

RichFaces Photo Album Application Guide



This documentation is work in progress, thus some mistakes or incompleteness is possible

1. Introduction	1
2. Getting started	3
2.1. Environment Configuration	3
2.2. Installation	3
2.3. Running Functional(Selenium) Tests TBR	4
2.4. Context Help	5
3. Application Overview TBR	7
3.1. Page flows	7
3.1.1. Page flows: implementation details	9
3.1.2. Data Model	10
4. How it works	13
4.1. Used Components	13
4.2. Skinnability	14
4.3. Navigation tree	15
4.3.1. Navigation tree for a guest	16
4.3.2. Navigation tree for a registered user	18
4.4. Album View	23
4.4.1. Image Size Control with <rich:inputNumberSlider>	24
4.4.2. Slideshow	27
4.5. Image View	32
4.5.1. Custom images scroller widget	34
4.6. Upload Images	37
4.7. Context Menus TBR	39
4.8. ToolTips TBR	41
4.9. User Input Data Validation	44
4.10. How the button is created and how it acts	46
4.11. The <a4j:status> component	50
4.12. Errors Reports	52

Introduction

The "Photo Album" web application is a desktop-like on-line photo manager. It provides social services for uploading photos, storing and previewing them, creating your own albums and sharing them with other users, searching albums, photos or users, managing the environment. Newcomers or not registered users can navigate through other users' shelves and albums in "Anonymous mode", but in this case no possibilities to create personal shelves or albums or upload photos are available.

The "Photo Album" represents and implements strict "Shelves - Albums - Photos" hierarchy. It means that photos are kept in albums and albums are stored on shelves, but albums can not contain another albums or shelves and shelves can not keep another shelves and can not be stored in albums.

The Photo Album web application is designed and developed with RichFaces and by RichFaces team. This application demonstrates:

- *wide variety of UI components* - the RichFaces provides a Lego-like way of building user interfaces for web applications;
- *Built-in Ajax capability* - the RichFaces offers both component-wide and very flexible page-wide Ajax support with no need to write any JavaScript code;
- *Highly customizable look-and-feel* - the RichFaces have special feature called [Skinnability](#) [../devguide/html_single/index.html/#Skinnability].

The Photo Album application also encompasses technologies and frameworks such as:

- [Facelets](https://facelets.dev.java.net/) [https://facelets.dev.java.net/]
- [Enterprise Java Beans \(EJB\) 3.0](http://java.sun.com/products/ejb/) [http://java.sun.com/products/ejb/]
- [Seam framework](http://seamframework.org/) [http://seamframework.org/]

The application is available online at [Photo Album](http://livedemo.exadel.com/photoalbum/) [http://livedemo.exadel.com/photoalbum/] page.

Getting started

2.1. Environment Configuration

In order to download, build, modify, and deploy the Photo Album application you need to have the following installed and configured:

- *JDK 1.5 and higher* [<http://java.sun.com/javase/downloads/index.jsp>]
- *Maven 2.0.10* [<http://maven.apache.org/download.html>]
- *JBoss Tools* [<http://www.jboss.org/tools>] (or *Eclipse* [<http://www.eclipse.org/>])
- *JBoss Server* [<http://www.jboss.org/jbossas/downloads/>] (4.2.3.GA, 5.0.x.GA)
- *SVN client* [<http://subversion.tigris.org/>]

2.2. Installation

Once you have configured the environment you can install the Photo Album application:

- *Checkout the project*

```
svn co http://anonsvn.jboss.org/repos/richfaces/trunk/examples/photoalbum/
```

- *Build the project.* Open command line console, point to the folder with checkouted project, and tell Maven:

```
mvn clean install eclipse:eclipse -Dwtpversion=1.5
```

- *Import the project into Eclipse IDE.* You can just deploy Photo Album application onto the server, but the convenient way is to import the project into your IDE. We recommend Eclipse with JBoss Tools since this bundle is more preferable to ensure rapid development process with Seam and RichFaces. You can find a step-by-step tutorial "Importing existing projects" at the [Eclipse documentation](http://help.eclipse.org/ganymede/topic/org.eclipse.platform.doc.user/tasks/tasks-importproject.htm) [<http://help.eclipse.org/ganymede/topic/org.eclipse.platform.doc.user/tasks/tasks-importproject.htm>] page or simply in the *Help > Help Contents* of the Eclipse. As the result three modules of Photo Album project appear in the Workspace:

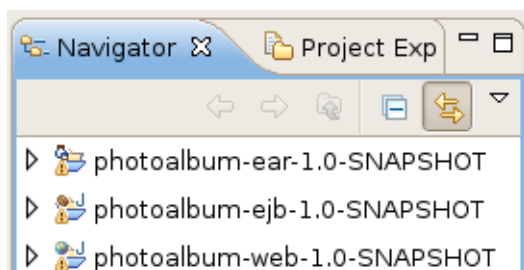


Figure 2.1. Modules of Photo Album in the Workspace

- *Add JBoss AS.* Now you need to add JBoss Application Server runtime. Detailed instructions on how you can do it are given in the [Runtimes and Servers in the JBoss AS plugin](http://download.jboss.org/jbosstools/nightly-docs/en/as/html/runtimes_servers.html) [http://download.jboss.org/jbosstools/nightly-docs/en/as/html/runtimes_servers.html] chapter from the [JBoss Server Manager Reference Guide](http://download.jboss.org/jbosstools/nightly-docs/en/as/html/index.html) [http://download.jboss.org/jbosstools/nightly-docs/en/as/html/index.html].
- *Run photoalbum-ear-1.0-SNAPSHOT* on the JBoss Application Server you have just installed.
- Browse to `http://localhost:8080/photoalbum`.

Tip:

By default Photo Album is assembled with a limited set of images (4-5 in each album). In order to build the version of the application with a full set of images you need to use `livedemo` profile while building Photo Album like this:

```
mvn clean install -Plivedemo
```

2.3. Running Functional(Selenium) Tests TBR

Before starting Selenium test please make sure that you have Firefox browser installed on your local machine, as the Photo Album application is designed to be deployed and run on JBoss Application server, so please make sure that the `<jboss.installer.url>` property of the project `pom.xml` (`examples/photoalbum/`) points to an existing JBoss Application server copy.

You also need to build the Photo Album project in `inexamples/photoalbum/reource`.

Then, you need to go to the test folder of the project (`examples/photoalbum/test/`) and run the

```
...
mvn clean integration-test
```

...

By default Selenium tests are executed in the Firefox browser, hence you need to have it installed. If you configured everything like it is said above you will see tests being executed in the Firefox browser. When the tests are finished you can read test reports in the `examples/photoalbum/tests/target/surefire-reports/` folder.

2.4. Context Help

The Photo Album application was developed in the first place to demonstrate the mighty power of RichFaces thus most of UI elements in the application has a context help article that tells how a particular element works, providing technical details about it. A context help article is displayed when you click on the question mark icon ().

Application Overview TBR

3.1. Page flows

The page flow of the application is illustrated in the diagram.

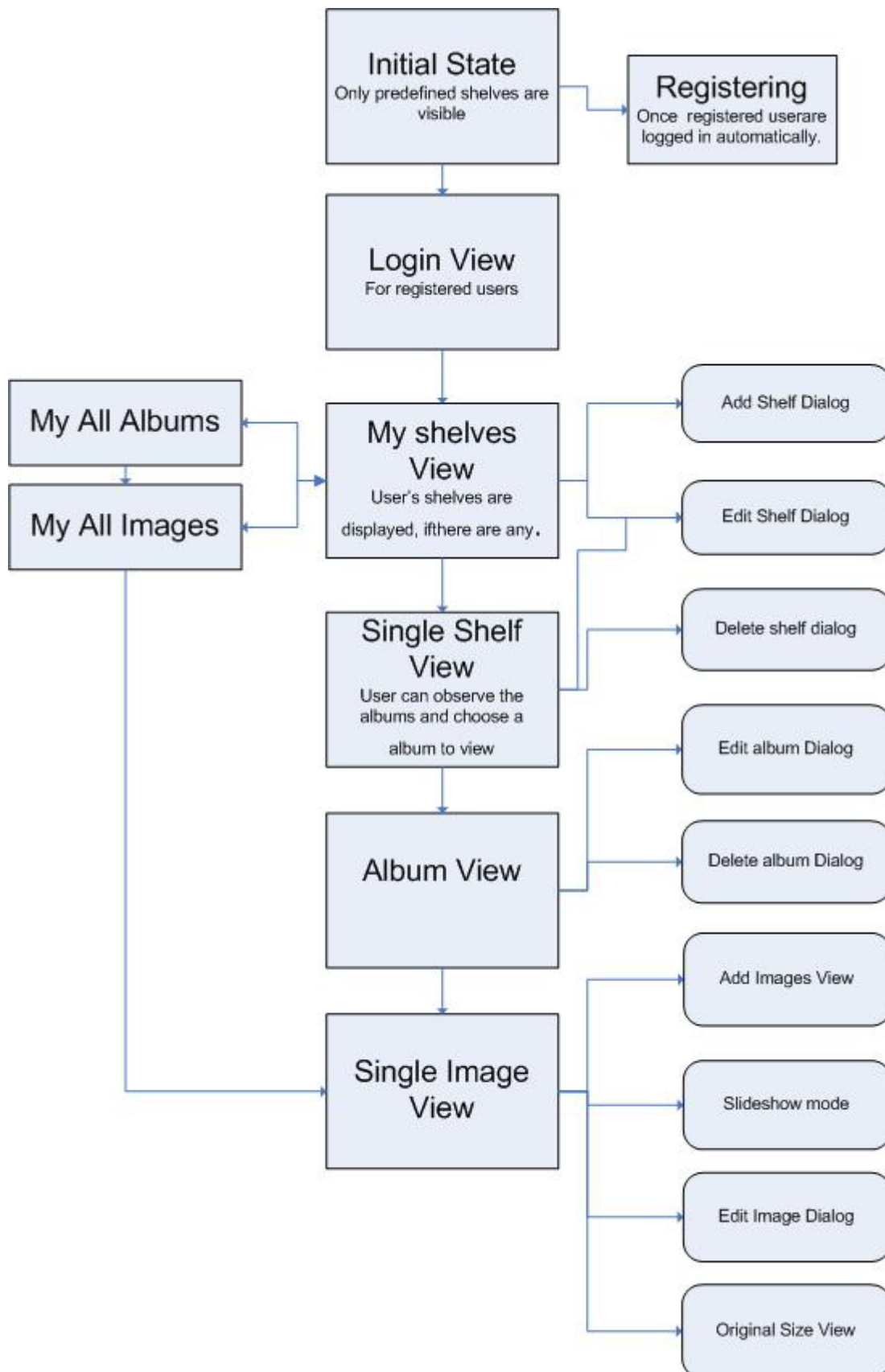


Figure 3.1. Page Flow diagram

3.1.1. Page flows: implementation details

This section covers how the particular elements that govern page flow are implemented in the application.

3.1.1.1. Registering

Registering in is basically the first step a user takes in the application if he/she wants to have access to all features of the application. Have a look at a piece of code from `\includes\index\index.xhtml`:

```
...
<h:panelGroup rendered="#{!identity.loggedIn}" styleClass="top-right-bottom-menu-item-link" layout="block">
  <h:form style="margin: 0px">
    <a4j:commandLink value="#{messages['login.register']}" actionListener="#{authenticator.goToRegister}" regRender="mainArea"
    >
      </h:form>
    </h:panelGroup>
  ...
```

When the button is hit the `goToRegister` method of the `Authenticator` class is invoked and the `START_REGISTER_EVENT` is raised. These action display the registration form that is included from `\includes\register.xhtml`.

