Component Reference

A reference guide to the components of the RichFaces *(draft)* framework

by Sean Rogers (Red Hat)

Introduction

This book is a guide to the various components available in the RichFaces 4.0 framework. It includes descriptions of the role of the components, details on how best to use them, coded examples of their use, and basic references of their properties and attributes.

For full in-depth references for all component classes and properties, refer to the *API Reference* available from the RichFaces website.

1.1. Libraries

The RichFaces framework is made up of two tag libraries: the `a4j` library and the `rich` library. The `a4j` tag library represents Ajax4jsf, which provides page-level Ajax support with core Ajax components. This allows developers to make use of custom Ajax behavior with existing components. The `rich` tag library provides Ajax support at the component level instead, and includes ready-made, self-contained components. These components don't require additional configuration in order to send requests or update.

Ajax support

All components in the `a4j` library feature built-in Ajax support, so it is unnecessary to add the `<a4j:support>` behavior.

Common Ajax attributes

The Ajax components in the `a4j` library share common attributes to perform similar functionality. Most RichFaces components in the `rich` library that feature built-in Ajax support share these common attributes as well.

Most attributes have default values, so they need not be explicitly set for the component to function in its default state. These attributes can be altered to customize the behavior of the component if necessary.

2.1. Rendering

2.1.1. `render`

The `render` attribute provides a reference to one or more areas on the page that need updating after an Ajax interaction. It uses the `UIComponent.findComponent()` algorithm to find the components in the component tree using their `id` attributes as a reference. Components can be referenced by their `id` attribute alone, or by a hierarchy of components' `id` attributes to make locating components more efficient. *Example 2.1, "render example"* shows both ways of referencing components. Each command button will correctly render the referenced panel grids, but the second button locates the references more efficiently with explicit hierarchy paths.

Example 2.1. render example

```
<h:form id="form1">
   <a4j:commandButton value="Basic reference" render="infoBlock, infoBlock2" />
   <a4j:commandButton value="Specific reference" render=":infoBlock,:sv:infoBlock2" />
</h:form>

<h:panelGrid id="infoBlock">
   ...
</h:panelGrid>

<f:subview id="sv">
   <h:panelGrid id="infoBlock2">
      ...
   </h:panelGrid>
</f:subview>
```

The value of the `render` attribute can also be an expression written using JavaServer Faces' Expression Language (EL); this can either be a Set, Collection, Array, or String.

`rendered` attributes

A common problem with using `render` occurs when the referenced component has a `rendered` attribute. JSF does not mark the place in the browser's Document Object Model (DOM) where the rendered component would be placed in case the `rendered` attribute returns `false`. As such, when RichFaces sends the render code to the client, the page does not update as the place for the update is not known.

To work around this issue, wrap the component to be rendered in an `<a4j:outputPanel>` with `layout="none"`. The `<a4j:outputPanel>` will receive the update and render the component as required.

2.1.2. `ajaxRendered`

A component with `ajaxRendered="true"` will be re-rendered with *every* Ajax request, even when not referenced by the requesting component's `render` attribute. This can be useful for updating a status display or error message without explicitly requesting it.

Rendering of components in this way can be repressed by adding `limitRender="true"` to the requesting component, as described in *Section 2.1.3, "limitRender"*.

2.1.3. `limitRender`

A component with `limitRender="true"` specified will *not* cause components with `ajaxRendered="true"` to re-render, and only those components listed in the `render` attribute will be updated. This essentially overrides the `ajaxRendered` attribute in other components.

Example 2.3, "Data reference example" describes two command buttons, a panel grid rendered by the buttons, and an output panel showing error messages. When the first button is clicked, the output panel is rendered even though it is not explicitly referenced with the `render` attribute. The second button, however, uses `limitRender="true"` to override the output panel's rendering and only render the panel grid.

Example 2.2. Rendering example

```
<h:form id="form1">
   <a4j:commandButton value="Normal rendering" render="infoBlock" />
   <a4j:commandButton value="Limited rendering" render="infoBlock" limitRender="true" />
</h:form>

<h:panelGrid id="infoBlock">
   ...
</h:panelGrid>
```

```
<a4j:outputPanel ajaxRendered="true">
   <h:messages />
</a4j:outputPanel>
```

2.2. Queuing and traffic control

2.2.1. `queue`

The `queue` attribute defines the name of the queue that will be used to schedule upcoming Ajax requests. Typically RichFaces does not queue Ajax requests, so if events are produced simultaneously they will arrive at the server simultaneously. This can potentially lead to unpredictable results when the responses are returned. The `queue` attribute ensures that the requests are responded to in a set order.

A queue name is specified with the `queue` attribute, and each request added to the named queue is completed one at a time in the order they were sent. In addition, RichFaces intelligently removes similar requests produced by the same event from a queue to improve performance, protecting against unnecessary traffic flooding and

2.2.2. `requestDelay`

The `requestDelay` attribute specifies an amount of time in milliseconds for the request to wait in the queue before being sent to the server. If a similar request is added to the queue before the delay is over, the original request is removed from the queue and not sent.

2.2.3. `ignoreDupResponses`

When set to `true`, the `ignoreDupResponses` attribute causes responses from the server for the request to be ignored if there is another similar request in the queue. This avoids unnecessary updates on the client when another update is expected. The request is still processed on the server, but if another similar request has been queued then no updates are made on the client.

2.3. Data processing

RichFaces uses a form-based approach for sending Ajax requests. As such, each time a request is sent the data from the requesting component's parent JSF form is submitted along with the XMLHTTPRequest object. The form data contains values from the input element and auxiliary information such as state-saving data.

2.3.1. `execute`

The `execute` attribute allows JSF processing to be limited to defined components. To only process the requesting component, `execute="@this"` can be used.

2.3.2. `immediate`

If the `immediate` attribute is set to `true`, the default ActionListener is executed immediately during the Apply Request Values phase of the request processing lifecycle, rather than waitingfor the Invoke Application phase. This allows some data model values to be updated regardless of whether the Validation phase is successful or not.

2.3.3. `bypassUpdates`

If the `bypassUpdates` attribute is set to `true`, the Update Model phase of the request processing lifecycle is bypassed. This is useful if user input needs to be validated but the model does not need to be updated.

2.4. Action and navigation

The `action` and `actionListener` attributes can be used to invoke action methods and define action events.

2.4.1. `action`

The `action` attribute is a method binding that points to the application action to be invoked. The method can be activated during the Apply Request Values phase or the Invoke Application phase of the request processing lifecycle.

The method specified in the `action` attribute must return `null` for an Ajax response with a partial page update.

2.4.2. `actionListener`

The `actionListener` attribute is a method binding for `ActionEvent` methods with a return type of `void`.

2.5. Events and JavaScript interactions

RichFaces allows for Ajax-enabled JSF applications to be developed without using any additional JavaScript code. However it is still possible to invoke custom JavaScript code through Ajax events.

2.5.1. `onsubmit`

The `onsubmit` attribute invokes the JavaScript code *before* the Ajax request is sent. The request is canceled if the JavaScript code defined for `onsubmit` returns `false`.

2.5.2. `onbegin`

The `onbegin` attribute invokes the JavaScript code *after* the Ajax request is sent.

2.5.3. `onclick`

The `onclick` attribute functions similarly to the `onsubmit` attribute for those components that can be clicked, such as `<a4j:commandButton>` and `<a4j:commandLink>`. It invokes the defined JavaScript before the Ajax request, and the request will be canceled if the defined code returns `false`.

2.5.4. `onsuccess`

The `onsuccess` attribute invokes the JavaScript code after the Ajax response has been returned but *before* the DOM tree of the browser has been updated.

2.5.5. `oncomplete`

The `oncomplete` attribute invokes the JavaScript code after the Ajax response has been returned *and* the DOM tree of the browser has been updated.

Reference consistency

The code is registered for further invocation of the XMLHTTP request object before an Ajax request is sent. As such, using JSF Expression Language (EL) value binding means the code will not be changed during processing of the request on the server. Additionally the `oncomplete` attribute cannot use the `this` keyword as it will not point to the component from which the Ajax request was initiated.

2.5.6. `onerror`

The `onerror` attribute invokes the JavaScript code when an error has occurred during Ajax communications.

2.5.7. `data`

The `data` attribute allows the use of additional data during an Ajax call. JSF Expression Language (EL) can be used to reference the property of the managed bean, and its value will be serialized in JavaScript Object Notation (JSON) and returned to the client side. The property can then be referenced through the `data` variable in the event attribute definitions. Both primitive types and complex types such as arrays and collections can be serialized and used with `data`.

