:when>
          <c:otherwise>
            <tr id="${self.uuid}!caption">
            <%-- title and closable button are generated by
                                           caption.dsp --%>
              <td class="l${titlesc}"></td>
              <td class="m${titlesc}">
                ${z:redraw(self.caption, null)}
              </td>
              <td class="r${titlesc}"></td>
            </tr>
          </c:otherwise>
        </c:choose>
      </table>
      <c:set var="wcExtStyle" value="border-top:0;"/>
      <%-- used below --%>
    </c:otherwise>
  </c:choose>
```

```
    <c:set var="wcExtStyle" value="${c:cat(wcExtStyle,
           self.contentStyle)}"/>
      <div id="${self.uuid}!cave" class="${self.contentSclass}"$
             {c:attr('style',wcExtStyle)}>
        <c:forEach var="child" items="${self.children}">
        <c:if test="${self.caption != child}">
          ${z:redraw(child, null)}</c:if>
        </c:forEach>
      </div>
      <%-- we don't generate shadow here since it looks odd when on
                                    top of modal mask --%>
    </div>
  </div>
```

Now, we can use the new component in our `login.zul`.

```
<mywindow width="400px" title="Login to the online media library"
          onOK="login()">
```

Now, if we call the changed `login.zul`, we get the following output.



```
<%-- window.dsp {{IS_NOTE Purpose: Description: z.autoz: Automatically adjust z-index onmousedown
(au.js) History: Tue May 31 19:37:23 2005, Created by tomyeh }}IS_NOTE Copyright (C) 2005 Potix
Corporation. All Rights Reserved. {{IS_RIGHT This program is distributed under GPL Version 2.0 in the
hope that it will be useful, but WITHOUT ANY WARRANTY. }}IS_RIGHT --%><%@ taglib
uri="/WEB-INF/tld/web/core.dsp.tld" prefix="c" %> <%@ taglib uri="/WEB-INF/tld/zk/core.dsp.tld"
prefix="z" %>
<%-- for animation effect --%> <%-- Bug 1579515: to clickable, a child with 100% width is required for
DIV --%>
<%-- title and closable button are generated by caption.dsp --%>
${z:redraw(self.caption, null)}
<%-- used below --%>
${z:redraw(child, null)}
<%-- we don't generate shadow here since it looks odd when on top of modal mask --%>
```

This isn't the expected result. The problem here is that the `.dsp` file should be parsed. To do that we have to register the `org.zkoss.web.servlet.dsp.InterpreterServlet` in the `web.xml`.

```
<servlet>
  <servlet-name>dspLoader</servlet-name>
  <servlet-class>
    org.zkoss.web.servlet.dsp.InterpreterServlet
  </servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>dspLoader</servlet-name>
  <url-pattern>*.dsp</url-pattern>
</servlet-mapping>
```

A `.dsp` file is not a JSP page. However, we can use JSP syntax and tags inside such a `.dsp` file. For completion, we show the complete `web.xml` of the application. A custom component renders its dynamic HTML code based on the XML attributes and the content of the control in a JSP fashion.

```
<?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[A online media library]]></description>
  <display-name>zkmedialib</display-name>

  <!-- ZK -->
  <listener>
    <description>
      Used to cleanup when a session is destroyed</description>
    <display-name>ZK Session Cleaner</display-name>
    <listener-class>
      org.zkoss.zk.ui.http.HttpSessionListener
    </listener-class>
  </listener>

  <filter>
    <filter-name>SecurityFilter</filter-name>
    <filter-class>
      com.packtpub.zk.media.controller.SecurityFilter
    </filter-class>
    <init-param>
      <param-name>loginPage</param-name>
      <param-value>login.zul</param-value>
    </init-param>
  </filter>

  <filter-mapping>
    <filter-name>SecurityFilter</filter-name>
    <url-pattern>*.zul</url-pattern>
  </filter-mapping>