The `<a4j:commandLink>` displays the link to the registration form and invokes the `goToRegister` method.

When all fields are filled out with correct values the `authenticator.register(user)` is triggered and a new user object is declared.

3.1.1.2. Navigation Between Pages

Technically, user does not browse between pages of the application: every content page is included into the content area of `index.xhtml` file after a certain action performed by user.

```
...
<h:panelGroup styleClass="content_box" layout="block">
  <ui:include src="#{model.mainArea.template}" />
</h:panelGroup>
...
```

Figure 3.2. Content Area

The `NavigationEnum` enumeration encapsulated all possible states, that can be applied to content area ("mainArea") on the page.

```
...
public enum NavigationEnum {
    ANONYM("includes/publicShelves.xhtml"),
    FILE_UPLOAD("includes/fileUpload.xhtml"),
    USER_PREFS("includes/userPrefs.xhtml"),
    REGISTER("includes/register.xhtml"),
    SEARCH("includes/search.xhtml"),
    ALBUM_PREVIEW("includes/album.xhtml"),
    ALBUM_IMAGE_PREVIEW("/includes/image.xhtml"),
    SHELF_PREVIEW("/includes/shelf.xhtml"),
    ALL_SHELFS("/includes/userShelves.xhtml"),
    TAGS("includes/tag.xhtml"),
    ALL_ALBUMS("/includes/userAlbums.xhtml"),
    ALL_IMAGES("/includes/userImages.xhtml"),
    ALBUM_IMAGE_EDIT("/includes/imageEdit.xhtml"),
    ALBUM_EDIT("/includes/albumEdit.xhtml"),
    SHELF_EDIT("/includes/shelfEdit.xhtml"),
    SHELF_UNVISITED("/includes/shelfUnvisited.xhtml"),
    USER_SHARED_ALBUMS("/includes/userSharedAlbums.xhtml"),
    USER_SHARED_IMAGES("/includes/userSharedImages.xhtml"),
    ALBUM_UNVISITED("/includes/albumUnvisited.xhtml");
    ...
}
```

This class specifies which file is included depending on some user action. The template to be loaded is identified according to some condition in the Controller (`Controller.java`) class and is saved to the Model (`Model.java`). During `index.xhtml` page rendering the value is taken from the Model to define what should be rendered to the page.

3.1.2. Data Model

The data model of the application has the following structure:



Figure 3.3. Photo Album Data Model

How it works

In this chapter we explain how the Photo Album works.

4.1. Used Components

Have a look at the list of components used in the Photo Album application.

Table 4.1. Components used in "Photo Album Demo"

Name	Value
<a4j:commandLink>	The component is very similar to the <h:commandLink> component, the only difference is that an Ajax form submit is generated on a click and it allows dynamic rerendering after a response comes back. It's not necessary to plug any support into the component, as Ajax support is already built in.
<a4j:commandButton>	The component is very similar to the <h:commandButton> component, the only difference is that an Ajax form submit is generated on a click and it allows dynamic rerendering after a response comes back. It's not necessary to plug any support into the component, as Ajax support is already built in.
<a4j:poll>	The component allows periodical sending of Ajax requests to the server and is used for a page update according to a specified time interval.
<rich:calendar>	The component is used for creating monthly calendar elements on a page.
<rich:contextMenu>	The component is used for creation of multileveled context menus that are activated after a user defines an event ("onmouseover", "onclick", etc.) on any element on the page.
<rich:dataGrid>	The component to render data as a grid that allows choosing data from a model and obtains built-in support of Ajax updates.
<rich:datascroller>	The component is designed for providing the functionality of tables scrolling using Ajax requests.

Name	Value
<rich:fileUpload>	The component is designed to perform Ajax-ed files upload to the server.
<rich:inplaceInput>	The component is an input component used for displaying and editing data inputted.
<rich:inplaceSelect>	The component is used to create select based inputs: it shows the value as text in one state and enables editing the value, providing a list of options in another state.
<rich:mediaOutput>	The component implements one of the basic features specified in the framework. The component is a facility for generating images, video, sounds and other binary resources defined by you on-the-fly.
<rich:modalPanel>	The component implements a modal dialog window. All operations in the main application window are locked out while this window is active. Opening and closing the window is done with client JavaScript code.
<rich:progressBar>	The component is designed for displaying a progress bar which shows the current status of the process.
<rich:tree>	The component is designed for hierarchical data presentation and is applied for building a tree structure with a drag-and-drop capability. The component also uses built-in drag and drop.

4.2. Skinnability

The Photo Album application employs such feature of RichFaces framework as skinnability. If you have a look at the web.xml you will see that the `org.richfaces.SKIN` parameter has "photoalbum" value.

```
...
<context-param>
  <param-name>org.richfaces.SKIN</param-name>
  <param-value>photoalbum</param-value>
</context-param>
...
```

This means that the application uses the custom "photoalbum" skin. The skin parameters are stored in the `photoalbum.skin.properties` file that is located in the `photoalbum\source\web\src\main\resources\META-INF\skins\` folder.

Each visual RichFaces component has a XCSS file where some CSS selectors are defined with style properties mapped to the skin parameters. Here is a fragment of the XCSS file of **<rich:calendar>**.

```
...
<u:selector name=".rich-calendar-header">
  <u:style name="border-bottom-color" skin="panelBorderColor"/>
  <u:style name="background-color" skin="additionalBackgroundColor"/>
  <u:style name="font-size" skin="generalSizeFont"/>
  <u:style name="font-family" skin="generalFamilyFont"/>
</u:selector>
...
```

This code sets style for upper part of the calendar. Hence, for example, `font-family` property is mapped to the `generalFamilyFont` property which in its turn has `Arial, Verdana, sans-serif` value.

These are all values the `.rich-calendar-header` has.

```
...
panelBorderColor=#636363
additionalBackgroundColor=#F2F2F2
generalSizeFont=11px
generalFamilyFont=Arial, Verdana, sans-serif
...
```

You can find more information about the Skinnability feature in [RichFaces Developer Guide](#) [../devguide/html_single/index.html/#Skinnability].

4.3. Navigation tree

The **<rich:tree>** component takes one of the main places in the Photo Album and is tightly bound with the application logic. It helps to represent and implement inherently the "Shelves - Albums" hierarchy. Shelf is the highest possible level in the tree hierarchy, that is used to group thematic albums and may contain as many albums as needed.

There are two types of navigation tree in the application: for a registered user and for a guest. The difference between them is that the first one has a context menu and drag-and-drop possibility.

4.3.1. Navigation tree for a guest

Navigation tree for a guest is represented as a simple `<rich:tree>` component.

There are several ways to implement the `<rich:tree>` on a page. In the current application the `<rich:tree>` is designed using a model tag `<rich:treeNodesAdaptor>`.

The `<rich:treeNodesAdaptor>` component has a *"nodes"* attribute that accepts a collection of elements, so `<rich:treeNodesAdaptor>` iterates over the collection and renders a hierarchical tree structure on a page.

According to the "Shelves - Albums" hierarchy we need two nested `<rich:treeNodesAdaptor>` components. The first one iterates over the Shelves collection that is returned from the `getPredefinedShelves()` method of `ShelfManager.java` class:

```
...
public List<Shelf> getPredefinedShelves() {
    if (shelves == null) {
        shelves = shelfAction.getPredefinedShelves();
    }
    return shelves;
}
}
...
```

The second `<rich:treeNodesAdaptor>` component iterates over the Albums collection of the current Shelf which is available via *"var"* attribute. The *"var"* attribute is used to get access to the data object of the current collection element Shelf, so it is possible to output any necessary data. Let's see the `src/main/webapp/includes/index/tree.xhtml` file:

```
...
<rich:tree adviseNodeOpened="#{treeManager.adviseNodeSelected}"
    adviseNodeSelected="#{treeManager.adviseNodeSelected}"
    ajaxSubmitSelection="false" id="PreDefinedTree"
    treeNodeVar="treeNode" switchType="client"
    iconCollapsed="/img/shell/tree_icon_plus.png"
    iconExpanded="/img/shell/tree_icon_minus.png"
    showConnectingLines="false">
    <rich:treeNodesAdaptor nodes="#{shelfManager.getPredefinedShelves()}" var="shelf">

        <rich:treeNode style="cursor:pointer" reRender="treePanel,mainArea" selectedClass="tree-
selected-node">
            <f:facet name="icon">
                <h:graphicImage style="border: none" value="/img/shell/tree_icon_shelf.png">
```

```

<a4j:support reRender="treePanel,
mainArea" event="onclick" actionListener="#{controller.showShelf(shelf)}" similarityGroupingId="sel"
>
    </h:graphicImage>
</f:facet>
<a4j:outputPanel >
    <h:outputText style="cursor:pointer" value="#{shelf.name}" />
    <h:outputText value=" :: " />
    <strong>#{shelf.unvisitedImages.size()}</strong> new
    <a4j:support reRender="treePanel,
mainArea" event="onclick" actionListener="#{controller.showShelf(shelf)}" similarityGroupingId="sel"
>
        </a4j:outputPanel>
</rich:treeNode>

<rich:treeNodesAdaptor var="album" nodes="#{shelf.albums}">

    <rich:treeNode style="cursor:pointer" reRender="treePanel,mainArea" selectedClass="tree-
selected-node" icon="img/shell/tree_icon_album.png">
        <f:facet name="iconLeaf">
            <h:graphicImage style="border: none" value="img/shell/tree_icon_album.png">
                <a4j:support reRender="treePanel,
mainArea" event="onclick" actionListener="#{controller.showAlbum(album)}" similarityGroupingId="sel"
>
                    </h:graphicImage>
                </f:facet>
                <a4j:outputPanel>
                    <h:outputText style="cursor:pointer" value="#{album.name}" />
                    <h:outputText value=" :: " />
                    <strong>#{album.unvisitedImages.size()}</strong> new
                    <a4j:support reRender="treePanel,
mainArea" event="onclick" actionListener="#{controller.showAlbum(album)}" similarityGroupingId="sel"
>
                        </a4j:outputPanel>
                    </rich:treeNode>
                </rich:treeNodesAdaptor>
            </rich:treeNodesAdaptor>
        </rich:tree>
    ...

```

The image below shows how the navigation tree for a guest is rendered on the page.

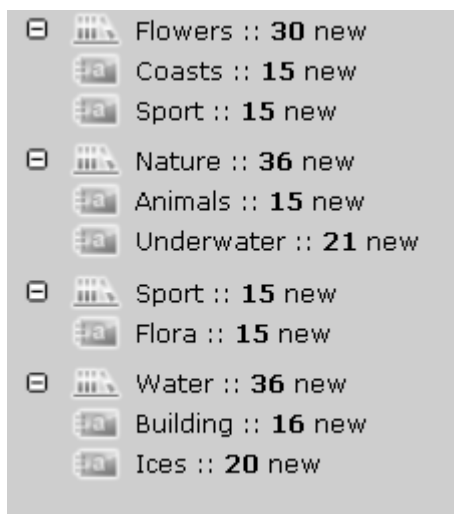


Figure 4.1. Shelves and albums nodes rendered with the help of the `<rich:treeNodesAdaptor>`

4.3.2. Navigation tree for a registered user

As it was mentioned before a navigation tree for a registered user has two main features: drag-and-drop and context menu. Context menu is described in the ["Context menu"](#) chapter.