Example 2.3. Data reference example

```
<a4j:commandButton value="Update" data="#{userBean.name}" oncomplete="showTheName(data.name)"
>
```

Common features

This chapter covers those attributes and features that are common to many of the components in the tag libraries.

3.1. Positioning and appearance of components

A number of attributes relating to positioning and appearance are common to several components.

`disabled`
: Specifies whether the component is disabled, which disallows user interaction.

`focus`
: References the `id` of an element on which to focus after a request is completed on the client side.

`height`
: The height of the component in pixels.

`dir`
: Specifies the direction in which to display text that does not inherit its writing direction. Valid values are `LTR` (left-to-right) and `RTL` (right-to-left).

`style`
: Specifies Cascading Style Sheet (CSS) styles to apply to the component.

`styleClass`
: Specifies one or more CSS class names to apply to the component.

`width`
: The width of the component in pixels.

3.2. Calling available JavaScript methods

Client-side JavaScript methods can be called using component events. These JavaScript methods are defined using the relevant event attribute for the component tag. Methods are referenced through typical Java syntax within the event attribute, while any parameters for the methods are obtained through the `data` attribute, and referenced using JSF Expression Language (EL). *Example 2.3, "Data reference example"* a simple reference to a JavaScript method with a single parameter.

Refer to *Section 2.5, "Events and JavaScript interactions"* or to event descriptions unique to each component for specific usage.

Part I. a4j tag library

Actions

This chapter details the basic components that respond to a user action and submit an Ajax request.

4.1. `<a4j:actionParam>`

- component-type: `org.ajax4jsf.ActionParameter`
- component-class: `org.ajax4jsf.component.html.HTMLActionParameter`

The `<a4j:actionParam>` behavior combines the functionality of the JavaServer Faces (JSF) components `<f:param>` and `<f:actionListener>`.

Basic usage of the `<a4j:actionParam>` requires three main attributes:

- `name`, for the name of the parameter;
- `value`, for the initial value of the parameter; and
- `assignTo`, for defining the bean property. The property will be updated if the parent command component performs an action event during the *Process Request* phase.

Example 4.1, "<a4j:actionParam> example" shows a simple implementation along with the accompanying managed bean. When the **Set name to Alex** button is pressed, the application sets the `name` parameter of the bean to `Alex`, and displays the name in the output field.

Example 4.1. `<a4j:actionParam>` example

```
<h:form id="form">
   <a4j:commandButton value="Set name to Alex" reRender="rep">
     <a4j:actionparam name="username" value="Alex" assignTo="#{actionparamBean.name}"/>
   </a4j:commandButton>
   <h:outputText id="rep" value="Name: #{actionparamBean.name}"/>
</h:form>
```

```
public class ActionparamBean {
   private String name = "John";

   public String getName() {
```

```
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

The `<a4j:actionParam>` behavior can be used with non-Ajax components in addition to Ajax components. In this way, data model values can be updated without an JavaScript code on the server side.

The `converter` attribute can be used to specify how to convert the value before it is submitted to the data model. The property is assigned the new value during the *Update Model* phase.

Validation failure

If the validation of the form fails, the *Update Model* phase will be skipped and the property will not be updated.

Variables from JavaScript functions can be used for the `value` attribute. In such an implementation, the `noEscape` attribute should be set to `true`. Using `noEscape="true"`, the `value` attribute can contain any JavaScript expression or JavaScript function invocation, and the result will be sent to the server as the `value` attribute.

4.2. `<a4j:ajaxListener>`

- listener-class: `org.ajax4jsf.event.AjaxListener`
- event-class: `org.ajax4jsf.event.AjaxEvent`
- tag-class: `org.ajax4jsf.taglib.html.jsp.AjaxListenerTag`

The `<a4j:ajaxListener>` component adds an action listener to a parent component. It works similar to the JavaServer Faces `<f:actionListener>` or `<f:valueChangeListener>` components, except that the invocation of `<a4j:ajaxListener>` is not canceled if validation of the *Update Model* phase fails. The `<a4j:ajaxListener>` component is guaranteed to be invoked with each Ajax response.

Basic usage requires only the `type` attribute, which defines the fully-qualified Java class name for the listener. This Java class should implement the `org.ajax4jsf.event.AjaxListener` interface, which is a base listener for all listeners and is capable of receiving Ajax events. The object from which the event originated could be accessed using the `java.util.EventObject.getSource()` method.

The <a4j:ajaxListener> component is not invoked for non-Ajax requests, or when the RichFaces works in the *Ajax request generates non-Ajax response* mode, so the <a4j:ajaxListener> invocation is a good indicator that an Ajax response is going to be processed.

4.3. <a4j:commandButton>

- component-type: org.ajax4jsf.CommandButton
- component-family: javax.faces.Command
- component-class: org.ajax4jsf.component.html.HtmlAjaxCommandButton
- renderer-type: org.ajax4jsf.components.AjaxCommandButtonRenderer

Figure 4.1. <a4j:commandButton>

The <a4j:commandButton> is similar to the JavaServer Faces (JSF) component <h:commandButton>, but additionally includes Ajax support. When the command button is clicked it generates an Ajax form submit, and when a response is received the command button can be dynamically rendered.

The <a4j:commandButton> requires only the value and render attributes to function. The value attribute specifies the text of the button and the render attribute specifies which areas are to be updated. The <a4j:commandButton> uses the onclick event instead of the onsubmit event, but otherwise uses all common Ajax attributes as listed in *Chapter 2, Common Ajax attributes*.

Set disabledDefault="true"

When attaching a JavaScript function to a <a4j:commandButton> with the help of a <rich:componentControl>, do not use the attachTo attribute of <rich:componentControl>. The attribute adds event handlers using Event.observe but <a4j:commandButton> does not include this event.

4.4. <a4j:commandLink>

- component-type: org.ajax4jsf.CommandLink
- component-family: javax.faces.Command
- component-class: org.ajax4jsf.component.html.HtmlAjaxCommandLink
- renderer-type: org.ajax4jsf.components.AjaxCommandLinkRenderer

Figure 4.2. `<a4j:commandLink>`

The `<a4j:commandLink>` is similar to the JavaServer Faces (JSF) component `<h:commandLink>`, but additionally includes Ajax support. When the command link is clicked it generates an Ajax form submit, and when a response is received the command link can be dynamically rendered.

The `<a4j:commandLink>` requires only the `value` and `render` attributes to function. The `value` attribute specifies the text of the link and the `render` attribute specifies which areas are to be updated. The `<a4j:commandLink>` uses the `onclick` event instead of the `onsubmit` event, but otherwise uses all common Ajax attributes as listed in *Chapter 2, Common Ajax attributes*.

4.5. `<rich:componentControl>`

The following reference data is taken from the old `<rich:componentControl>` reference. The details may be different now that the component is part of the `a4j` tag library.

- component-type: `org.richfaces.ComponentControl`
- component-family: `org.richfaces.ComponentControl`
- component-class: `org.richfaces.component.html.HtmlComponentControl`
- renderer-type: `org.richfaces.ComponentControlRenderer`
- tag-class: `org.richfaces.taglib.ComponentControlTag`

The `<rich:componentControl>` allows JavaScript API functions to be called on components after defined events. Initialization variants and activation events can be customized, and parameters can be passed to the target component.

The `event`, `for`, and `operation` attributes are all that is required to attach JavaScript functions to the parent component. The `event` attribute specifies the event that triggers the JavaScript API function call. The `for` attribute defines the target component, and the `operation` attribute specifies the JavaScript function to perform.

Example 4.2. rich:componentControl basic usage

```
<h:commandButton value="Show Modal Panel">
   <!--componentControl is attached to the commandButton-->
   <rich:componentControl for="ccModalPanelID" event="onclick" operation="show"/>
</h:commandButton>
```

The example contains a single command button, which when clicked shows the modal panel with the identifier `ccModalPanelID`.

The `attachTo` attribute can be used to attach the event to a component other than the parent component. If no `attachTo` attribute is supplied, the `<rich:componentControl>` component's parent is used, as in *Example 4.2, "rich:componentControl basic usage"*.

Example 4.3. Attaching `<rich:componentControl>` to a component