  <servlet>
    <description>ZK loader for ZUML pages</description>
    <servlet-name>zkLoader</servlet-name>
    <servlet-class>
      org.zkoss.zk.ui.http.DHtmlLayoutServlet
    </servlet-class>
    <!-- Required. Specifies URI of the update engine
        (DHtmlUpdateServlet).It must be the same as <url-pattern>
        for the update engine.-->
    <init-param>
      <param-name>update-uri</param-name>
      <param-value>/zkau</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet>
    <description>
      The asynchronous update engine for ZK
    </description>
    <servlet-name>auEngine</servlet-name>
    <servlet-class>
      org.zkoss.zk.au.http.DHtmlUpdateServlet</servlet-class>
  </servlet>

  <servlet>
    <servlet-name>dspLoader</servlet-name>
    <servlet-class>
      org.zkoss.web.servlet.dsp.InterpreterServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>zkLoader</servlet-name>
    <url-pattern>*.zul</url-pattern>
  </servlet-mapping>

  <servlet-mapping>
    <servlet-name>zkLoader</servlet-name>
    <url-pattern>*.zhtml</url-pattern>
  </servlet-mapping>

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

```
<servlet-mapping>
  <servlet-name>auEngine</servlet-name>
  <url-pattern>/zkau/*</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>dspLoader</servlet-name>
  <url-pattern>*.dsp</url-pattern>
</servlet-mapping>

<!-- Miscellaneous -->
<session-config>
  <session-timeout>120</session-timeout>
</session-config>

<!-- MIME mapping -->
<mime-mapping>
  <extension>doc</extension>
  <mime-type>application/vnd.ms-word</mime-type>
</mime-mapping>

<mime-mapping>
  <extension>gif</extension>
  <mime-type>image/gif</mime-type>
</mime-mapping>

<mime-mapping>
  <extension>htm</extension>
  <mime-type>text/html</mime-type>
</mime-mapping>

<mime-mapping>
  <extension>html</extension>
  <mime-type>text/html</mime-type>
</mime-mapping>

<mime-mapping>
  <extension>jad</extension>
  <mime-type>text/vnd.sun.j2me.app-descriptor</mime-type>
</mime-mapping>

<mime-mapping>
  <extension>jnlp</extension>
  <mime-type>application/x-java-jnlp-file</mime-type>
```

```
    </mime-mapping>

    <mime-mapping>
      <extension>jpeg</extension>
      <mime-type>image/jpeg</mime-type>
    </mime-mapping>

    <mime-mapping>
      <extension>jpg</extension>
      <mime-type>image/jpeg</mime-type>
    </mime-mapping>

    <mime-mapping>
      <extension>js</extension>
      <mime-type>application/x-javascript</mime-type>
    </mime-mapping>

    <mime-mapping>
      <extension>pdf</extension>
      <mime-type>application/pdf</mime-type>
    </mime-mapping>

    <mime-mapping>
      <extension>png</extension>
      <mime-type>image/png</mime-type>
    </mime-mapping>

    <mime-mapping>
      <extension>txt</extension>
      <mime-type>text/plain</mime-type>
    </mime-mapping>

    <mime-mapping>
      <extension>xls</extension>
      <mime-type>application/vnd.ms-excel</mime-type>
    </mime-mapping>

    <mime-mapping>
      <extension>xml</extension>
      <mime-type>text/xml</mime-type>
    </mime-mapping>

    <mime-mapping>
      <extension>xul</extension>
      <mime-type>application/vnd.mozilla.xul-xml</mime-type>
    </mime-mapping>

    <mime-mapping>
      <extension>zhtml</extension>
      <mime-type>text/html</mime-type>
    </mime-mapping>

    <mime-mapping>
      <extension>zip</extension>
      <mime-type>application/x-zip</mime-type>
    </mime-mapping>

    <mime-mapping>
      <extension>zul</extension>
      <mime-type>text/html</mime-type>