Drag-and-drop feature supported in the Photo Album application is not so complicated as it may seem from the first view. In this application we can mark out two types of drag-and-drop: one type takes place only inside the tree (between tree nodes) and another one - between the watching area and the tree. The difference is not considerable enough to describe two types separately, but also not at all insignificant to be omitted here.

The tree related components (`<rich:tree>` and `<rich:treeNode>`) have their own attributes that provide drag-and-drop functionality. These attributes can be divided into two groups: those which provide drag (`dragValue`, `dragListener`, `dragIndicator`, `dragType` attributes) and those which provide drop operations (`dropValue`, `dropListener`, `acceptedTypes`, `typeMapping`).

Note:

Due to "Shelves - Albums - Photos" hierarchy we can say that photos could be moved between albums, albums could be moved between shelves. To avoid a mishmash, it's not allowed to place photos directly in shelves as well as nesting shelves inside shelves or albums inside albums.

Let's explore how drag-and-drop works for albums.

All albums, that are represented as `TreeNode`s, must be marked somehow for dragging. For this purpose we use previously mentioned `"dragValue"`, `"dragType"` attributes. Let's have a look at the `src/main/webapp/includes/index/tree.xhtml` file:

```

<rich:treeNodesAdaptor var="album" nodes="#{shelf.albums}">
  <rich:treeNode style="cursor:pointer"
    reRender="mainArea, treePanel"
    dragType="album"
    dragValue="#{album}"
    dropValue="#{album}"
    acceptedTypes="image"
    selectedClass="tree-selected-node"
    icon="img/shell/tree_icon_album.png">
    ...
    <rich:dndParam name="label" type="drag" value="#{album.name}" />
    ...
  </rich:treeNode>
</rich:treeNodesAdaptor>

```

To provide drop functionality for the marked albums, we should mark Shelves as drop zones in the application code too. For this purpose we add the *"dropValue"* and *"acceptedTypes"* attributes to the "Shelf" node in the same `src/main/webapp/includes/index/tree.xhtml` file:

```

<rich:treeNodesAdaptor nodes="#{shelfManager.getUserShelves()}" var="shelf">
<rich:treeNode style="cursor:pointer" acceptedTypes="album" dropValue="#{shelf.name}" reRender="mainArea,
  treePanel" selectedClass="tree-selected-node">
  ...
  </rich:treeNode>
</rich:treeNodesAdaptor>

```

The *"acceptedTypes"* attribute tells the "Shelf" node what types of dragged zones (albums in this case) it can accept. We have specified "Album" node *"dragType"* as "album", so the "Shelf" node can accept it.

Finally in order to process drop on the server side we need to specify a listener for the **<rich:tree>** in the *"dropListener"* attribute (`src/main/webapp/includes/index/tree.xhtml` file):

```

<rich:tree adviseNodeOpened="#{treeManager.adviseNodeSelected}"
  adviseNodeSelected="#{treeManager.adviseNodeSelected}"
  ajaxSubmitSelection="false" dragIndicator="dragIndicator"
  treeNodeVar="treeNode" switchType="client"
  iconCollapsed="/img/shell/tree_icon_plus.png"
  iconExpanded="/img/shell/tree_icon_minus.png"
  dropListener="#{dndManager.processDrop}"

```

```
        showConnectingLines="false">
        ...
    </tree>
```

The code for the `<rich:dragIndicator>` looks like the following:

```
<rich:dragIndicator id="dragIndicator" />
```

The `processDrop()` method of `DnDManager.java` class is shown in the listing below:

```
...
public void processDrop(DropEvent dropEvent) {
    Dropzone dropzone = (Dropzone) dropEvent.getComponent();
    Object dragValue = dropEvent.getDragValue();
    Object dropValue = dropzone.getDropValue();
    if(dragValue instanceof Image){
        if(!((Album)dropValue).getOwner().getLogin().equals(user.getLogin())){
            Events.instance().raiseEvent(Constants.ADD_ERROR_EVENT, Constants.DND_PHOTO_ERROR);
            return;
        }
        handleImage((Image)dragValue, (Album)dropValue);
    }else if(dragValue instanceof Album){
        if(!((Shelf)dropValue).getOwner().getLogin().equals(user.getLogin())){
            Events.instance().raiseEvent(Constants.ADD_ERROR_EVENT, Constants.DND_ALBUM_ERROR);
            return;
        }
        handleAlbum((Album)dragValue, (Shelf)dropValue);
    }
}
...
```

Here is the whole example of the "Navigation tree for a registered user":

```
<h:panelGroup id="tree" rendered="#{identity.hasRole('admin')}}" layout="block">
    <a4j:commandLink actionListener="#{controller.selectShelves()}" reRender="mainArea,
treePanel"><h2><h:outputText value="My shelves:"/></h2></a4j:commandLink><br/>
    <rich:dragIndicator
        id="dragIndicator" />
    <rich:tree
        adviseNodeOpened="#{treeManager.adviseNodeSelected}"
```

```

        adviseNodeSelected="#{treeManager.adviseNodeSelected}"
        ajaxSubmitSelection="false" dragIndicator="dragIndicator"
        treeNodeVar="treeNode" switchType="client"
        iconCollapsed="/img/shell/tree_icon_plus.png"
        iconExpanded="/img/shell/tree_icon_minus.png"
        dropListener="#{dndManager.processDrop}"
        showConnectingLines="false">

        <f:facet name="icon">
            <h:graphicImage style="border: none" value="/img/shell/tree_icon_shelf.png">
                <a4j:support reRender="treePanel,
mainArea" event="onclick" actionListener="#{controller.showShelf(shelf)}" similarityGroupingId="sel"
            >

                </h:graphicImage>
            </f:facet>
            <ui:include src="/includes/contextMenu/CMForShelf.xhtml" >
                <ui:param name="shelf" value="#{shelf}" />
            </ui:include>
            <a4j:outputPanel>
                <h:outputText style="cursor:pointer" value="#{shelf.name}" />
                <h:outputText value=" :: " />
                <strong>#{shelf.unvisitedImages.size()}</strong> new
                <a4j:support reRender="treePanel,
mainArea" event="onclick" actionListener="#{controller.showShelf(shelf)}" similarityGroupingId="sel"
            >

            </a4j:outputPanel>
        </rich:treeNode>

        <rich:treeNodesAdaptor var="album"
            nodes="#{shelf.albums}">
            <rich:treeNode style="cursor:pointer" reRender="mainArea,
treePanel" dragType="album"
                dragValue="#{album}" dropValue="#{album}"
                acceptedTypes="image"
                selectedClass="tree-selected-node"
                icon="img/shell/tree_icon_album.png">
                <f:facet name="iconLeaf">
                    <h:graphicImage style="border: none" value="img/shell/tree_icon_album.png">
                        <a4j:support reRender="treePanel,
mainArea" event="onclick" actionListener="#{controller.showAlbum(album)}" similarityGroupingId="sel"
                    >

                        </h:graphicImage>
                    </f:facet>
                    <ui:include src="/includes/contextMenu/CMForAlbum.xhtml" >

```

```

<ui:param name="album" value="#{album}" />
</ui:include>
<rich:dndParam name="label" type="drag" value="#{album.name}" />
<a4j:outputPanel >
    <h:outputText style="cursor:pointer" value="#{album.name}" />
    <h:outputText value=" :: " />
    <strong>#{album.unvisitedImages.size()}</strong> new
    <a4j:support reRender="treePanel,
mainArea" event="onclick" actionListener="#{controller.showAlbum(album)}" similarityGroupId="sel"
    >

    </a4j:outputPanel>
</rich:treeNode>

</rich:treeNodesAdaptor>
</rich:treeNodesAdaptor>

</rich:tree>
</h:panelGroup>

```

The image below shows how the described above drag-and-drop features are rendered in the Photo Album.

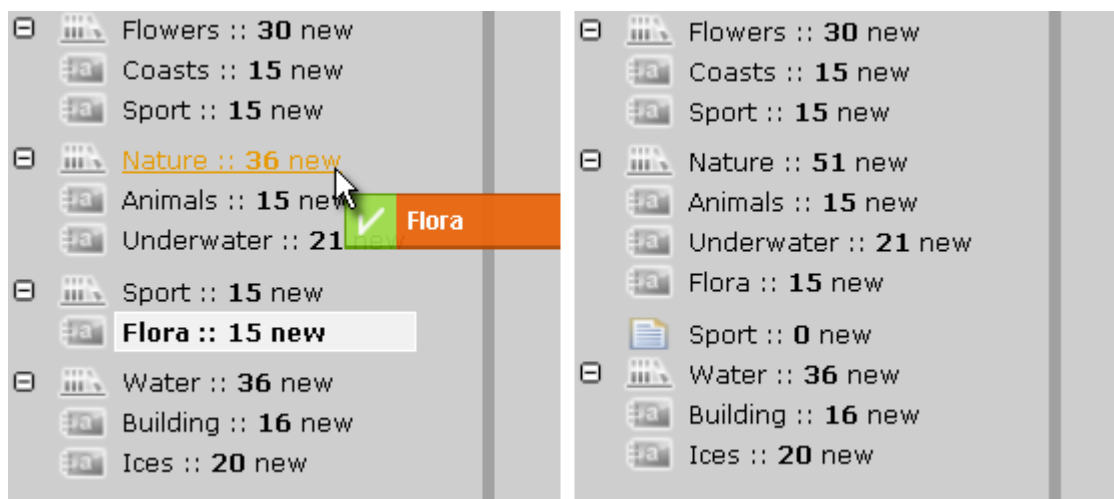


Figure 4.2. Dragging the "Flora" album from "Sport" shelf into the "Nature" (left) and the tree after drag-and-drop (right).

Vizit following pages at RichFaces Live Demo for more information, examples and sources on the components used in the application and described in this chapter:

- [Tree](http://livedemo.exadel.com/richfaces-demo/richfaces/tree.jsf?c=tree) [http://livedemo.exadel.com/richfaces-demo/richfaces/tree.jsf?c=tree] for the `<rich:tree>` component;

- [TreeNodeAdaptor](#) [http://livedemo.exadel.com/richfaces-demo/richfaces/treeNodesAdaptor.jsf?c=treeNodesAdaptor] for the **<rich:treeNodesAdaptor>** component;
- [DragIndicator](#) [http://livedemo.exadel.com/richfaces-demo/richfaces/dragSupport.jsf?c=dragIndicator] for the **<rich:dragIndicator>** component;
- [DragDropParameter](#) [http://livedemo.exadel.com/richfaces-demo/richfaces/dragSupport.jsf?c=dndParam] for the **<rich:dndParam>** component.

4.4. Album View

Album view allows you to observe album items as thumbnails, scale the size of the thumbnails with the slider control, as well as to switch to slideshow mode. By clicking on an image in the Album View the Image View is opened.