```
<rich:componentControl attachTo="doExpandCalendarID" event="onclick" operation="Expand" for="ccCalendarID"
>
```

In the example, the `onclick` event of the component with the identifier `ccCalendarID` will trigger the `Expand` operation for the component with the identifier `doExpandCalendarID`.

The operation can receive parameters either through the `params` attribute, or by using `<f:param>` elements.

Example 4.4. Using parameters

The `params` attribute

```
<rich:componentControl name="func" event="onRowClick" for="menu" operation="show" params="#{car.model}"/
>
```

`<f:param>` elements

```
<rich:componentControl event="onRowClick" for="menu" operation="show">
   <f:param value="#{car.model}" name="model"/>
</rich:componentControl>
```

The `name` attribute can be used to define a normal JavaScript function that triggers the specified operation on the target component.

The `attachTiming` attribute can determine the page loading phase during which the `<rich:componentControl>` is attached to the source component:

`immediate`
: attached during execution of the script.

`onavailable`
: attached after the target component is initialized.

`onload`
: attached after the page is loaded.

4.6. <a4j:hashParam>

Incomplete

4.7. <a4j:htmlCommandLink>

- component-type: `org.ajax4jsf.HtmlCommandLink`
- component-family: `javax.faces.Command`
- component-class: `org.ajax4jsf.component.html.HtmlCommandLink`
- renderer-type: `org.ajax4jsf.HtmlCommandLinkRenderer`

The `<a4j:htmlCommandLink>` component functions similarly to the standard `<h:commandLink>` component, but addresses some of the potential issues that can occur.

When using the standard component, hidden fields were not rendered to child elements if they were deemed unnecessary, so command links relating to content on the initial page could become broken if they were later updated through Ajax. The `<a4j:htmlCommandLink>` component addresses this by always rendering all hidden fields.

4.8. <a4j:jsFunction>

- component-type: `org.ajax4jsf.Function`
- component-family: `org.ajax4jsf.components.ajaxFunction`
- component-class: `org.ajax4jsf.component.html.HtmlajaxFunction`
- renderer-type: `org.ajax4jsf.components.ajaxFunctionRenderer`

The `<a4j:jsFunction>` component allows Ajax requests to be performed directly from JavaScript code, and server-side data to be invoked and returned in JavaScript Object Notation (JSON) format to use in client-side JavaScript calls.

The `<a4j:jsFunction>` component has all the common Ajax action attributes as listed in *Chapter 2, Common Ajax attributes*; the `action` and `actionListener` attributes can be invoked and parts of the page can be re-rendered after a successful call to the JavaScript function. *Example 4.5, "<a4j:jsFunction> example"* shows how an Ajax request can be initiated from the JavaScript and a partial page update performed. The JavaScript function can be invoked with the data returned by the Ajax response.

Example 4.5. <a4j:jsFunction> example

```
<h:form>
```

```
    ...
<a4j:jsFunction name="callScript" data="#{bean.someProperty1}" reRender="someComponent" oncomplete="myScript(data.subProperty1,
 data.subProperty2)">
     <a4j:actionParam name="param_name" assignTo="#{bean.someProperty2}"/>
 </a4j:jsFunction>
 ...
</h:form>
```

The `<a4j:jsFunction>` component allows the use of the `<a4j:actionParam>` component or the JavaServer Faces `<f:param>` component to pass any number of parameters for the JavaScript function.

The `<a4j:jsFunction>` component is similar to the `<a4j:commandButton>` component, but it can be activated from the JavaScript code. This allows some server-side functionality to be invoked and the returned data to subsequently be used in a JavaScript function invoked by the `oncomplete` event attribute. In this way, the `<a4j:jsFunction>` component can be used instead of the `<a4j:commandButton>` component.

4.9. `<a4j:poll>`

- component-type: `org.ajax4jsf.Poll`
- component-family: `org.ajax4jsf.components.AjaxPoll`
- component-class: `org.ajax4jsf.component.html.AjaxPoll`
- renderer-type: `org.ajax4jsf.components.AjaxPollRenderer`

The `<a4j:poll>` component allows periodical sending of Ajax requests to the server. It is used for repeatedly updating a page at specific time intervals.

The `interval` attribute specifies the time in milliseconds between requests. The default for this value is 1000 ms (1 second).

The `timeout` attribute defines the response waiting time in milliseconds. If a response isn't received within the timeout period, the connection is aborted and the next request is sent. By default, the timeout is not set.

The `<a4j:poll>` component can be enabled and disabled using the `enabled` attribute. Using Expression Language (EL), the `enabled` attribute can point to a bean property to apply a particular attribute value.

4.10. `<a4j:push>`

- component-type: `org.ajax4jsf.Push`

- component-family: `org.ajax4jsf.components.AjaxPush`
- component-class: `org.ajax4jsf.component.html.AjaxPush`
- renderer-type: `org.ajax4jsf.components.AjaxPushRenderer`

The `<a4j:push>` component periodically performs an Ajax request to the server, simulating "push" functionality. While it is not strictly pushing updates, the request is made to minimal code only, not to the JSF tree, checking for the presence of new messages in the queue. The request registers `EventListener`, which receives messages about events, but does not poll registered beans. If a message exists, a complete request is performed. This is different from the `<a4j:poll>` component, which performs a full request at every interval.

The `interval` attribute specifies the time in milliseconds between checking for messages. The default for this value is 1000 ms (1 second). It is possible to set the interval value to `0`, in which case it is constantly checking for new messages.

The `timeout` attribute defines the response waiting time in milliseconds. If a response isn't received within the timeout period, the connection is aborted and the next request is sent. By default, the timeout is not set. In combination with the `interval` attribute, checks for the queue state can short polls or long connections.

4.11. `<a4j:repeat>`

- component-type: `org.ajax4jsf.Repeat`
- component-family: `javax.faces.Data`
- component-class: `org.ajax4jsf.component.html.HtmlAjaxRepeat`
- renderer-type: `org.ajax4jsf.components.RepeatRenderer`

REVIEW COMMENT - a4j:repeat is not an action, should be moved. The `<a4j:repeat>` component is used to iterate changes through a repeated collection of components. It allows specific rows of items to be updated without sending Ajax requests for the entire collection. The `<a4j:repeat>` component forms the basis for many of the tabular components detailed in *Chapter 9, Tables and grids*.

The contents of the collection are determined using Expression Language (EL). The data model for the contents is specified with the `value` attribute. The `var` attribute names the object to use when iterating through the collection. This object is then referenced in the relevant child components. After an Ajax request, only the rows specified with the `ajaxKeys` attribute are updated rather than the entire collection. *Example 4.6, "<a4j:repeat> example"* shows how to use `<a4j:repeat>` to maintain a simple table.

Example 4.6. `<a4j:repeat>` example

```
<table>
```

```
    <tbody>

  <a4j:repeat value="#{repeatBean.items}" var="item" ajaxKeys="#{updateBean.updatedRow}">
          <tr>
            <td><h:outputText value="#{item.code}" id="item1" /></td>
            <td><h:outputText value="#{item.price}" id="item2" /></td>
          </tr>
       </a4j:repeat>
    </tbody>
</table>
```

Each row of a table contains two cells: one showing the item code, and the other showing the item price. The table is generated by iterating through items in the `repeatBeans.items` data model.

The `<a4j:repeat>` component uses other attributes common to iteration components, such as the `first` attribute for specifying the first item for iteration, and the `rows` attribute for specifying the number of rows of items to display.

4.12. `<a4j:ajax>`

- component-type: `org.ajax4jsf.Ajax`
- component-family: `org.ajax4jsf.Ajax`
- component-class: `org.ajax4jsf.component.html.HtmlAjaxSupport`
- renderer-type: `org.ajax4jsf.components.AjaxSupportRenderer`

REVIEW COMMENT: a4j:support is a4j:ajax in 4.0, paths need to be updated. The `<a4j:ajax>` component allows Ajax capability to be added to any non-Ajax component. It is placed as a direct child to the component that requires Ajax support. The `<a4j:ajax>` component uses the common attributes listed in *Chapter 2, Common Ajax attributes*.

Attaching JavaScript functions

When attaching the `<a4j:ajax>` component to non-Ajax JavaServer Faces command components, such as `<h:commandButton>` and `<h:commandLink>`, it is important to set `disabledDefault="true"`. If this attribute is not set, a non-Ajax request is sent after the Ajax request and the page is refreshed unexpectedly.

Example 4.7. `<a4j:ajax>` example