    </mime-mapping>

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

If we want to provide just a new mold to a component, it's possible to declare a `mold-uri` in the component tag. A new mold means a new `.dsp` file.

Summary

In this chapter, we have learned how to customize existing components. We have the ability to use styles which are known from HTML to change the layout of the components. Additionally we have seen that it is possible to build new components (macro components) on the basis of existing components. And last but not least, in the last section of this chapter we have learned how to create a complete custom component that is based on a `.dsp` file.

Our advice is first to try to build on an existing component, instead of creating a complete component of your own. Use object-oriented inheritance, not code duplication, which would loose advantage of error corrections in newer versions.

7
Development Tools for the ZK Framework

This chapter discusses some IDE tools for working with ZK. Definitely for effective development with a framework like ZK it is a must to have some IDE support. Right now there is a tool called 'ZeroKode' which recently became an open source project. With ZeroKode it is possible to design a ZUL page. Another tool is 'zk-bench', which is designed as an Eclipse plugin. In the next sections, we discuss the early beta version of zk-bench, which will be provided as a commercial product. Zk-bench is very promising because it gives the developer much more than just 'page designing'.

ZK-Bench

Zk-bench is designed as an Eclipse plug-in and the installation is pretty easy. There is an update site available as well as an installer for download. Further details are available from `http://www.ir-team.com/index.php/zk-bench`. The company behind zk-bench even offers a forum to discuss problems and features.

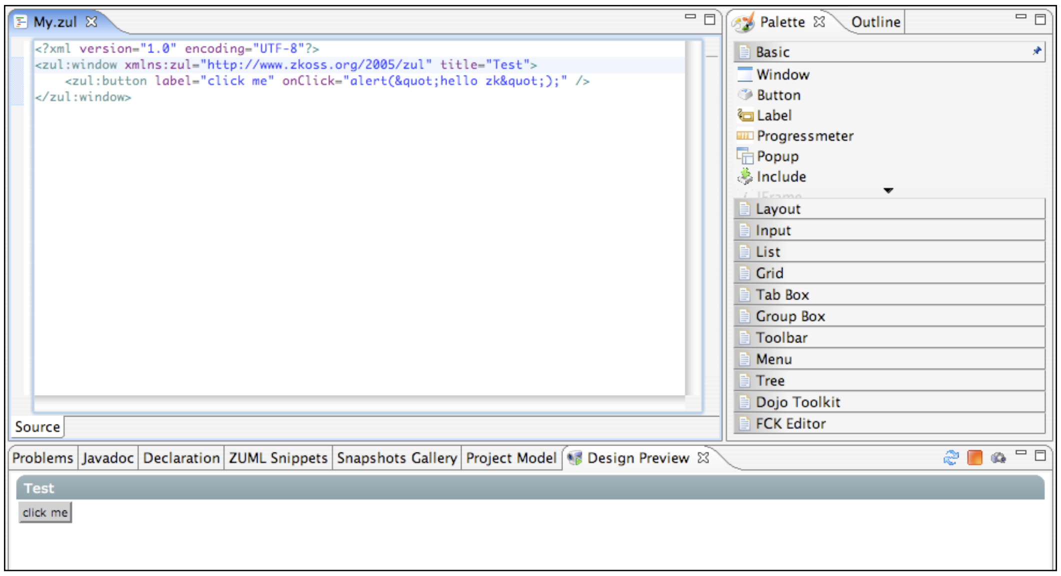
The installer asks a few options but it runs very smoothly. After the installation we find a new zk-bench project type available. Zk-bench provides a visual designer for ZUL pages, event handling, and property changes within the IDE, a structural pattern, and persistence support.

The zk-bench project follows the Maven 2.x directory structure, but that is not visible for the developer, at least not at the first step of work. Maven-specific items like `artifactId` and `groupId` are wrapped in more easily understood labels. The default view of the project is a kind of meta-view. So there are folders for images, pages, Java classes, and the project structure looks very clean even for big projects. The zk-bench project opens a perspective with a graphical editor, the Meta-project explorer, Snippet views, snapshot gallery, and a palette for the ZK components.

With the palette and the visual designer it's just a breeze to design a web page. The selection of a component shows all the properties and it's even possible to manage the events very intuitively. With a click on the component the property menu appears and there is an entry for the action method. Developers who knows Swing Development will feel very comfortable with this kind of design. The result of the design work is a ZUL page and the related Java class.

Dynamic Preview of Pages

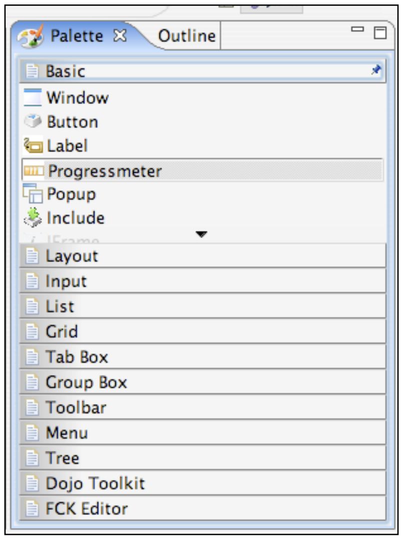
During the development of the visual ZUL page there is an option to show the preview of the page. The preview is done in real time so it's nice to see the result of the design process all the time.