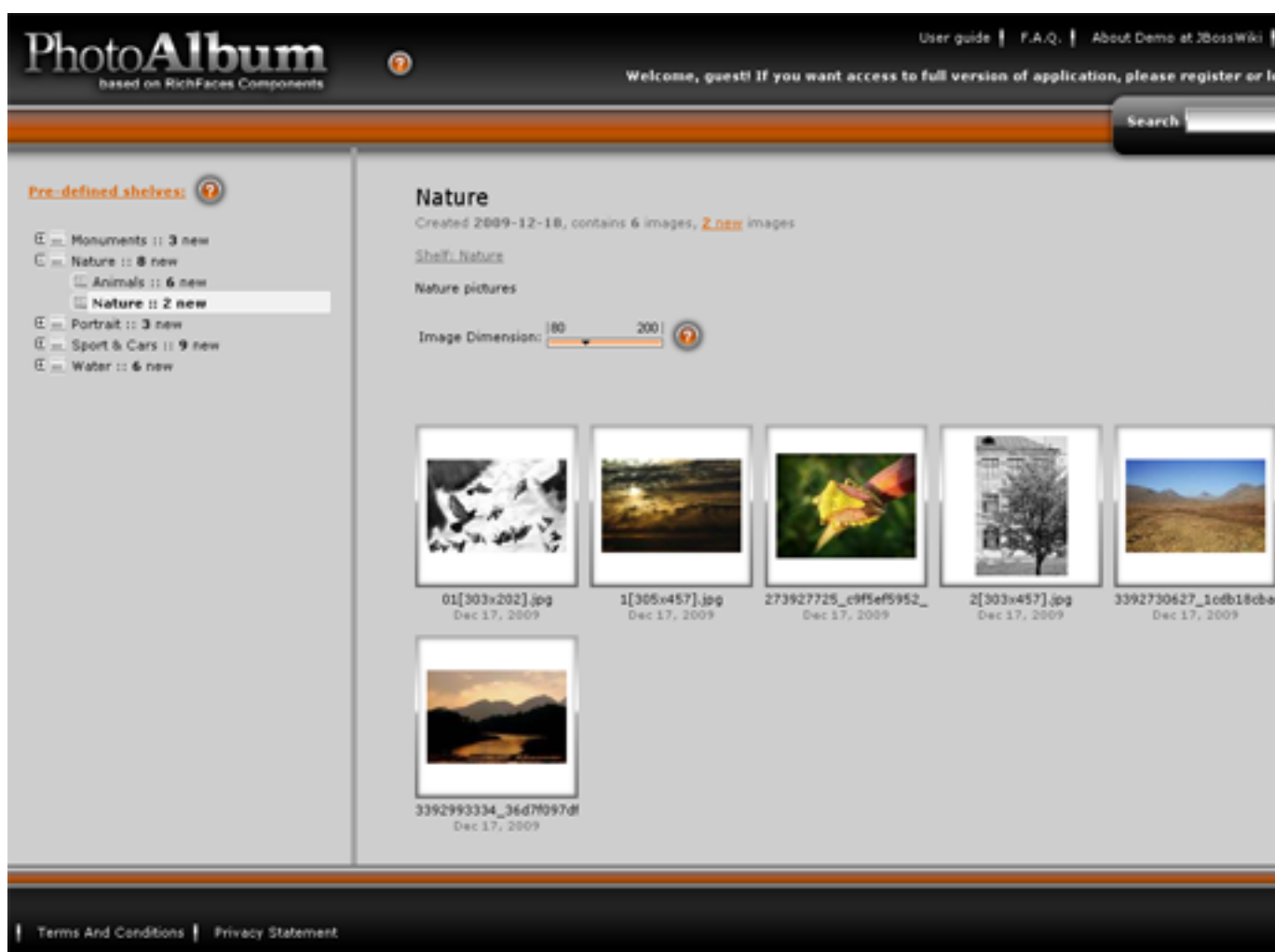


Figure 4.3. Some title

4.4.1. Image Size Control with <rich:inputNumberSlider>

The **rich:inputNumberSlider** component in the Photo Album Demo is used as a control that helps a user to change photos size while previewing an album. A handler position on the slider track corresponds to a particular value of an image size. The component is included into the page with the help of **ui:include**:

```
...
<ui:include src="src/main/webapp/includes/image/inputNumberSlider.xhtml"/>
...
```

Now let's have a look at `src/main/webapp/includes/image/inputNumberSlider.xhtml` file:

```
...
<ui:composition ...>
  <div>
    <rich:inputNumberSlider value="#{imageSizeHelper.value}"
      minValue="80"
      maxValue="200"
      step="40"
      enableManualInput="false"
      showArrows="false"
      showBoundaryValues="true"
      showInput="false">
      <a4j:support event="onchange" reRender="userAlbumImages"/>
    </rich:inputNumberSlider>
  </div>
</ui:composition>
...
```

There is special Enumeration type that contains four predefined values for image size. Each type has a set of image related attributes such as CSS class for new photo size, postfix for a new file name, image background. The `setValue` method of the `ImageSizeHelper.java` class is triggered on each slider's position change. This method sets one of four predefined values for image size and due to slider's position.

```
public void setValue(int value) {
    currentDimension = ImageDimension.getInstance(value);
    this.value = currentDimension.getX();
}
```

And here is the `ImageDimension.java` class:

```
...
public enum ImageDimension {

    SIZE_80(80), SIZE_120(120), SIZE_160(160), SIZE_200(200), SIZE_MEDIUM(600), ORIGINAL(0);

    final static String CSS_CLASS = "preview_box_photo_";
    final static String FILE_POSTFIX = "_small";
    final static String IMAGE_BG = "/img/shell/frame_photo_%1$d.png";
    final static String IMAGE_BG_STYLE = "width: %1$dpx; height: %1$dpx";

    int x;
    String bgStyle;
    String cssClass;
    String imageBgSrc;
    String filePostfix;

    private ImageDimension(int x) {
        this.x = x;
        this.bgStyle = String.format(IMAGE_BG_STYLE, x + 20);
        cssClass = CSS_CLASS + x;
        imageBgSrc = String.format(IMAGE_BG, (x == 160) ? 200 : x);
        if(x == 600){
            filePostfix = "_medium";
        }else if(x == 0){
            filePostfix = "";
        }else{
            filePostfix = FILE_POSTFIX + x;
        }
    }
}
...
```

After the `<a4j:support>` is worked out user photos (more exactly, the `h:panelGroup` with `userAlbumImages` id that contains user photos) are rendered correspondingly to a new set value. Here is `web/src/main/webapp/includes/image/imageList.xhtml`:

```
...
<h:panelGroup id="userAlbumImages">
    <a4j:repeat id="imageList" value="#{model.images}" var="image" rows="20">

        <h:panelGroup layout="block" styleClass="#{imageSizeHelper.currentDimension.cssClass}">
```



```

eClass="style photo_img" imageSizeHelper.currentDimension.imageBgStyle="imageSizeHelper.currentDimension.imageBg}"
>
<h:panelGrid cellpadding="0">
  <h:panelGroup>
    <a4j:commandLink
      actionListener="#{controller.showImage(image)}"
      reRender="mainArea, treePanel">
      <a4j:mediaOutput id="img" element="img"
        createContent="#{imageLoader.paintImage}"
        style="border : 1px solid #FFFFFF;"
        value="#{fileManager.transformPath(image.fullPath,
imageSizeHelper.currentDimension.filePostfix)}">
        <f:param value="#{imageSizeHelper.currentDimension.x}" name="x" />

    <rich:dragSupport rendered="#{controller.isUserImage(image)}" reRender="mainArea,
treePanel" id="dragSource" dragIndicator="dragIndicator"
      dragType="image" dragValue="#{image}">
        <rich:dndParam id="dragParam" name="label" value="#{image.name}" />
      </rich:dragSupport>
    </a4j:mediaOutput>
  </a4j:commandLink>
  <br/>
</h:panelGroup>
<ui:include src="/includes/contextMenu/CMForImage.xhtml" >
  <ui:param name="image" value="#{image}" />
  <ui:param name="mediaOutput" value="#{rich:clientId('img')}" />
</ui:include>

</h:panelGrid>
<h:panelGroup layout="block" styleClass="photo_name">#{image.name}</h:panelGroup>

<h:panelGroup layout="block" styleClass="photo_data">
  <h:outputText value="#{image.created}">
    <f:convertDateTime />
  </h:outputText>
</h:panelGroup>
</h:panelGroup>
</a4j:repeat>
</h:panelGroup>
...

```

When the **<rich:inputNumberSlider>** is rendered, at first its default value for image size is 120 px.

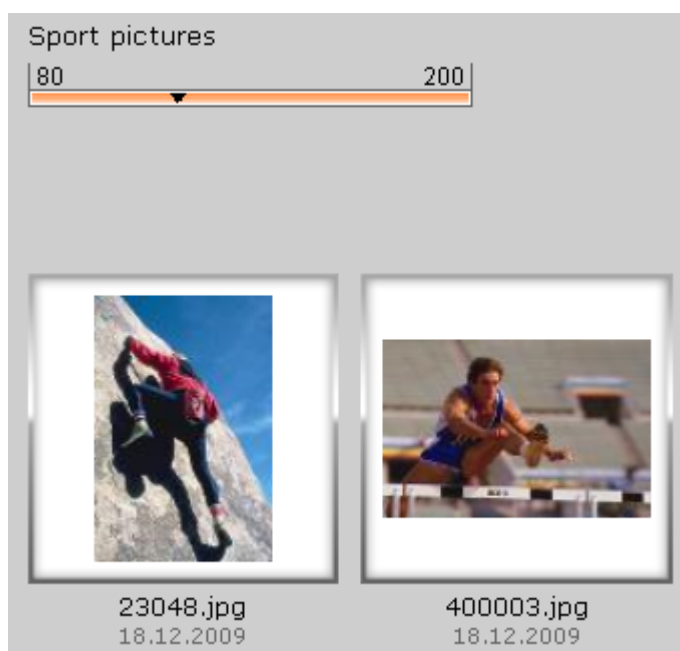


Figure 4.4. Image size control

Visit following pages at RichFaces Live Demo for more information, examples and sources on the components used in the application and described in this chapter:

- [InputNumberSlider](http://livedemo.exadel.com/richfaces-demo/richfaces/inputNumberSlider.jsf?c=inputNumberSlider) [http://livedemo.exadel.com/richfaces-demo/richfaces/inputNumberSlider.jsf?c=inputNumberSlider] page for the `<rich:inputNumberSlider>` component;
- [AjaxSupport](http://livedemo.exadel.com/richfaces-demo/richfaces/support.jsf?c=support) [http://livedemo.exadel.com/richfaces-demo/richfaces/support.jsf?c=support] for the `<a4j:suport>` component.

4.4.2. Slideshow

The slideshow feature in the Photo Album Demo can be enabled by clicking on "**Start Slideshow**" link from two different places in the application: 1) from user's album preview (`/web/src/main/webapp/image/albumInfo.xhtml`) and 2) from a particular photo preview (`src/main/webapp/image/imageInfo.xhtml`). Both of two mentioned XHTML files include slideshow with the help of Facelets `<ui:include` tag (for more information about `<ui:include` see Facelets Reference Guide — <http://www.jsftoolbox.com/documentation/facelets/01-Introduction/index.jsf>).

The components that implement the slideshow functionality are:

- `<rich:modalPanel>`; located in `web/src/main/webapp/includes/image/slideshow.xhtml` that is hidden by default as the attribute `showWhenRendered="#{slideshow.active}"` and the active property of `SlideshowManager.java` is set to "false" by default.

- **<a4j:poll>** located in `includes/misc/slideshowPooler.xhtml` which is also inactive due to the mentioned active property (`active=#{slideshow.active}`)

After activation, **<a4j:poll>** will send asynchronous requests to the server with some certain interval, as the result of these requests modal panel will display the next image in the row.