```
<h:panelGrid columns="2">
```

```
    <h:inputText id="myinput" value="#{userBean.name}">
        <a4j:ajax event="onkeyup" reRender="outtext" />
    </h:inputText>
    <h:outputText id="outtext" value="#{userBean.name}" />
</h:panelGrid>
```

Resources

This chapter covers those components used to handle and manage resources and beans.

5.1. <a4j:loadBundle>

- component-type: `org.ajax4jsf.Bundle`
- component-family: `org.ajax4jsf.Bundle`
- component-class: `org.ajax4jsf.component.html.AjaxLoadBundle`

The `<a4j:loadBundle>` component is used to load resource bundles to aid in localization of an application. The bundles are localized to the locale of the current view, and properties are stored as a map in the current request attributes.

The `<a4j:loadBundle>` component allows bundles to be accessed by Ajax requests working in their own address scopes. This solves the problem of using the JSF `<h:loadBundle>` component with Ajax, where bundle information loaded with the page was unavailable for later Ajax requests.

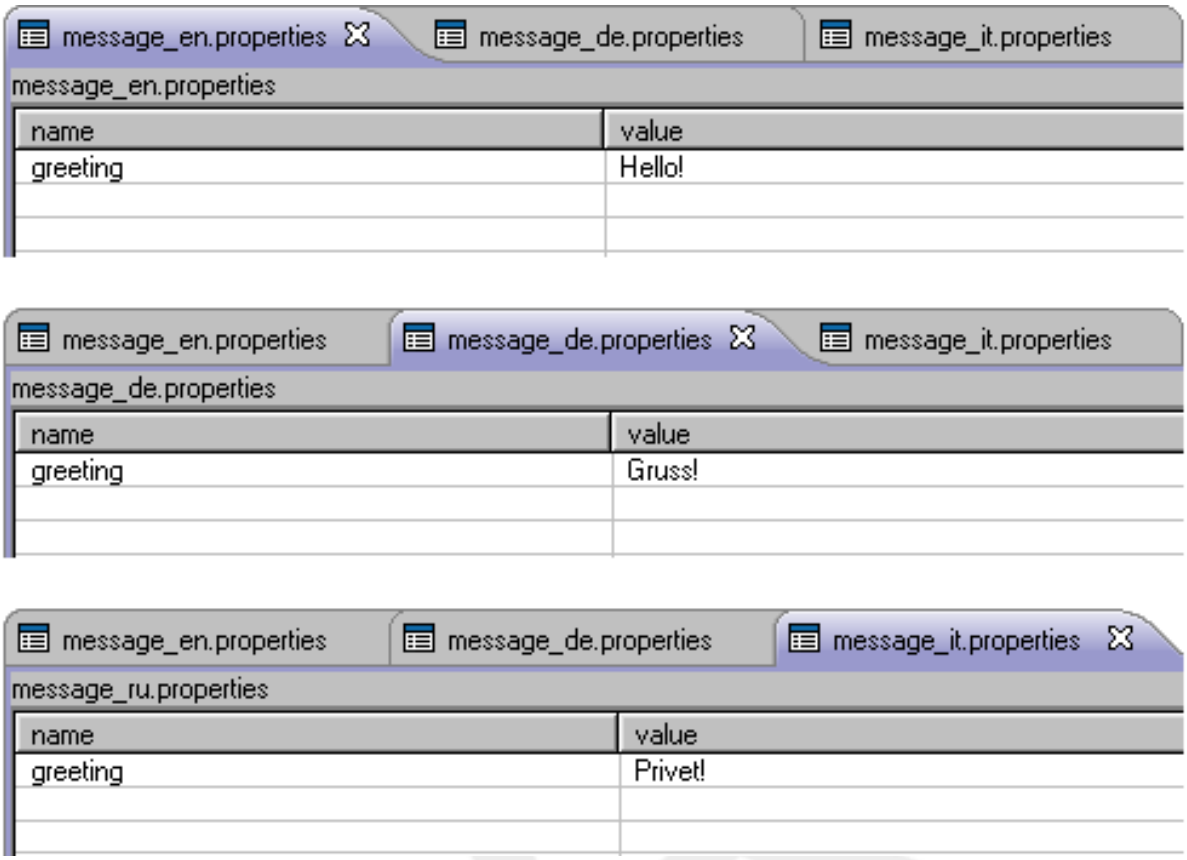
Resource bundles are registered in the Faces configuration file, `faces-config.xml`.

Example 5.1. <a4j:loadBundle> example

This example shows a simple application capable of switching between different localized languages.

1. **Create resource bundles**

   String resource bundles are contained in files with a `.properties` extension. The files consist of a list of entries, each with a unique name and a corresponding value. A seperate file is required for each language. Append the filename with the locale identifier (`en` for English, for example).



2. **Register bundles in Faces configuration file**

   The resource bundles need to be registered in the Faces configuration file, `faces-config.xml`. The filename is defined with the `<message-bundle>` tag, without the locale code and without the extension. The supported locale codes are listed with `<supported-locale>` tags.

```
<application>
   <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>en</supported-locale>
      <supported-locale>de</supported-locale>
      <supported-locale>it</supported-locale>
   </locale-config>
   <message-bundle>demo.message</message-bundle>
</application>
```

3. **Create method to set locale**

   The JSF `javax.faces.component.UIViewRoot.setLocale()` method can be used to update the locale through a Java class:

```
package demo;

import java.util.Locale;
import javax.faces.context.FacesContext;

public class ChangeLocale {

   public String germanAction() {

      FacesContext context = FacesContext.getCurrentInstance();
      context.getViewRoot().setLocale(Locale.GERMAN);
      return null;
   }


   public String englishAction() {

      FacesContext context = FacesContext.getCurrentInstance();
      context.getViewRoot().setLocale(Locale.ENGLISH);
      return null;
   }

   public String italianAction() {

      FacesContext context = FacesContext.getCurrentInstance();
      context.getViewRoot().setLocale(Locale.ITALIAN);
      return null;
   }
}
```

4. **Add `<a4j:loadBundle>` to the JSP page**

   The `<a4j:loadBundle>` component is added to the application, and links to change the locale will update the displayed text.

```
<h:form>
   <a4j:loadBundle var="msg" basename="demo.message"/>
   <h:outputText id="messageBundle" value="#{msg.greeting}"/>
```

```
<a4j:commandLink value="De" action="#{changeLocale.germanAction}" reRender="messageBundle"
    >
<a4j:commandLink value="Eng" action="#{changeLocale.englishAction}" reRender="messageBundle"
    >
<a4j:commandLink value="It" action="#{changeLocale.italianAction}" reRender="messageBundle"
    >
    </h:form>
```

Clicking on the different links will render the localized string as appropriate.

5.2. `<a4j:loadScript>`

- component-type: `org.ajax4jsf.LoadScript`
- component-family: `org.ajax4jsf.LoadScript`
- component-class: `org.ajax4jsf.component.html.HtmlLoadScript`
- renderer-type: `org.ajax4jsf.LoadScriptRenderer`

The `<a4j:loadScript>` component allows scripts to be loaded from external sources such as `jar` files.

The required `src` attribute defines the path to the script. A leading slash (`/`) represents the root of the web context. The `resource://` prefix can be used to access a file in the RichFaces resource framework. The path is passed to the `getResourceURL()` method of the application's `ViewHandler`, with the result then being passed through the `encodeResourceURL()` method of `ExternalContext`.

Example 5.2. `<a4j:loadScript>` example

```
<a4j:loadScript src="resource:///org/mycompany/assets/script/focus.js" />
```

5.3. <a4j:loadStyle>

- component-type: org.ajax4jsf.LoadStyle
- component-family: org.ajax4jsf.LoadStyle
- component-class: org.ajax4jsf.component.html.HtmlLoadStyle
- renderer-type: org.ajax4jsf.LoadStyleRenderer

The <a4j:loadStyle> component allows a style sheet to be loaded from an external source, such as a jar file. The style sheet links are inserted into the head element.

The required src attribute defines the path to the script. A leading slash (/) represents the root of the web context. The resource:// prefix can be used to access a file in the RichFaces resource framework. The path is passed to the getResourceURL() method of the application's ViewHandler, with the result then being passed through the encodeResourceURL() method of ExternalContext.

Example 5.3. <a4j:loadStyle> example