```
<?xml version="1.0" encoding="UTF-8"?>
<zul:window xmlns:zul="http://www.zkoss.org/2005/zul" title="Test">
    <zul:button label="click me" onClick="alert(&quot;hello zk&quot;);" />
</zul:window>
```

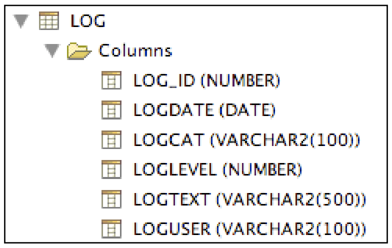
The Palette

The visual designer provides a palette for all ZK components. The components can be moved from the palette into the designer frame with drag-and-drop. The related palette for a specific ZK version is dynamically loaded. The ZK version is managed in the preferences and zk-bench will dynamically recognize the components.



Databinding

Zk-bench has its own database explorer and to create a link between a database column and a ZK component we can just drag-and-drop from the database column onto the component in the visual designer.

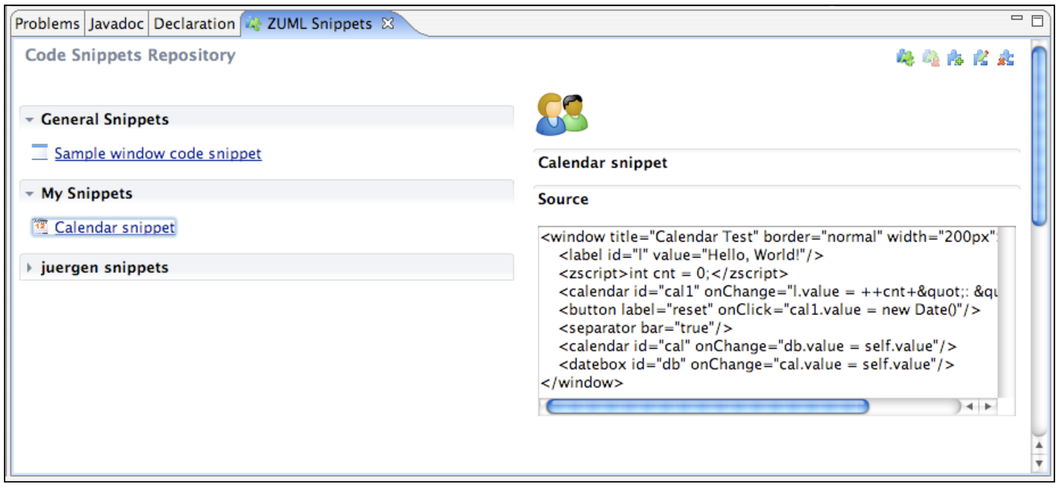


Deployment

While creating the project the target package type will be specified. In the case of a WAR type, zk-bench will create the WAR package automatically. Internally Maven 2 is used along with IRT—the developers of the zk-bench plug-in plan to support and ANT as well.

Snippets

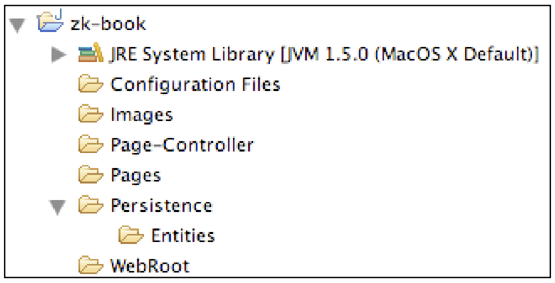
Zk-bench supports an personal snippet repository. It can be used for personal code fragments as well as to share snippets with other users.