Now let's have a look at the details of the slideshow implementation.

The `startSlideshow()` method of `SlideshowManager.java` is invoked when no photo is selected in the current image list. The method iterates over all photos of a particular album starting from the first one in the list. Look at the `SlideshowManager.java` listing below:

```
...
...
public void startSlideshow(){
    active = true;
    this.slideshowIndex = 0;
    if(model.getImages() == null || model.getImages().size() < 1){
        stopSlideshow();
        Events.instance().raiseEvent(Constants.ADD_ERROR_EVENT, "No images for
slideshow!");
        return;
    }
    this.selectedImage = model.getImages().get(this.slideshowIndex);
    this.selectedImage.getAlbum().visitImage(selectedImage, true);
}
...
```

The second variation of the `startSlideshow()` method is activated when a link to slide-show is clicked from a particular photo preview. This method iterates over the rest of photos starting from the current selected one:

```
...
public void startSlideshow(Image selectedImage){
    active = true;
    if(model.getImages() == null || model.getImages().size() < 1){
        stopSlideshow();
        Events.instance().raiseEvent(Constants.ADD_ERROR_EVENT, "No images for
slideshow!");
        return;
    }
    this.slideshowIndex = model.getImages().indexOf(selectedImage);
    this.selectedImage = selectedImage;
    this.selectedImage.getAlbum().visitImage(selectedImage, true);
}
```

```
}  
...
```

Both variants of `startSlideshow()` method set the `active` property to "true" as a result the poller is activated and modal panel becomes visible.

The `slideshow` modal panel is kept in the `web/src/main/webapp/includes/image/slideshow.xhtml` file and referred from the corresponding pages with the help of `<ui:include>` Facelets tag:

```
...  
<ui:include src="/includes/image/slideshow.xhtml"/>  
...
```

Have a look at `web/src/main/webapp/includes/image/slideshow.xhtml` file:

```
...  
<ui:composition xmlns="http://www.w3.org/1999/xhtml" ...>  
  
    <rich:modalPanel showWhenRendered="#{slideshow.active}"  
  
        domElementAttachment="parent"  
  
        id="slideShowModalPanel"  
  
        width="650"  
  
        onshow="showPictureEffect();"   
  
        height="650">  
  
        <f:facet name="controls">  
  
            <h:panelGroup  
  
                <h:graphicImage value="/img/modal/close.png" style="cursor:pointer" id="hidelink">  
  
<a4j:support event="onclick" actionListener="#{slideshow.stopSlideshow}" reRender="slideShowForm,  
    mainArea, tree" />  
  
            </h:graphicImage>
```

```
        </h:panelGroup>

        </f:facet>

        ...

    </rich:modalPanel>

</ui:composition>

...
```

This is the source code of `includes/misc/slideShowPooler.xhtml`:

```
...
<!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:s="http://jboss.com/products/seam/taglib"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:rich="http://richfaces.org/rich"
    xmlns:a4j="http://richfaces.org/a4j"
    xmlns:richx="http://richfaces.org/richx">
    <a4j:form id="slideShowForm">
        <a4j:poll reRender="slideshowImage"
            actionListener="#{slideshow.showNextImage()}"
            interval="#{slideshow.interval}"
            enabled="#{slideshow.active}"
            onsubmit="hidePictureEffect()"
            oncomplete="showPictureEffect();"/>
    </a4j:form>
</ui:composition>
...
```

The slideshow poller sends the request for the next image (`showNextImage()` method) each four seconds. The interval is defined in the `interval` property of the `SlideshowManager.java` and refers to a `INITIAL_DELAY` constant (`constants.java`).

The described above example implements a modal panel with photos that rotate them in the order they are stored in an album.

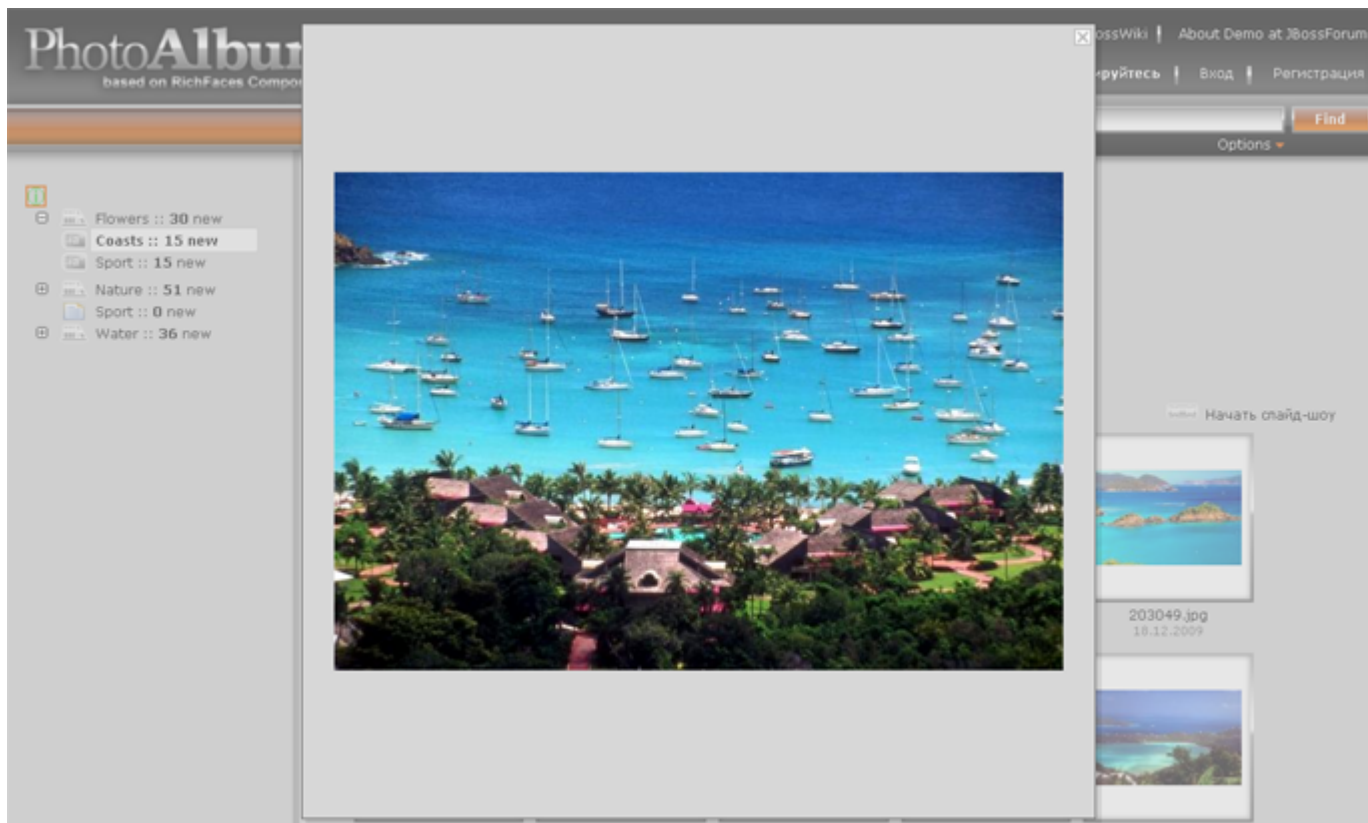


Figure 4.5. Image size control

To stop the slide-show user clicks **Close window** button on the slide-show panel and `stopSlideshow()` method is invoked.

```
...
@Observer("stopSlideshow")
public void stopSlideshow(){
    active = false;
    this.selectedImage = null;
    this.slideshowIndex = 0;
}
...
```

The `active` field is set to "false" again, consequently the poller becomes inactive and the modal panel becomes invisible too.

Vizit following pages at RichFaces Live Demo for more information, examples and sources on the components used in the application and described in this chapter:

- [ModalPanel](http://livedemo.exadel.com/richfaces-demo/richfaces/modalPanel.jsf?c=modalPanel) [http://livedemo.exadel.com/richfaces-demo/richfaces/modalPanel.jsf?c=modalPanel] page for the `<rich:modalPanel>` component;

- [Effect](http://livedemo.exadel.com/richfaces-demo/richfaces/effect.jsf?c=effect) [http://livedemo.exadel.com/richfaces-demo/richfaces/effect.jsf?c=effect] for the **<rich:effect>** component;
- [MediaOutput](http://livedemo.exadel.com/richfaces-demo/richfaces/mediaOutput.jsf?c=mediaOutput) [http://livedemo.exadel.com/richfaces-demo/richfaces/mediaOutput.jsf?c=mediaOutput] for the **<a4j:mediaOutput>** component;
- [AjaxSupport](http://livedemo.exadel.com/richfaces-demo/richfaces/support.jsf?c=support) [http://livedemo.exadel.com/richfaces-demo/richfaces/support.jsf?c=support] for the **<a4j:suport>** component;
- [CommandLink](http://livedemo.exadel.com/richfaces-demo/richfaces/commandLink.jsf?c=commandLink) [http://livedemo.exadel.com/richfaces-demo/richfaces/commandLink.jsf?c=commandLink] for the **<a4j:commandLink>** component;
- [AjaxForm](http://livedemo.exadel.com/richfaces-demo/richfaces/form.jsf?c=form) [http://livedemo.exadel.com/richfaces-demo/richfaces/form.jsf?c=form] for the **<a4j:form>** component;
- [Poll](http://livedemo.exadel.com/richfaces-demo/richfaces/poll.jsf?c=poll) [http://livedemo.exadel.com/richfaces-demo/richfaces/poll.jsf?c=poll] for the **<a4j:poll>** component.

4.5. Image View

Image View in the Photo Album application is a page where only one image is displayed. In this view you can also browse the current album with the image scroller as well as to switch to slideshow mode. If you are a registered user you can leave comments under the current image.