```
<a4j:loadStyle src="resource:///org/mycompany/assets/script/focus.js" />
```

5.4. <a4j:keepAlive>

- component-type: org.ajax4jsf.components.KeepAlive
- component-family: org.ajax4jsf.components.AjaxKeepAlive
- component-class: org.ajax4jsf.components.AjaxKeepAlive

The <a4j:keepAlive> component allows the state of a managed bean to be retained between Ajax requests.

Managed beans can be declared with the request scope in the faces-config.xml configuration file, using the <managed-bean-scope> tag. Any references to the bean instance after the request has ended will cause the server to throw an illegal argument exception. The <a4j:keepAlive> component avoids this be maintaining the state of the whole bean object for subsequent requests.

The beanName attribute uses JSF Expression Language (EL) to define the request scope bean name to keep alive. The expression must resolve to a managed bean instance. The ajaxOnly attribute determines whether or not the value of the bean should be available during non-Ajaxx requests; if ajaxOnly="true", the request-scope bean keeps its value during Ajax requests, but any non-Ajax requests will re-create the bean as a regular request-scope bean.

Example 5.4. `<a4j:keepAlive>` example

```
<a4j:keepAlive beanName="#{myClass.testBean}" />
```

5.5. `<a4j:mediaOutput>`

- component-type: `org.ajax4jsf.Push`
- component-family: `org.ajax4jsf.components.AjaxPush`
- component-class: `org.ajax4jsf.component.html.AjaxPush`
- renderer-type: `org.ajax4jsf.components.AjaxPushRenderer`

The `<a4j:mediaOutput>` component is used for generating images, video, sounds, and other resources defined on the fly.

The `createContent` attribute points to the method used for generating the displayed content. If necessary, the `value` attribute can be used to pass input data to the content generation method specified with `createContent`. The `cacheable` attribute specifies whether the resulting content will be cached or not.

The `mimeType` attribute describes the type of output content, and corresponds to the type in the header of the HTTP request. The `element` attribute defines XHTML element used to display the content:

- `img`
- `object`
- `applet`
- `script`
- `link`
- `a`

Example 5.5. `<a4j:mediaOutput>` example

This example uses the `<a4j:mediaOutput>` component to generate a JPEG image of verification digits. The code on the application page is a single element:

```
<a4j:mediaOutput element="img" cacheable="false" createContent="#{mediaBean.paint}" value="#{mediaBean.number}" mimeType="image/jpeg" />
```

The `<a4j:mediaOutput>` component uses the `MediaBean.paint` method to create the image. The method generates a random number, which is then converted into an output stream and rendered to a JPEG image. The `MediaBean` class is as follows:

```
package demo;

import java.awt.Graphics2D;
import java.awt.image.BufferedImage;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Random;
import javax.imageio.ImageIO;

public class MediaBean {

   public void paint(OutputStream out, Object data) throws IOException {

      Integer high = 9999;
      Integer low = 1000;
      Random generator = new Random();
      Integer digits = generator.nextInt(high - low + 1) + low;

      if (data instanceof MediaData) {
         MediaData paintData = (MediaData) data;
         BufferedImage img = new BufferedImage(paintData.getWidth(),paintData.getHeight(),BufferedImage.TYPE_
         Graphics2D graphics2D = img.createGraphics();
         graphics2D.setBackground(paintData.getBackground());
         graphics2D.setColor(paintData.getDrawColor());
         graphics2D.clearRect(0,0,paintData.getWidth(),paintData.getHeight());
         graphics2D.setFont(paintData.getFont());
         graphics2D.drawString(digits.toString(), 20, 35);
         ImageIO.write(img,"png",out);
      }
   }
}
```

Another class, `MediaData` is required by the `value` attribute for keeping data to be used as input for the content creation method. The `MediaData` class is as follows:

```
package demo;
```

```
import java.awt.Color;
import java.awt.Font;
import java.io.Serializable;

public class MediaData implements Serializable {

   private static final long serialVersionUID = 1L;

   Integer Width=110;
   Integer Height=50;

   Color Background=new Color(190, 214, 248);
   Color DrawColor=new Color(0,0,0);

   Font font = new Font("Serif", Font.TRUETYPE_FONT, 30);

   /* Corresponding getters and setters */
   ...

}
```

The `<a4j:mediaOutput>` component uses the `MediaBean` and `MediaData` classes to generate a new image on each page refresh, which appears as shown in *Figure 5.1, "<a4j:mediaOutput> example result"*

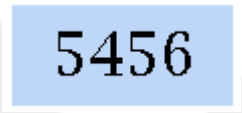


Figure 5.1. `<a4j:mediaOutput>` example result

`Serializable` interface

A bean class passed using the `value` attribute of `<a4j:mediaOutput>` should implement the `Serializable` interface so that it will be encoded to the URL of the resource.

Containers

This chapter details those components in the `a4j` tag library which define an area used as a container or wrapper for other components.

6.1. `<a4j:form>`

- component-type: `org.ajax4jsf.Form`
- component-family: `javax.faces.Form`
- component-class: `org.ajax4jsf.component.html.AjaxForm`
- renderer-type: `org.ajax4jsf.FormRenderer`

The `<a4j:form>` builds on the functionality of the JavaServer Faces (JSF) component `<h:form>`, adding Ajax capabilities to the form.

Command link rendering fixed

The JSF component `<h:form>`, on which the `<a4j:form>` component is based, had an issue whereby the `<h:commandLink>` component could not be re-rendered without re-rendering the entire form. `<a4j:form>` and `<a4j:commandLink>` fix this issue.

The `<a4j:form>` component can add indirect Ajax support to non-Ajax components on the form by setting `ajaxSubmit="true"`. It then uses the standard Ajax component attributes and updates components specified with the `render` attribute.

`ajaxSubmit`

`<a4j:form>` should not use `ajaxSubmit="true"` if it contains other Ajax-capable components.

Additionally, due to security reasons the file upload form element cannot be indirectly made Ajax-capable with `<a4j:form>`.

6.2. `<a4j:include>`

- component-type: `org.ajax4jsf.Include`
- component-family: `javax.faces.Output`
- component-class: `org.ajax4jsf.component.html.Include`
- renderer-type: `org.ajax4jsf.components.AjaxIncludeRenderer`

The `<a4j:include>` component allows one view to be included as part of another page. This is useful for applications where multiple views might appear on the one page, with navigation between the views. Views can use partial page navigation in Ajax mode, or standard JSF navigation for navigation between views.

The `viewId` attribute is required to reference the resource that will be included as a view on the page. It uses a full context-relative path to point to the resource, similar to the paths used for the `<from-view-id>` and `<to-view-id>` tags in the `faces-config.xml` JSF navigation rules.

Example 6.1. A wizard using `<a4j:include>`

The page uses `<a4j:include>` to include the first step of the wizard:

```
<h:panelGrid width="100%" columns="2">
   <a4j:keepAlive beanName="profile" />
   <rich:panel>
      <f:facet name="header">
      <h:outputText value="A wizard using a4j:include" />
      </f:facet>
      <h:form>
         <a4j:include viewId="/richfaces/include/examples/wstep1.xhtml" />
      </h:form>
   </rich:panel>
</h:panelGrid>
```

The first step is fully contained in a separate file, `wstep1.xhtml`. Subsequent steps are set up similarly with additional **Previous** buttons.

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:a4j="http://richfaces.org/a4j"
      xmlns:rich="http://richfaces.org/rich">

   <div style="position:relative;height:140px">
      <h:panelGrid rowClasses="s1row" columns="3" columnClasses="wfcol1,wfcol2,wfcol3">
         <h:outputText value="First Name:" />
         <h:inputText id="fn" value="#{profile.firstName}" label="First Name" required="true" />
         <rich:message  for="fn" />
```

```
        <h:outputText value="Last Name:" />
        <h:inputText  id="ln" value="#{profile.lastName}"  label="Last Name"  required="true" />
        <rich:message  for="ln" />
      </h:panelGrid>
      <div class="navPanel" style="width:100%;">
        <a4j:commandButton style="float:right" action="next" value="Next &gt;&gt;"/>
      </div>
    </div>
</ui:composition>
```

The navigation is defined in the `faces-config.xml` configuration file:

```
<navigation-rule>
  <from-view-id>/richfaces/include/examples/wstep1.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>next</from-outcome>
    <to-view-id>/richfaces/include/examples/wstep2.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
<navigation-rule>
  <from-view-id>/richfaces/include/examples/wstep2.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>previous</from-outcome>
    <to-view-id>/richfaces/include/examples/wstep1.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>next</from-outcome>
    <to-view-id>/richfaces/include/examples/finalStep.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
<navigation-rule>
  <from-view-id>/richfaces/include/examples/finalStep.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>previous</from-outcome>
    <to-view-id>/richfaces/include/examples/wstep2.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
```

6.3. `<a4j:outputPanel>`

- component-type: `org.ajax4jsf.OutputPanel`

- component-family: `javax.faces.Panel`
- component-class: `org.ajax4jsf.component.html.HtmlAjaxOutputPanel`
- renderer-type: `org.ajax4jsf.components.AjaxOutputPanelRenderer`

The `<a4j:outputPanel>` component is used to group together components in to update them as a whole, rather than having to specify the components individually.