```
<window title="Calendar Test" border="normal" width="200px"
    <label id="l" value="Hello, World!"/>
    <zscript>int cnt = 0;</zscript>
    <calendar id="cal1" onChange="l.value = ++cnt+&quot;: &qu
    <button label="reset" onClick="cal1.value = new Date()"/>
    <separator bar="true"/>
    <calendar id="cal" onChange="db.value = self.value"/>
    <datebox id="db" onChange="cal.value = self.value"/>
</window>
```

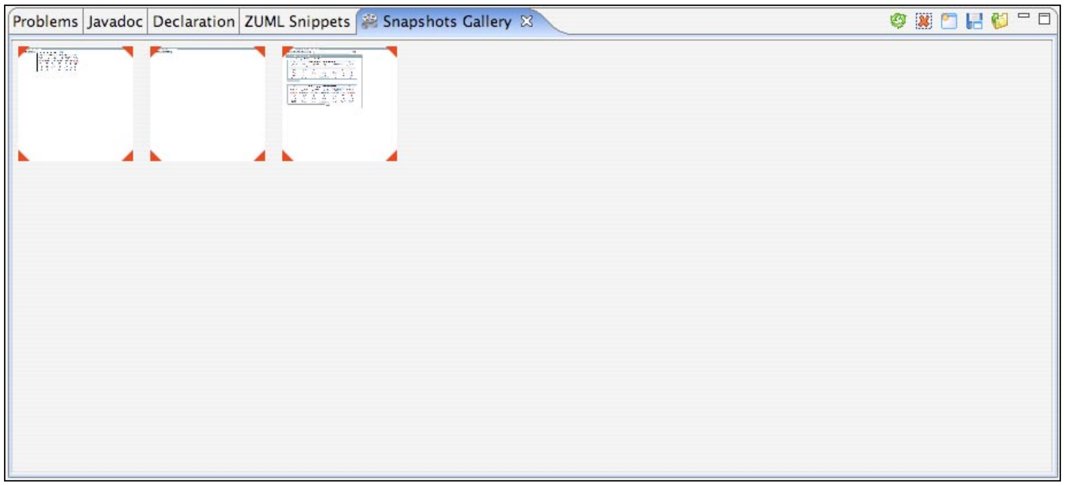
Project Explorer

Zk-bench provides its own way to find project files. The following picture shows a separate folder for Page-Controller. Folders are automatically created by zk-bench during the design of ZUL pages.



Snapshot Gallery

With a simple click on an icon a snapshot from the actual view in the preview pane will be created. That is a very useful feature especially in projects with a large number of pages.



Summary

Zk-bench is a very useful tool and it supports much more than just designing ZUL pages. It simplifies the development of web application a lot and it is built around the ZK framework. By the time this book is written IRT plan—the EAP for the 16th, January 2008 and more features will be available. It will be a pleasure to use such a tool.

A
Configuration Files in ZK

The following tables contain information about the configuration files, which are important in the context of a ZK application. You should use this appendix in conjunction with Chapter 1 to get a better understanding of how to configure your ZK application.

WEB.XML

`org.zkoss.zk.ui.http.DHtmlLayoutServlet`	ZKLoader
`org.zkoss.zk.au.http.DHtmlUpdateServlet`	ZK AU Engine
`org.zkoss.zk.ui.http.HttpSessionListener`	Session Cleaner
`org.zkoss.zk.ui.http.DHtmlLayoutFilter`	Post process filter

zk.xml

Tag	**Default**
`<richlet>`	-
`<listener>`	-
`<log>`	-
`<desktop-config>`	-
`... <desktop-timeout>`	3600
`... <file-check-period>`	5
`... <processing-prompt-delay>`	900 [in milliseconds]
`... <tooltip-delay>`	800 [in milliseconds]

Tag	Default
`... <theme-uri>`	-
`... <disable-default-theme>`	-
`... <el-config>`	-
`... ... <evaluator-class>`	`org.apache.commons.el.ExpressionEvaluatorImpl`
`... <language-config>`	-
`... ... <addon-uri>`	-
`... <session-config>`	-
`... ... <timeout-uri>`	-
`... ... <session-timeout>`	[see Webserver]
`... ... <max-desktops-per-session>`	10
`... <system-config>`	-
`... ... <max-events-threads>`	100
`... ... <max-upload-size>`	5120 [in KB]
`... ... <response-charset>`	UTF-8
`... ... <locale-provider-class>`	-
`... ... <time-zone-provider-class>`	-
`... ... <cache-provider-class>`	`org.zkoss.zk.ui.impl.SessionDesktopCacheProvider`
`... ... <ui-factory-class>`	`org.zkoss.zk.ui.http.SimpleUiFactory`
`... ... <engine-class>`	`org.zkoss.zk.ui.impl.UiEngineImpl`
`... ... <web-app-class>`	`org.zkoss.zk.ui.http.SimpleWebApp`
`... <zscript-config>`	-
`... ... <zscript-language>`	-
`... ... ...<language-name>`	-
`... ... ... <interpreter-class>`	-
`... <error-page>`	-
`... <preference>`	-
`... ... <name>`	-
`... ... <value>`	-

Configuration of ZK.XML

<richlet> Tag

The richlet tag declares a richlet. It uses two parameters, `<richlet-class>` and the `<richlet-uri>`. For example:

```
<richlet>
  <richlet-class>com.irteam.TestRichlet</richlet-class>
  <richlet-url>/test</richlet-url>
</richlet>
```

The richlet class must implement the `org.zkoss.zk.ui.Richlet` class. Be aware of the `richlet-url` because the "/" is necessary.

<listener> Tag

More than one listener may be specified in `zk.xml`. The listener may implement various interfaces. The possible interfaces are:

`org.zkoss.zk.ui.event.EventThreadInit`	Initialize an event processing thread before an event is dispatched.
`org.zkoss.zk.ui.event.EventThreadCleanup`	Clean up an event thread after execution.
`org.zkoss.zk.ui.event.EventThreadSuspend`	Called before an event is going to be suspended.
`org.zkoss.zk.ui.event.EventThreadResume`	Called when an event is resumed or aborted.
`org.zkoss.zk.ui.util.WebAppInit`	Called when a ZK application is initialized.
`org.zkoss.zk.ui.util.WebAppCleanup`	Clean up a ZK application, which is destroyed.
`org.zkoss.zk.ui.util.SessionInit`	Invoked when a new Session is initialized.
`org.zkoss.zk.ui.util.SessionCleanup`	Invoked to clean up a session that is destroyed.
`org.zkoss.zk.ui.util.DesktopInit`	Initialize when a new desktop is created.
`org.zkoss.zk.ui.util.DesktopCleanup`	To clean up a desktop that is destroyed.
`org.zkoss.zk.ui.util.ExecutionInit`	Called when a new execution is initialized.

`org.zkoss.zk.ui.util.ExecutionCleanup`	Clean up an execution which is destroyed.
`org.zkoss.zk.ui.util.URIInterceptor`	This is used to intercept the loading of ZUML pages with the URI. It may be used to confirm if the current user should have access to the URI.
`org.zkoss.zk.ui.util.Monitor`	This is used to monitor the status of the ZK application.

<log> Tag

Normally, the logging will be performed according to the settings in the web container. This log tag can overwrite the existing settings. If the whole package needs logging then use this tag.