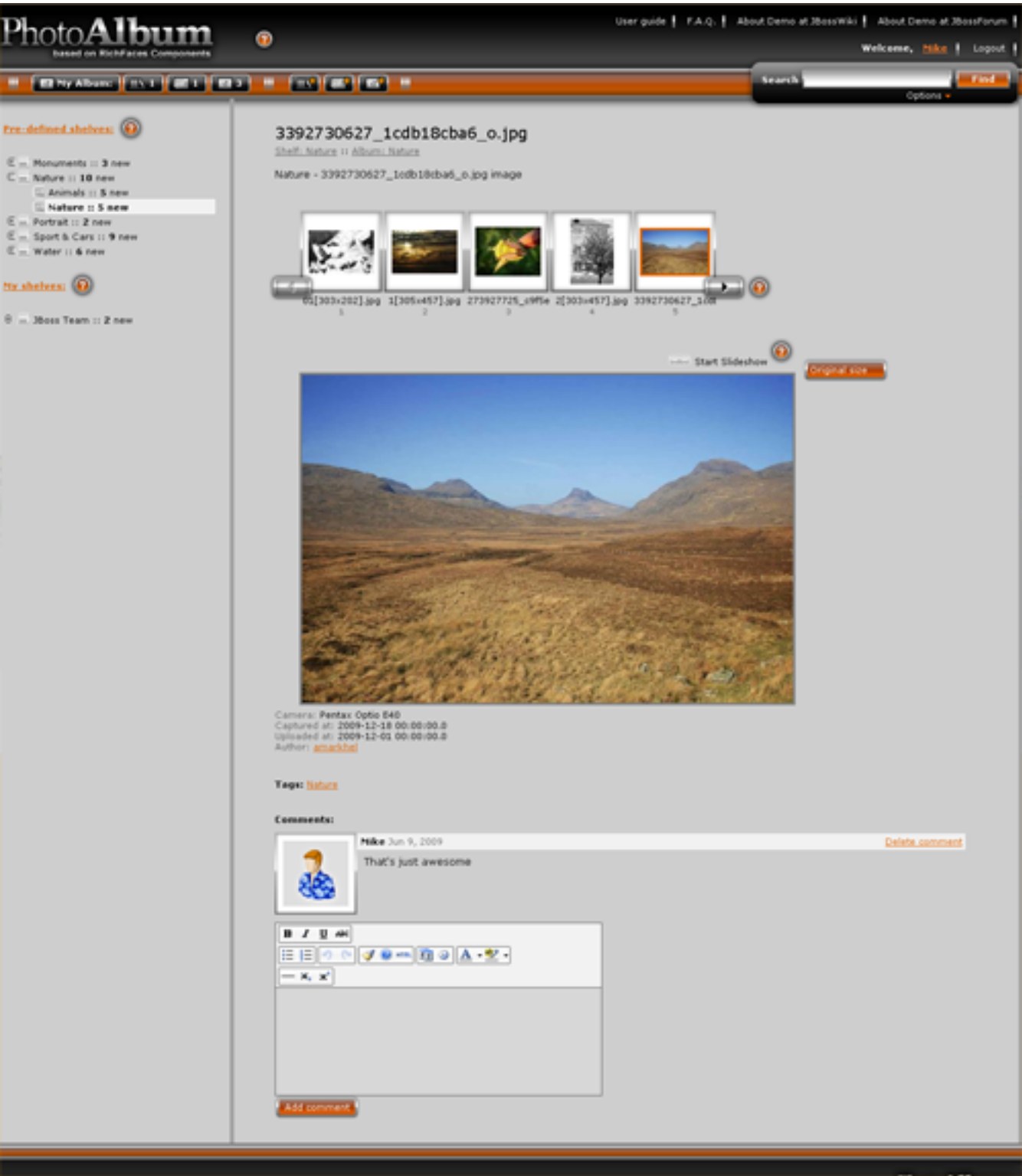


Figure 4.6. Some title

4.5.1. Custom images scroller widget

The Images Scroller implementation in the Photo Album application is basically `<a4j:repeat>` with the value attribute bound to `#{model.selectedAlbum.images}`, which is a collection of images of the selected album and the `<rich:dataScroller>` component tied to the `<a4j:repeat>`.

The source code you can find in the `includes/images/imageScroller.xhtml` file. Now let's go deeper into the details. The main component here is `<a4j:repeat>`:

```
...
<a4j:repeat value="#{model.selectedAlbum.images}" rows="5"
    var="img" id="repeat" rowKeyVar="rk">

    <a4j:outputPanel layout="block"
        styleClass="preview_box_photo_nav #{model.selectedImage == img ?
'preview_box_photo_current' : 'preview_box_photo_default'}">
        <h:panelGroup layout="block" styleClass="preview_box_photo_80">
            <h:graphicImage styleClass="pr_photo_bg"
                value="/img/shell/frame_photo_80.png" />
            <h:panelGrid cellpadding="0" cellspacing="2">
                <h:panelGroup layout="block">
                    <a4j:mediaOutput element="img"
                        createContent="#{imageLoader.paintImage}"
                        value="#{fileManager.transformPath(img.fullPath, '_small80')}" />
                    </a4j:mediaOutput>
                    <br />
                </h:panelGroup>
            </h:panelGrid>
            <h:panelGroup layout="block" styleClass="photo_name">
                <h:outputText value="#{img.name}" />
            </h:panelGroup>
            <h:panelGroup layout="block" styleClass="photo_data">
                <h:outputText value="#{rk + 1}" />
            </h:panelGroup>
        </h:panelGroup>

        <a4j:support event="onclick" rendered="#{model.selectedImage != img}"
            reRender="mainArea,treePanel, imagesTable" action="#{controller.showImage(img)}" />
    </a4j:outputPanel>
</a4j:repeat>
...
```

Each element of the `<a4j:repeat>` has a corresponding `<a4j:outputPanel>` with the `<a4j:mediaOutput>` as a nested element. `<a4j:mediaOutput>` renders the thumbnail of the

image. As the rows attribute is set to "5" (`rows="5"`), only 5 images are displayed on the page at a time.

As you've noticed, the currently selected image in the images scroller has different style, namely: a red frame around thumbnail, which is implemented with this code:

```
...
<a4j:outputPanel layout="block"
styleClass="preview_box_photo_nav      #{model.selectedImage      ==      img      ?
'preview_box_photo_current' : 'preview_box_photo_default'}">
...

```

As you can see from the code snippet, identification of whether the currently selected image is the same image displayed by the `<a4j:repeat>` is performed in the styleClass, if it returns "true", different style is applied.

Each `<a4j:repeat>` has a corresponding `<a4j:support>` configured like this:

```
...
<a4j:support event="onclick"
    rendered="#{model.selectedImage != img}"
    reRender="mainArea,treePanel, imagesTable"
    action="#{controller.showImage(img)}" />
...

```

On every click `<a4j:support>` calls `#{controller.showImage(img)}` method that sets the current image, thumbnail of which has just been clicked on. For more details please see Controller.java class.

To implement thumbnails scrolling effect the `<rich:datascroller>` is attached to the `<a4j:repeat>`:

```
...
<rich:datascroller page="#{controller.getPage()}"
    styleClass="image-scroller" lastPageMode="full" for="repeat" reRender="imagesTable"
    boundaryControls="hide" stepControls="hide">
    <f:facet name="pages">
        <h:outputText />
    </f:facet>
    <f:facet name="fastforward">
        <h:graphicImage styleClass="image-scroller-right-arrow"
            value="img/shell/arr_right.png" />
    </f:facet>
</rich:datascroller>

```

```

</f:facet>
<f:facet name="fastforward_disabled">
  <h:graphicImage styleClass="image-scroller-right-arrow"
    value="img/shell/arr_right_dis.png" />
</f:facet>
<f:facet name="fastrewind">
  <h:graphicImage styleClass="image-scroller-left-arrow"
    value="img/shell/arr_left.png" />
</f:facet>
<f:facet name="fastrewind_disabled">
  <h:graphicImage styleClass="image-scroller-left-arrow"
    value="img/shell/arr_left_dis.png" />
</f:facet>
</rich:dataScroller>

...

```

The page attribute identifies which page should be displayed right now. For instance, if you have only 20 images and the current image has the 12th index in the collection, then the 3rd page will be displayed:

```

...
public Integer getPage(){
    final Integer index = model.getSelectedAlbum().getIndex(model.getSelectedImage());
    return index / 5 + 1;
}

...

```

The `lastPageMode="full"` attribute ensures that 5 thumbnails are always shown on the page. If this attribute hadn't been configured like this, in case the 19th thumbnail out of 20 had been selected then only 2 last thumbnails would have been displayed.

As you can see, `<rich:dataScroller>` has a slightly different look-and-feel, the trick is in the redefinition of `fastforward`, `fastforward_disabled`, `fastrewind` and `fastrewind_disabled` facets on which places we display our images. We didn't redefine other facets because they are not rendered to the page which is achieved with `boundaryControls="hide"` and `stepControls="hide"` attributes of `<rich:dataScroller>`.

Vizit following pages at RichFaces Live Demo for more information, examples and sources on the components used in the application and described in this chapter:

- [DataTableScrolling](http://livedemo.exadel.com/richfaces-demo/richfaces/dataTableScroller.jsf?c=dataTableScroller) [http://livedemo.exadel.com/richfaces-demo/richfaces/dataTableScroller.jsf?c=dataTableScroller] page for the `<rich:dataScroller>` component;

- [Repeat](http://livedemo.exadel.com/richfaces-demo/richfaces/repeat.jsf?c=repeat) [http://livedemo.exadel.com/richfaces-demo/richfaces/repeat.jsf?c=repeat] for the `<a4j:repeat>` component.

4.6. Upload Images

In the previous chapter we have discussed how to create Navigation Trees that represent "Shelves - Albums" hierarchy. Now it is time to upload images.

The `<rich:fileUpload>` component in the Photo Album application uses the embedded Flash module that adds extra functionality to the component:

- Multiple files choosing;
- Definition of permitted file types in the "Open File" dialog window;
- A number of additional client-side object properties.

The code for the `<rich:fileUpload>` is contained in the `/includes/fileUpload/fileUploader.xhtml` page:

```
...
<rich:fileUpload
    allowFlash="true" maxFilesQuantity="100" autoclear="true"
    fileUploadListener="#{fileUploadManager.listener}" id="fileUpload"
    disabled="#{model.selectedAlbum == null}"
    immediateUpload="false" acceptedTypes="jpg,jpeg">
        <a4j:support event="onuploadcomplete" reRender="filesPanel,
treeform" actionListener="#{fileWrapper.setComplete(true)}"/>
        <a4j:support event="onfileuploadcomplete" />
</rich:fileUpload>
...
```

The `"allowFlash"` attribute set to `"true"` enables the Flash module.

The `"acceptedTypes"` attribute specifies `"jpg"`, `"jpeg"` as the permitted file types you can upload.

The `"fileUploadListener"` attribute represents the action listener method `listener()` of the `FileUploadManager` class that is notified after file is uploaded. This method makes the main job on the upload:

```
...
public void listener(UploadEvent event) throws Exception {
    UploadItem item = event.getUploadItem();
    Image image = constructImage(item);
    try {
        extractMetadata(item, image);
    }
}
```

```

    } catch (Exception e1) {
        addError(item, image, Constants.FILE_PROCESSING_ERROR);
        return;
    }
    image.setAlbum(model.getSelectedAlbum());
    if(image.getAlbum() == null){
        addError(item, image, Constants.NO_ALBUM_TO_DOWNLOAD_ERROR);
        return;
    }
    try{
        if(imageAction.isImageWithThisPathExist(image)){
            image.setPath(generateNewPath(image.getPath()));
        }
        imageAction.addImage(image);
    } catch (Exception e){
        addError(item, image, Constants.IMAGE_SAVING_ERROR);
        return;
    }
    if(!fileManager.addImage(image.getFullPath(), item.getFile().getPath())){
        addError(item, image, Constants.FILE_SAVE_ERROR);
        return;
    }
    fileWrapper.GetFiles().add(image);
    Events.instance().raiseEvent(Constants.IMAGE_ADDED_EVENT, image);
    item.getFile().delete();
}
...

```

The `listener()` method is called at server side after every file uploaded and server saves these files in a temporary folder or in RAM depending on configuration. In the Photo Album application the uploaded files are stored in the temporary folder because the value of the `createTempFile` parameter is set to `true`. See the code from the `web.xml` descriptor:

```

...
<filter>
    <filter-name>Seam Filter</filter-name>
    <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
    <init-param>
        <param-name>createTempFiles</param-name>
        <param-value>true</param-value>
    </init-param>
    <init-param>
        <param-name>maxRequestSize</param-name>

```

```
<param-value>20000000</param-value>
</init-param>
</filter>
...
```

The `listener()` method creates an `Image` object and extracts all image metadata such as Camera name, Image size, etc. It performs scaling of an image and saves six different image's dimensions in order to create thumbnails. After that the photo is added into the database the temporary file is removed.