The `layout` attribute can be used to determine how the component is rendered in HTML:

- `layout="inline"` is the default behavior, which will render the component as a pair of `<span>` tags containing the child components.
- `layout="block"` will render the component as a pair of `<div>` tags containing the child components, which will use any defined `<div>` element styles.
- `layout="none"` will render the component as a pair of `<span>` tags with an identifier equal to that of a child component. If the child component is rendered then the `<span>` are not included, leaving no markup representing the `<a4j:outputPanel>` in HTML.

Setting `ajaxRendered="true"` will cause the `<a4j:outputPanel>` to be updated with each Ajax response for the page, even when not listed explicitly by the requesting component. This can in turn be overridden by specific attributes on any requesting components.

6.4. `<a4j:page>`

- component-type: `org.ajax4jsf.components.Page`
- component-family: `org.ajax4jsf.components.AjaxRegion`
- component-class: `org.ajax4jsf.component.html.HtmlPage`
- renderer-type: `org.ajax4jsf.components.AjaxPageRenderer`

The `<a4j:page>` component encodes a full HTML-page structure using only one tag. It renders complete `<!DOCTYPE>`, `<html>`, `<head>`, `<title>`, and `<body>` tags using the specified attributes and facets. Additionally, the `<a4j:page>` component solves an incompatibility issue between early versions of MyFaces and the Ajax4jsf framework.

The `pageTitle` attribute is rendered as a `<title>` element. The component uses the `head` facet to define the contents of the HTML `<head>` element. The `format` facet defines the page layout format for encoding the `<!DOCTYPE>` element. There rest of the contents of the `<a4j:page>` component are rendered as part of the `<body>` element.

Example 6.2. `<a4j:page>` rendering

An `<a4j:page>` component can be defined as follows:

```
<a4j:page format="xhtml" pageTitle="myPage">
```

```
    <f:facet name="head">
        <!--Head Content here-->
    </f:facet>
    <!--Page Content Here-->
</a4j:page>
```

This definition will render on an XHTML page as follows:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
  <head>
    <title>myPage</title>
    <!--Head Content here-->
  </head>
  <body>
    <!--Page Content Here-->
  </body>
</html>
```

When using the Ajax4jsf framework with MyFaces version 1.2.2 and lower, the `<f:view>` JSP tag does not receive control for encoding contents during the `RENDER_RESPONSE` phase. As a result, Ajax fails to receive control and send responses. The `<a4j:page>` component solves this problem by wrapping the Ajax areas to be updated. Later versions of MyFaces do not have this problem, and as such do not require the use of the `<a4j:page>` component.

6.5. `<a4j:region>`

- component-type: `org.ajax4jsf.AjaxRegion`
- component-family: `org.ajax4jsf.AjaxRegion`
- component-class: `org.ajax4jsf.component.html.HtmlAjaxRegion`
- renderer-type: `org.ajax4jsf.components.AjaxRegionRenderer`

The `<a4j:region>` component specifies a part of the document object model (DOM) tree to be processed on the server. The processing includes data handling during decoding, conversion, validation, and model updating. When not using `<a4j:region>`, the entire view functions as a region.

The whole form is still submitted to the server, but only the specified region is processed. Regions can be nested, in which case only the immediate region of the component initiating the request will be processed.

Validation

This chapter covers those components that validate user input. The components enhance JSF validation capabilities with Ajax support and the use of **Hibernate** validators.

7.1. `<rich:ajaxValidator>`

- component-type: `org.richfaces.ajaxValidator`
- component-class: `org.richfaces.component.html.HtmlajaxValidator`
- component-family: `org.richfaces.ajaxValidator`
- renderer-type: `org.richfaces.ajaxValidatorRenderer`
- tag-class: `org.richfaces.taglib.ajaxValidatorTag`

The `<rich:ajaxValidator>` component provides Ajax validation for JSF inputs. It is added as a child component to a JSF tag, and the `event` attribute specifies when to trigger the validation.

Example 7.1. `<rich:ajaxValidator>` example

This example shows the use of `<rich:ajaxValidator>` with standard JSF validators. The validators check the length of the entered name, and the range of the entered age.

```
<rich:panel>
   <f:facet name="header">
      <h:outputText value="User Info:" />
   </f:facet>
   <h:panelGrid columns="3">

      <h:outputText value="Name:" />
      <h:inputText value="#{userBean.name}" id="name" required="true">
         <f:validateLength minimum="3" maximum="12"/>
         <rich:ajaxValidator event="onblur"/>
      </h:inputText>
      <rich:message for="name" />

      <h:outputText value="Age:" />
      <h:inputText value="#{userBean.age}" id="age" required="true">
         <f:convertNumber integerOnly="true"/>
         <f:validateLongRange minimum="18" maximum="99"/>
```

```
            <rich:ajaxValidator event="onblur"/>
        </h:inputText>
        <rich:message for="age"/>

    </h:panelGrid>
</rich:panel>
```

The `<rich:ajaxValidator>` component can also work with custom validators made using the JSF Validation API in the javax.faces.validator package, or with Hibernate Validator. Refer to the *Hibernate Validator documentation* for details on how to use Hibernate Validator.

Example 7.2. Using `<rich:ajaxValidator>` with Hibernate Validator

This example shows the use of `<rich:ajaxValidator>` with Hibernate Validator. It validates the entered name, email, and age.

```
<h:form id="ajaxValidatorForm2">
    <rich:panel>
        <f:facet name="header">
            <h:outputText value="User Info:" />
        </f:facet>
        <h:panelGrid  columns="3">
            <h:outputText value="Name:" />
            <h:inputText value="#{validationBean.name}" id="name" required="true">
                <rich:ajaxValidator event="onblur" />
            </h:inputText>
            <rich:message for="name" />
            <h:outputText value="Email:" />
            <h:inputText value="#{validationBean.email}" id="email">
                <rich:ajaxValidator event="onblur" />
            </h:inputText>
            <rich:message for="email" />
            <h:outputText value="Age:" />
            <h:inputText value="#{validationBean.age}" id="age">
                <rich:ajaxValidator event="onblur" />
            </h:inputText>
            <rich:message for="age" />
        </h:panelGrid>
    </rich:panel>
</h:form>
```

The validation is performed using the `ValidationBean` class:

```
package org.richfaces.demo.validation;

import org.hibernate.validator.Email;
import org.hibernate.validator.Length;
import org.hibernate.validator.Max;
import org.hibernate.validator.Min;
import org.hibernate.validator.NotEmpty;
import org.hibernate.validator.NotNull;
import org.hibernate.validator.Pattern;

public class ValidationBean {

    private String progressString="Fill the form in";

    @NotEmpty
    @Pattern(regex=".*[^\\s].*", message="This string contains only spaces")
    @Length(min=3,max=12)
    private String name;
    @Email
    @NotEmpty
    private String email;

    @NotNull
    @Min(18)
    @Max(100)
    private Integer age;

    public ValidationBean() {
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }
```