```
<log>
  <log-base></log-base>
</log>
```

Or use

```
<log>
  <log-base>com.irteam.myapp</log-base>
</log>
```

if only the package's com.irteam.myapp.* needs logging.

<desktop-config> Tag

This has the following possible child tags: `theme-uri`, `disable-default-theme`, `desktop-timeout`, `file-check-period`, `tooltip-delay`, and `processing-prompt-delay`. The following list describes these child tags.

1. **<desktop-timeout>**

   This is the time between client requests, when the desktop will be timed out by the server. A negative value will never time out.

2. **<disable-default-theme>**

   This defines the component set whose default theme will be disabled.

3. **<file-check-period>**

   This is the waiting time before a file is checked for whether it is modified.

4. **<processing-prompt-delay>**

   This is the waiting time before a prompt will be displayed.

5. **<tooltip-delay>**

   This is the waiting time before a tool tip will be displayed.

6. **<theme-uri>**

   This defines an additional URI for style sheets.

<el-config> Tag

This tag has one child, the `<evaluator-class>`, and it specifies the class used to evaluate the EL expressions.

<language-config> Tag

This tag has one child, the `<addon-uri>` element. It defines the URI of language add-ons. It is used to add new components. This is explained in some of the later chapters in more detail.

<session-config> Tag

This element has three child tags: `<timeout-uri>`, `<session-timeout>`, and `<max-desktops-per-session>`.

1. **<timeout-uri>**

   This defines the URI the user will be redirected to, when a session is timed out.

2. **<session-timeout>**

   This defines the time between client requests when the session will be timed out by the server.

3. **<max-desktops-per-session>**

   This defines the maximum number of desktops within a session.

<system-config> Tag

This has nine child tags: `<max-event-threads>`, `<max-upload-size>`, `<response-charset>`, `<locale-provider-class>`, `<time-zone-provider-class>`, `<cache-provider-class>`, `<ui-factory-class>`, `<engine-class>`, and `<web-app-class>`.

1. **<max-event-threads>**

   This specifies the maximum allowed event handling threads.

2. **<max-upload-size>**

   This specifies the maximum size of a file that can be uploaded by a client.

3. **<response-charset>**

   This defines the charset that is used for rendering the ZUML pages.

4. **<locale-provider-class>**

   This defines the class that is used to determine the locale. The class must implement the `org.zkoss.zk.ui.sys.LocaleProvider` interface.

5. **<time-zone-provider-class>**

   This defines the class that is used to determine the time zone. The class must implement the `org.zkoss.zk.ui.sys.TimeZoneProvider` interface.

6. **<cache-provider-class>**

   This defines the class that is used for the desktop cache. The class must implement the `org.zkoss.zk.ui.sys.DesktopCacheProvider` interface. Right now (zk 2.3.0) there are two implementations available:

   - `org.zkoss.zk.ui.impl.SessionDesktopCacheProvider`
     All desktops from a session are stored in a single cache so clustering is not possible.
   - `org.zkoss.zk.ui.impl.GlobalDesktopCacheProvider`
     All desktops from a zk application are stored in a single cache.

7. **<ui-factory-class>**

   This defines the class that creates desktops and pages. It also maps URLs to pages. The class must implement the `org.zkoss.zk.ui.sys.UiFactory` interface. Right now (zk 2.3.0) there are two implementations available:

   - `org.zkoss.zk.ui.http.SimpleUiFactory`
     The generated sessions are not serializeable.
   - `org.zkoss.zk.ui.http.SerializableUiFactory`
     Here the sessions are serializable.

   The sessions can be restored after a restart of the web container.

8. **<engine-class>**

   This defines the class that implements the UI engine. This class must implement the `org.zkoss.zk.ui.sys.UiEngine` interface.

9. **<webb-app-class>**

   This implements the web application. It can be extended by `org.zkoss.zk.ui.impl.AbstractWebApp` or `org.zkoss.zk.ui.http.SimpleWebApp`.

<zscript-config> Tag

This defines the language and the implementation class.

```
<zscript-config>
  <zscript-language>
    <language-name>ZKJava</language-name>
    <interpreter-class>com.irteam.ZKJavaInterpreter</interpreter-
class>
  </zscript-language>
</zscript-config>
```

<error-page> Tag

This defines a URI for a specific uncaught exception.

```
<error-page>
  <exception-type>com.irteam.Special</exception-type>
  <location>/error/special.zul</location>
</error-page>
```

<preference> Tag

Any pair of name / value may be defined.

```
<preference>
  <name>com.irteam.preference.AdminAccount</name>
  <value>Administrator</value>
</preference>
```

The values can be read by the `getPreference` method of the `org.zkoss.zk.ui.util.Configuration` class.

Index