Visit following pages at RichFaces Live Demo for more information, examples and sources on the components used in the application and described in this chapter:

- [FileUpload](http://livedemo.exadel.com/richfaces-demo/richfaces/fileUpload.jsf?c=fileUpload) [http://livedemo.exadel.com/richfaces-demo/richfaces/fileUpload.jsf?c=fileUpload] page for the `<rich:fileUpload>` component;
- [AjaxSupport](http://livedemo.exadel.com/richfaces-demo/richfaces/support.jsf?c=support) [http://livedemo.exadel.com/richfaces-demo/richfaces/support.jsf?c=support] for the `<a4j:support>` component.

4.7. Context Menus TBR

Context menus are called when you right-click on a UI element. RichFaces library provides a special component `<rich:contextMenu>` to implement this type of functionality.

Context menu is made for the following UI controls:

- Album
- Image
- Self
- User

Let's have a look at the context menu for single image is constructed.

```
...
<rich:contextMenu disableDefaultMenu="false" style="text-align:left;" rendered="#{controller.isUserImage(image)}"
  event="oncontextmenu" attached="true" submitMode="ajax" attachTo="#{mediaOutput}">
  <rich:menuItem value="#{messages['image.delete']}" limitToList="true"
    actionListener="#{confirmationPopupHelper.initImagePopup('deleteImage',messages['image.delete.confirm'])}"
    oncomplete="#{rich:component('confirmation').show()}"
    reRender="confirmation">
  </rich:menuItem>
```

```

<rich:menuItem value="#{messages['image.edit']}" limitToList="true"
    actionListener="#{controller.startEditImage(image)}"
    reRender="mainArea">
</rich:menuItem>
<rich:menuItem value="#{messages['image_show']}" limitToList="true"
    actionListener="#{controller.showImage(image)}"
    reRender="mainArea">
</rich:menuItem>
</rich:contextMenu>
...

```

That is a listing from `\includes\contextMenu\CMForImage.xhtml`. This code is included into the very bottom of `imageList.xhtml` file like this:

```

...
<ui:include src="/includes/contextMenu/CMForImage.xhtml" >
    <ui:param name="image" value="#{image}" />
    <ui:param name="mediaOutput" value="#{rich:clientId('img')}" />
</ui:include>
...

```

The context menu code is included with 2 parameters: "image" and "mediaOutput". The first ("image") is the name of the current image. The **<a4j:repeat>** iterates over a collection of images(see the next listing), the name of the current image is stored in the "image" variable. "mediaOutput" parameter is set with the help of `rich:clientId('id')` that returns client id by short id or null if the component with the id specified hasn't been found.

This is the block of code that displays each image:

```

...
<a4j:repeat id="imageList" value="#{model.images}" var="image" rows="20">

    <h:panelGroup layout="block" styleClass="#{imageSizeHelper.currentDimension.cssClass}"
        styleClass="#{imageSizeHelper.currentDimension.imageBgStyle}"
        styleClass="#{imageSizeHelper.currentDimension.imageBgStyle}"
    >

        <h:panelGrid cellpadding="0">
            <h:panelGroup>
                <a4j:commandLink
                    actionListener="#{controller.showImage(image)}"
                    reRender="mainArea, treePanel">
                        <a4j:mediaOutput id="img" element="img"

```

```

        createContent="#{imageLoader.paintImage}"
        style="border : 1px solid #FFFFFF;"
        value="#{fileManager.transformPath(image.fullPath,
imageSizeHelper.currentDimension.filePostfix)}">
        <f:param value="#{imageSizeHelper.currentDimension.x}" name="x" />

        <rich:dragSupport rendered="#{controller.isUserImage(image)}" reRender="mainArea,
treePanel" id="dragSource" dragIndicator="dragIndicator"
        dragType="image" dragValue="#{image}">
            <rich:dndParam id="dragParam" name="label" value="#{image.name}" />
        </rich:dragSupport>
        </a4j:mediaOutput>
    </a4j:commandLink>
    <br/>
</h:panelGroup>
</h:panelGrid>
<h:panelGroup layout="block" styleClass="photo_name">#{image.name}</h:panelGroup>

    <h:panelGroup layout="block" styleClass="photo_data">
        <h:outputText value="#{image.created}">
            <f:convertDateTime />
        </h:outputText>
    </h:panelGroup>
</h:panelGroup>
<ui:include src="/includes/contextMenu/CMForImage.xhtml" >
    <ui:param name="image" value="#{image}" />
    <ui:param name="mediaOutput" value="#{rich:clientId('img')}" />
</ui:include>
</a4j:repeat>

...

```

The key attribute of **<contextMenu>** is *attachTo* that specifies for which control the context menu is displayed. As you can see this attribute has `#{mediaOutput}` as its value(`attachTo="#{mediaOutput}"`) so this way the id of the current image is passed to **<rich:contextMenu>** and this is how it know what photo is affected by user actions.

4.8. ToolTips TBR

When using RichFaces components library you've got nearly everything to build UI, making a tooltip is not an exception. RichFaces provides a separate component to make a bubble appear when the user hovers a UI element or layout area. The component is **<rich:toolTip>**. There's nothing complicated in using **<rich:toolTip>**: you just need to set the text to be shown in the tooltip with

the *"value"* attribute and specify for which component you want the tooltip to be shown with the *"for"* attribute that takes the id of the targeted component as a parameter.

```
...
<rich:panel id="panelId">
  <p>Element which has a tooltip</p>
</rich:panel>
<rich:toolTip value="This is a tooltip." for="panelId"/>
...
```

Alternatively, you can just place **<rich:toolTip>** as a nested element of container for which the tooltip is shown.

```
...
<div>
  <p>Element which has a tooltip</p>

  <rich:toolTip>
    <p>Tooltip text</p>
  </rich:toolTip>
</div>
...
```

This approach was adopted in the Photo Alum to display tooltips for

This code outputs an album's image.

```
...
<a4j:mediaOutput id="img" element="img" styleClass="main-image"
  createContent="#{imageLoader.paintImage}" style="border : 3px solid #808080;"
  value="#{fileManager.transformPath(model.selectedImage.fullPath, '_medium')}"/>

</rich:dragSupport>
  <rich:toolTip followMouse="true" direction="top-right"
    showDelay="500" styleClass="tooltip" rendered="#{model.selectedImage.showMetaInfo}">
    <span style="white-space:nowrap; display:block; text-align:left;">
      <h:outputText value="Size in bytes: #{model.selectedImage.size}" />
      <br />
      <h:outputText
        value="#{messages['original_size']}:
        #{model.selectedImage.height}x#{model.selectedImage.width}" />
    </span>
  </rich:toolTip>
</a4j:mediaOutput>
```

```
<br />
<h:outputText value="Captured at: #{model.selectedImage.created}" />
<br />
<h:outputText
    value="#{messages['camera']}: #{model.selectedImage.cameraModel}" />
</span>
</rich:toolTip>
</a4j:mediaOutput>
...
```

The **<rich:toolTip>** is nested in **<a4j:mediaOutput>** and prints the size of the image, size in pixels, when the picture was taken and the type of camera used to take the picture.



Figure 4.7. Tooltip

4.9. User Input Data Validation

Validation of user input is a very frequent situation for a developer. RichFaces library offers 3 component to get this job done: `<rich:beanValidator>`, `<rich:graphValidator>`, and `<rich:ajaxValidator>`. The latter two components are used in the Photo Album application. `<rich:graphValidator>` is intended to validate the whole object or the graph of interrelated objects and the validation occurs when the whole form is submitted. While `<rich:ajaxValidator>` validates only one input field or a value at a time, validation is activated upon some event and adds

interactivity to the application. Both components use Hibernate validators which helps to locate validation logic in one place, such approach is really helpful given that usually data validation logic is stored in multiple places including UI pages and in Java code that interacts with a database.

Let's have a look at the components usage on the registration page. This is how the page looks like (some irrelevant details were removed from the example):

```
...
<ui:composition xmlns="http://www.w3.org/1999/xhtml" ...

<rich:graphValidator>
    ...
    <h:inputText id="loginName" value="#{user.login}" />
    <rich:messages for="loginName" />
    <h:inputSecret required="true" id="password" value="#{user.password}" />
    <rich:messages for="password" />
    <h:inputSecret required="true" id="confirmPassword"
        value="#{user.confirmPassword}" />
    <rich:messages for="confirmPassword" />
    <h:inputText id="firstname" value="#{user.firstName}" />
    <rich:messages for="firstname" style="color:red;" />
    <h:inputText id="secondname" value="#{user.secondName}" />
    <rich:messages for="secondname" />
    <h:selectOneRadio required="true" id="sex" value="#{user.sex}">
        <f:selectItems value="#{userPrefsBean.sexes}" />
        <s:convertEnum />
    </h:selectOneRadio>
    <rich:messages for="sex" />
    <a4j:outputPanel id="calendar" layout="block">
        <rich:calendar id="birthDate" value="#{user.birthDate}"

        <rich:ajaxValidator event="oninputblur" />
        </rich:calendar>
    </a4j:outputPanel>
    <rich:messages for="birthDate" />
    <h:inputText id="email" value="#{user.email}" />
    <rich:messages for="email" />
</rich:graphValidator>
<richx:commandButton actionListener="#{authenticator.register(user)}"
    value="Register" />
<richx:commandButton actionListener="#{controller.cancelRegistration()}"
    immediate="true" value="Cancel" />
</ui:composition>
```

...

<rich:graphValidator> validates the entity User object, in which restrictions are set with the help of Hibernate annotations. When the **Register** button is clicked on the name, password, sex etc. fields are validated sequentially. In case of an error (for example, if a loginName contains only one character and the annotation restricts it to at least 3 characters to be typed in) an error message in red color is displayed next to the input field and the request is aborted. If all values are valid the `authenticator.register(user)` method will be invoked and the user will be saved to the database.

<rich:ajaxValidator> acts in a slightly different way, in our case it is attached to the `user.birthDate` field. When the value of the field is changed and the field loses focus it is immediately validated. If the input data is incorrect an error message will be displayed, which is a quick way to respond to user input errors and avoid sending incorrect data to the server.

If you would like to get more details about the validators that RichFaces library provides please visit [Live Demo](http://livedemo.exadel.com/richfaces-demo/richfaces/ajaxValidator.jsf) [http://livedemo.exadel.com/richfaces-demo/richfaces/ajaxValidator.jsf] web page and [RichFaces Developer Guide](http://richfaces.org/devguide/html_single/index.html) [http://richfaces.org/devguide/html_single/index.html].

4.10. How the button is created and how it acts

Due to specific design of the buttons in the Photo Album application, it's necessary to clarify some design and development points about the button. The button is visually represented by Facelets template stored in the `button.xhtml` file.

Please have a look at the content of the file:

...