```
    public void setEmail(String email) {
        this.email = email;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
    public void success() {
        setProgressString(getProgressString() + "(Stored successfully)");
    }

    public String getProgressString() {
        return progressString;
    }

    public void setProgressString(String progressString) {
        this.progressString = progressString;
    }
}
```

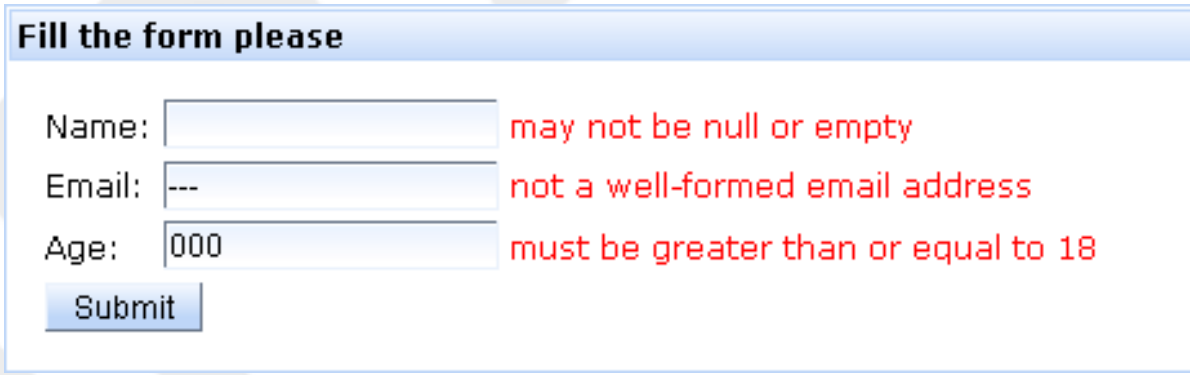


Figure 7.1. `<rich:ajaxValidator>` example result

7.2. `<rich:beanValidator>`

- component-type: `org.richfaces.beanValidator`
- component-class: `org.richfaces.component.html.HtmlbeanValidator`
- component-family: `org.richfaces.beanValidator`
- renderer-type: `org.richfaces.beanValidatorRenderer`

- tag-class: `org.richfaces.taglib.beanValidatorTag`

The `<rich:beanValidator>` component provides model-based constraints using Hibernate Validator. This allows Hibernate Validator to be used similar to its use with Seam-based applications.

The `summary` attribute is used for displaying messages about validation errors.

Example 7.3. `<rich:beanValidator>` example

This example shows the bean-based validation of a simple form, containing the user's name, email, and age. The `<rich:beanValidator>` component is defined in the same way as for JSF validators.

```
<h:form id="beanValidatorForm">
   <rich:panel>
      <f:facet name="header">
         <h:outputText value="#{validationBean.progressString}" id="progress"/>
      </f:facet>
      <h:panelGrid columns="3">
         <h:outputText value="Name:" />
         <h:inputText value="#{validationBean.name}" id="name">
            <rich:beanValidator summary="Invalid name"/>
         </h:inputText>
         <rich:message for="name" />
         <h:outputText value="Email:" />
         <h:inputText value="#{validationBean.email}" id="email">
            <rich:beanValidator summary="Invalid email"/>
         </h:inputText>
         <rich:message for="email" />
         <h:outputText value="Age:" />
         <h:inputText value="#{validationBean.age}" id="age">
            <rich:beanValidator summary="Wrong age"/>
         </h:inputText>
         <rich:message for="age" />
         <f:facet name="footer">

<a4j:commandButtonvalue="Submit"action="#{validationBean.success}"reRender="progress"/
>
         </f:facet>
      </h:panelGrid>
   </rich:panel>
</h:form>
```

The accompanying bean contains the validation data:

```
package org.richfaces.demo.validation;

import org.hibernate.validator.Email;
import org.hibernate.validator.Length;
import org.hibernate.validator.Max;
import org.hibernate.validator.Min;
import org.hibernate.validator.NotEmpty;
import org.hibernate.validator.NotNull;
import org.hibernate.validator.Pattern;

public class ValidationBean {

   private String progressString="Fill the form in";

   @NotEmpty
   @Pattern(regex=".*[^\\s].*", message="This string contains only spaces")
   @Length(min=3,max=12)
   private String name;
   @Email
   @NotEmpty
   private String email;

   @NotNull
   @Min(18)
   @Max(100)
   private Integer age;

   public ValidationBean() {
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public String getEmail() {
      return email;
   }
```

```
    public void setEmail(String email) {
        this.email = email;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
    public void success() {
        setProgressString(getProgressString() + "(Stored successfully)");
    }

    public String getProgressString() {
        return progressString;
    }

    public void setProgressString(String progressString) {
        this.progressString = progressString;
    }
}
```

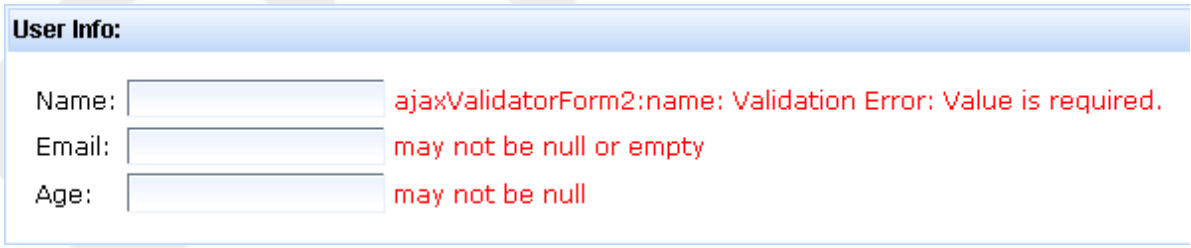


Figure 7.2. `<rich:beanValidator>` example result

7.3. `<rich:graphValidator>`

- component-type: `org.richfaces.graphValidator`
- component-class: `org.richfaces.component.html.HtmlgraphValidator`
- component-family: `org.richfaces.graphValidator`
- renderer-type: `org.richfaces.graphValidatorRenderer`

- tag-class: `org.richfaces.taglib.graphValidatorTag`

The `<rich:graphValidator>` component is used to wrap a group of input components for overall validation with Hibernate Validators. This is different from the `<rich:beanValidator>` component, which is used as a child element to individual input components.

The `summary` attribute is used for displaying messages about validation errors.

Example 7.4. `<rich:graphValidator>` example

This example shows the validation of a simple form, containing the user's name, email, and age. The `<rich:graphValidator>` component wraps the input components to validate them together.

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:a4j="http://richfaces.org/a4j"
    xmlns:rich="http://richfaces.org/rich">

    <h:form id="graphValidatorForm">
        <a4j:region renderRegionOnly="true">
            <rich:panel id="panel">
                <f:facet name="header">
                    <h:outputText value="User Info:" />
                </f:facet>
                <rich:graphValidator summary="Invalid values: ">
                    <h:panelGrid columns="3">
                        <h:outputText value="Name:" />
                        <h:inputText value="#{validationBean.name}" id="name">
                            <f:validateLength minimum="2" />
                        </h:inputText>
                        <rich:message for="name" />
                        <h:outputText value="Email:" />
                        <h:inputText value="#{validationBean.email}" id="email" />
                        <rich:message for="email" />
                        <h:outputText value="Age:" />
                        <h:inputText value="#{validationBean.age}" id="age" />
                        <rich:message for="age" />
                    </h:panelGrid>
                </rich:graphValidator>
                <a4j:commandButton value="Store changes" />
            </rich:panel>
        </a4j:region>
```

```
    </h:form>
</ui:composition>
```

The accompanying bean contains the validation data:

```
package org.richfaces.demo.validation;

import org.hibernate.validator.Email;
import org.hibernate.validator.Length;
import org.hibernate.validator.Max;
import org.hibernate.validator.Min;
import org.hibernate.validator.NotEmpty;
import org.hibernate.validator.NotNull;
import org.hibernate.validator.Pattern;

public class ValidationBean {

    private String progressString="Fill the form in";

    @NotEmpty
    @Pattern(regex=".*[^\\s].*", message="This string contains only spaces")
    @Length(min=3,max=12)
    private String name;
    @Email
    @NotEmpty
    private String email;

    @NotNull
    @Min(18)
    @Max(100)
    private Integer age;

    public ValidationBean() {
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
    public void success() {
        setProgressString(getProgressString() + "(Stored successfully)");
    }

    public String getProgressString() {
        return progressString;
    }

    public void setProgressString(String progressString) {
        this.progressString = progressString;
    }
}
```

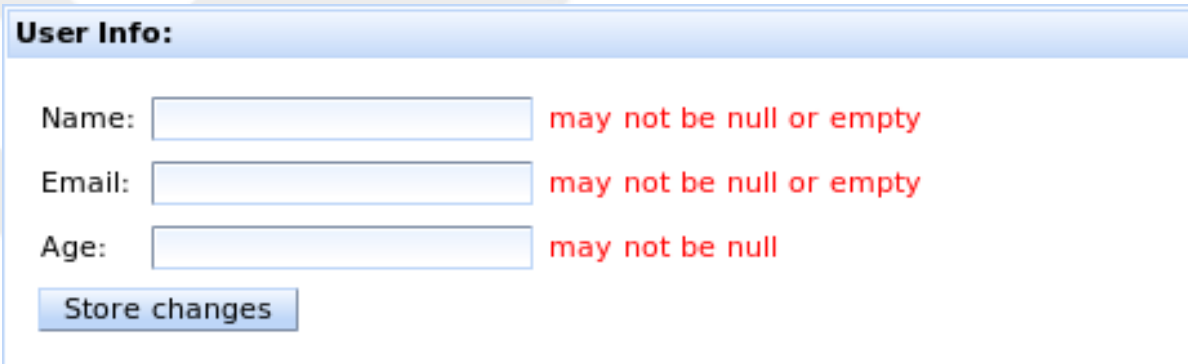


Figure 7.3. `<rich:graphValidator>` example result

The optional `value` attribute can be used to define a value bound to the bean. The bean properties are then validated again *after* the model has been updated.