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:s="http://jboss.com/products/seam/taglib"
    xmlns:c="http://java.sun.com/jstl/core"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:rich="http://richfaces.org/rich"
    xmlns:a4j="http://richfaces.org/a4j"
    xmlns:richx="http://richfaces.org/richx">
    <a4j:loadScript src="/scripts/buttons.js" />
    <richx:actionMapper>
        <a4j:outputPanel layout="block" style="#{style}" styleClass="photoalbumButton
        #{styleClass}" lang="#{lang}" dir="#{dir}" title="#{title}"
            rendered="#{empty rendered or rendered}"
            onmousedown="RF_RW_DEMO.toPressed(this)" onmouseup="RF_RW_DEMO.toReleased(this)" onmouseover="RF_RW_DEMO.toOver(this)" onmouseout="RF_RW_DEMO.toOut(this)" />
        <h:graphicImage value="/img/shell/button.png" alt="" />
        <h:graphicImage value="/img/shell/button_press.png" style="display: none;" alt="" />
    </richx:actionMapper>
</ui:composition>
```

```

<div>#{value}</div>

<a4j:commandButton accessKey="#{accessKey}" ajaxSingle="#{ajaxSingle}" alt="#{alt}" type="image" image="/
img/shell/spacer.gif"
    actionListener="#{mappedActionListener}" action="#{mappedAction}" bypassUpdates="#{bypassUpdates}"
    eventsQueue="#{eventsQueue}" focus="#{focus}" ignoreDupResponses="#{ignoreDupResponses}"
    onbeforeDOMupdate="#{onbeforeDOMupdate}" timeout="#{timeout}" tabIndex="#{tabIndex}" status="#{status}"
    reRender="#{reRender}" requestDelay="#{requestDelay}" process="#{process}" onComplete="#{onComplete}"
    onblur="#{onblur}" onclick="#{onclick}" ondblclick="#{ondblclick}" onfocus="#{onfocus}" onkeydown="#{onkeydown}"
    onkeypress="#{onkeypress}" onkeyup="#{onkeyup}" onmouseover="#{onmouseover}" onmouseout="#{onmouseout}" onmousemove="#{onmousemove}"
    onmousedown="#{onmousedown}" onmouseup="#{onmouseup}"
/>
</a4j:outputPanel>
</richx:actionMapper>
</ui:composition>

...

```

The `<richx:actionMapper>` tag is covered in more detail further in the text. In brief, it's a special tag developed deliberately to pass to the button a method expression of the action which must be performed when the button is clicked.

To make sure the button works correctly we include the required JavaScript code that is located in the button.js file using `<a4j:loadScript src="/scripts/buttons.js" />` component.

The button consists of several parts:

- 2 images (pressed and not pressed)
- `<div>` element that displays the button's text
- `<a4j:commandButton>` that sends Ajax request to the server

These elements are wrapped by `<a4j:outputPanel>` to adjust the look-and-feel.

In the application the button is used for example like this:

```

...
<richx:commandButton actionListener="#{authenticator.register(user)}" reRender="mainform,
headerPanel" value="#{messages['user.register']}" />
...

```

We can pass to the `<richx:commandButton>` all required attributes, in the example only `actionListener`, `reRender` and `value` are passed.

`<richx:commandButton>` is a custom tag that is declared in the photoalbum-taglib.xml tag library:

```
...
<?xml version="1.0"?>
<!DOCTYPE facelet-taglib PUBLIC
  "-//Sun Microsystems, Inc.//DTD Facelet Taglib 1.0//EN"
  "facelet-taglib_1_0.dtd">
<facelet-taglib>
  <namespace>http://richfaces.org/richx</namespace>
  <tag>
    <tag-name>commandButton</tag-name>
    <source>templates/button.xhtml</source>
  </tag>
  <tag>
    <tag-name>actionMapper</tag-name>
    <handler-class>org.richfaces.photoalbum.util.ActionMapperTagHandler</handler-class>
  </tag>
</facelet-taglib>
...
```

In order to use the **<richx:commandButton>** on the page the namespace of the taglib must be declared:

```
...
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
  ...
  xmlns:richx="http://richfaces.org/richx">
  ...
```

A more complex part of the button implementation, as we said earlier, is **<richx:actionMapper>** which is also described in tablib. But it is not just a simple tag-template since it has Facelets handler-class which specifies how it is handled when declared on the page. It is created because Facelets templates do not allow to make the MethodExpression a Facelets-template parameter. Please find below the code of the class(some irrelevant details are omitted):

```
...
public class ActionMapperTagHandler extends TagHandler {

  private static final Class<?>[] ACTION_PARAM_TYPES = new Class[0];

  private static final Class<?>[] ACTION_LISTENER_PARAM_TYPES = new Class[] {ActionEvent.class};

  private static final String ACTION = "action";
```

```
private static final String ACTION_LISTENER = "actionListener";

private static final String MAPPED_ACTION = "mappedAction";

private static final String MAPPED_ACTION_LISTENER = "mappedActionListener";

public ActionMapperTagHandler(TagConfig config) {
    super(config);
}

private MethodExpression remap(FaceletContext faceletContext, String varName,
    Class<?> expectedReturnType, Class<?>[] expectedParamTypes) {

    MethodExpression result = null;

    VariableMapper mapper = faceletContext.getVariableMapper();
    ValueExpression valueExpression = mapper.resolveVariable(varName);
    if (valueExpression != null) {
        ExpressionFactory ef = faceletContext.getExpressionFactory();
        ELContext elContext = faceletContext.getFacesContext().getELContext();

        result = ef.createMethodExpression(elContext, valueExpression.getExpressionString(),
            expectedReturnType, expectedParamTypes);
    }

    return result;
}

public void apply(FaceletContext ctx, UIComponent parent)
    throws IOException, FacesException, FaceletException, ELException {

    MethodExpression actionExpression = remap(ctx, ACTION, String.class, ACTION_PARAM_TYPES);
    MethodExpression actionListenerExpression = remap(ctx, ACTION_LISTENER, null, ACTION_LISTENER_PA

    if (actionExpression != null || actionListenerExpression != null) {
        VariableMapper initialVarMapper = ctx.getVariableMapper();
        try {
            VariableMapperWrapper varMapper = new VariableMapperWrapper(initialVarMapper);

            if (actionExpression == null) {
                actionExpression = NOOP_ACTION_EXPRESSION;
            }
        }
    }
}
```



```
varMapper.setVariable(MAPPED_ACTION,
    ctx.getExpressionFactory().createValueExpression(actionExpression,
        MethodExpression.class));

if (actionListenerExpression == null) {
    actionListenerExpression = NOOP_ACTION_LISTENER_EXPRESSION;
}

varMapper.setVariable(MAPPED_ACTION_LISTENER,
    ctx.getExpressionFactory().createValueExpression(actionListenerExpression,
        MethodExpression.class));

ctx.setVariableMapper(varMapper);

nextHandler.apply(ctx, parent);

} finally {
    ctx.setVariableMapper(initialVarMapper);
}
} else {
    nextHandler.apply(ctx, parent);
}
}
}

...
```

You can find more information about Facelets, custom tags, taglibs, Facelets tag handlers and Facelets templates [here](#).

4.11. The `<a4j:status>` component

`<a4j:status>` is a component, designed to create some visual effect during Ajax request. The component is usually attached to a certain request, which implies time consuming processing, so that the end user is aware the page is not hung up, it's responding to her actions: the user sees the processing progress (the component is frequently used to indicate file uploading process).

However, in the Photo Album application the `<a4j:status>` component is triggered on any Ajax request: to demonstrate the component itself and partially to display for the user that on every click on a link or a button something is happening, as all actions in the application occur on a single page which is not typical for usual web-workflow, when on each action the user navigates to a new page.

By default, `<a4j:status>` works for each Ajax components inside the local region. This means if you have no region defined on the page (the whole view is a region) and have only one `<a4j:status>`

on the page, the **<a4j:status>** will be activated during Ajax request by any of the Ajax component located on the page.

As there are no regions defined explicitly in the application, **<a4j:status>** is located in the main template (template.xhtml) for all pages:

```
...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    ...
    <body class="main-body">
        ...
        <ui:include src="/includes/index/status.xhtml" />
    </body>
</html>
...
```

Hence the default behavior of the component meets that application's requirements: the component is shown on every single Ajax request.

This is the page with the **<a4j:status>** component:

```
...
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:s="http://jboss.com/products/seam/taglib"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:rich="http://richfaces.org/rich"
    xmlns:a4j="http://richfaces.org/a4j"
    xmlns:richx="http://richfaces.org/richx">
    <a4j:status layout="block" stopStyle="display: none;"
        startStyle="height: 52px; width: 79px; position: absolute; top: 0px; left: 278px;">
        <f:facet name="start">
            <h:panelGroup>
                <h:graphicImage style="position: absolute; top: 0px; left: 0px;"
                    height="79" width="52" alt="" value="/img/shell/ai.png" />
                <h:graphicImage style="position: absolute; top: 26px; left: 13px;"
                    height="26" width="26" alt="" value="/img/shell/ai.gif" />
            </h:panelGroup>
        </f:facet>
    </a4j:status>
</ui:composition>
```

```
</a4j:status>
</ui:composition>
...
```

The `startStyle="height: 52px; width: 79px; position: absolute; top: 0px; left: 278px; "` attribute specifies what is displayed on the page after the request initiation, which means in our case that it is positioned absolutely: 278 pixels from the left border of the page and 0 pixels from the top, its width is 79px and the height is 52px.

The `stopStyle="display: none; "` attribute is responsible for displaying the component on the page when the request is finished, in our case the it will be hidden.

As we need to show only the beginning of the request, we customize only the `<f:facet name="start">` which is just an image(you can insert any image you like).

If you would like to get more details about the `<a4j:status>` please visit [Live Demo](http://livedemo.exadel.com/richfaces-demo/richfaces/status.jsf?c=status) [http://livedemo.exadel.com/richfaces-demo/richfaces/status.jsf?c=status] web page and [RichFaces Developer Guide](#) [../devguide/html_single/index.html/#a4j_status].

4.12. Errors Reports

The Photo Album application has a global mechanism for errors checking. You become informed about the error each time it occurs. It is possible because the main page of the application `web/src/main/webapp/index.xhtml` includes the `web/src/main/webapp/includes/misc/errorPanel.xhtml`:

```
...
<a4j:outputPanel id="errors" ajaxRendered="true">
    <h:panelGroup rendered="#{errorHandlerBean.errorExist}">

        <rich:modalPanel id="errorPanel" showWhenRendered="true" minWidth="300" minHeight="200" autosized="tr
            ...
        </rich:modalPanel>
    </h:panelGroup>
</a4j:outputPanel>
...
```

Here is the listing of the `errorHandlerBean` class:

```
package org.richfaces.photoalbum.ui;
import java.util.ArrayList;
import java.util.List;
import org.jboss.seam.ScopeType;
```

```
import org.jboss.seam.annotations.AutoCreate;
import org.jboss.seam.annotations.Name;
import org.jboss.seam.annotations.Observer;
import org.jboss.seam.annotations.Scope;
import org.richfaces.photoalbum.service.Constants;

@Name("errorHandlerBean")
@Scope(ScopeType.EVENT)
@AutoCreate
public class ErrorHandlerBean {
    private List<String> errors = new ArrayList<String>();

    public List<String> getErrors() {
        return errors;
    }

    public boolean isErrorExist(){
        return errors.size() > 0 ;
    }

    @Observer(Constants.ADD_ERROR_EVENT)
    public void addToErrors(String e){
        errors.add(e);
    }
}
```

The error panel contains the `<a4j:outputPanel>` component that is rendered on every Ajax request (`ajaxRendered="true"`). If an error is occurred the `isErrorExist()` method of `errorHandlerBean` class returns "true", so the `<h:panelGroup>` component is rendered. In order to show nested `<rich:modalPanel>` component with the collected errors the `"showWhenRendered"` attribute should be set to "true".

The `addToErrors()` method is annotated with `@Observer` annotation, thus it observes all events with `ADD_ERROR_EVENT` constant and adds errors to the `errors` List.