Example 7.5. Using the `value` attribute

```
<h:form id="graphValidatorForm2">
   <a4j:region renderRegionOnly="true">
      <rich:graphValidator summary="Invalid values: " value="#{dayStatistics}">
         <table>
            <thead>
               <tr>
                  <th>Activity</th>
                  <th>Time</th>
               </tr>
            </thead>
            <tbody>
               <a4j:repeat value="#{dayStatistics.dayPasstimes}" var="pt"
                  id="table">
                  <tr>
                     <td align="center" width="100px"><h:outputText
                        value="#{pt.title}" /></td>
                     <td align="center" width="100px"><rich:inputNumberSpinner
                        minValue="0" maxValue="24" value="#{pt.time}" id="time">
                     </rich:inputNumberSpinner></td>
                     <td><rich:message for="time" /></td>
                  </tr>
               </a4j:repeat>
            </tbody>
         </table>
      </rich:graphValidator>
      <a4j:commandButton value="Store my details"
         actionListener="#{dayStatistics.store}" reRender="panel" />
      <rich:messages infoClass="green" errorClass="red" />
   </a4j:region>
</h:form>
```

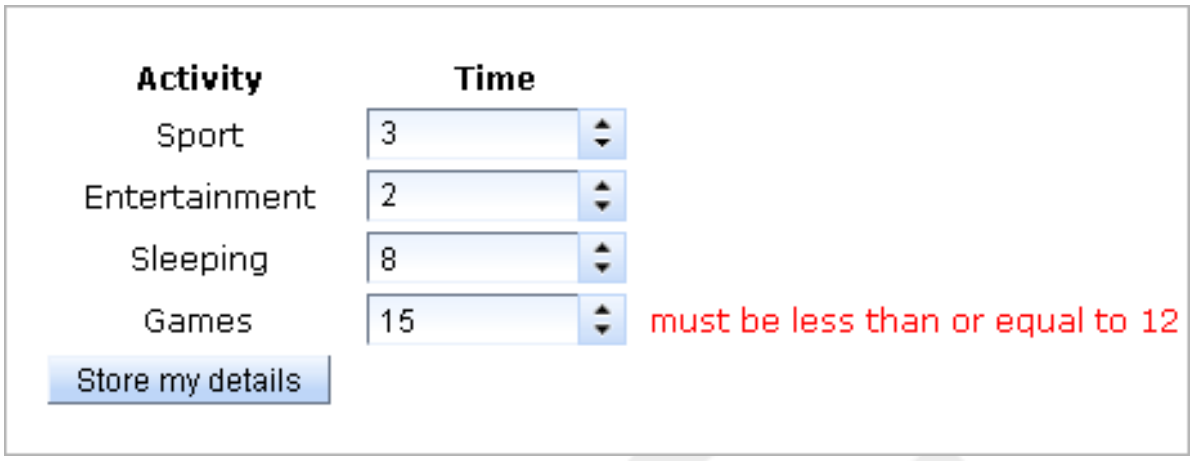


Figure 7.4. Result from using the `value` attribute

Processing management

This chapter covers those components that manage the processing of information, requests, and updates.

8.1. `<a4j:queue>`

- component-family: `org.ajax4jsf.Queue`
- component-class: `org.ajax4jsf.component.html.HtmlQueue`
- renderer-type: `org.ajax4jsf.QueueRenderer`
- tag-class: `org.ajax4jsf.taglib.html.jsp.QueueTag`

The `<a4j:queue>` component manages a queue of Ajax requests to control message processing.

The queue can be disabled by setting the `disabled` attribute to `true`. The `size` attribute specifies the number of requests that can be stored in the queue at a time; smaller queue sizes help prevent server overloads. When the queue's size is exceeded, the `sizeExceededBehavior` determines the way in which the queue handles the requests:

- `dropNext` drops the next request currently in the queue.
- `dropNew` drops the incoming request.
- `fireNext` immediately sends the next request currently in the queue.
- `fireNew` immediately sends the incoming request.

The `<a4j:queue>` component features several events relating to queuing actions:

- The `oncomplete` event attribute is fired after a request is completed. The request object is passed as a parameter to the event handler, so the queue is accessible using `request.queue` and the element which was the source of the request is accessible using `this`.
- The `onrequestqueue` event attribute is fired after a new request has been added to the queue.
- The `onrequestdequeue` event attribute is fired after a request has been removed from the queue.
- The `onsizeexceeded` event attribute is fired when the queue has been exceeded.
- The `onsubmit` event attribute is fired before the request is sent.
- The `onsuccess` event attribute is fired after a successful request bu before the DOM is updated on the client side.

8.2. <a4j:log>

- component-type: `org.ajax4jsf.Log`
- component-family: `org.ajax4jsf.Log`
- component-class: `org.ajax4jsf.component.html.AjaxLog`
- renderer-type: `org.ajax4jsf.LogRenderer`

The `<a4j:log>` component generates JavaScript that opens a debug window, logging application information such as requests, responses, and DOM changes.

The `popup` attribute causes the logging data to appear in a new pop-up window if set to `true`, or in place on the current page if set to `false`. The window is set to be opened by pressing the key combination **Ctrl**+**Shift**+**L**; this can be partially reconfigured with the `hotkey` attribute, which specifies the letter key to use in combination with **Ctrl**+**Shift** instead of **L**.

The amount of data logged can be determined with the `level` attribute:

- `ERROR`
- `FATAL`
- `INFO`
- `WARN`
- `ALL`, the default setting, logs all data.

Example 8.1. <a4j:log> example

```
<a4j:log level="ALL" popup="false" width="400" height="200" />
```

Log renewal

The log is automatically renewed after each Ajax request. It does not need to be explicitly re-rendered.

8.3. <a4j:status>

- component-type: `org.ajax4jsf.Status`

- component-family: `javax.faces.Panel`
- component-class: `org.ajax4jsf.component.html.HtmlAjaxStatus`
- renderer-type: `org.ajax4jsf.components.AjaxStatusRenderer`

The `<a4j:status>` component displays the status of current Ajax requests; the status can be either in progress or complete.

The `startText` attribute defines the text shown after the request has been started and is currently in progress. This text can be styled with the `startStyle` and `startStyleClass` attributes. Similarly, the `stopText` attribute defines the text shown once the request is complete, and text is styled with the `stopStyle` and `stopStyleClass` attributes. Alternatively, the text styles can be customized using facets, with the facet name set to either `start` or `stop` as required. If the `stopText` attribute is not defined, and no facet exists for the stopped state, the status is simply not shown; in this way only the progress of the request is displayed to the user.

Example 8.2. Basic `<a4j:status>` usage

```
<a4j:status startText="In progress..." stopText="Complete" />
```

The `<a4j:status>` component works for each Ajax component inside the local region. If no region is defined, every request made on the page will activate the `<a4j:status>` component. Alternatively, the `<a4j:status>` component can be linked to specific components in one of two ways:

- The `for` attribute can be used to specify the component for which the status is to be monitored.
- With an `id` identifier attribute specified for the `<a4j:status>`, individual components can have their statuses monitored by referencing the identifier with their own `status` attributes.

Example 8.3. Updating a common `<a4j:status>` component

```
<a4j:region id="extr">
   <h:form>
      <h:outputText value="Status:" />
      <a4j:status id="commonstatus" startText="In Progress...." stopText="" />

      <a4j:region id="intr">
         <h:panelGrid columns="2">
            <h:outputText value="Name" />
            <h:inputText id="name" value="#{userBean.name}">
```

```
            <a4j:support event="onkeyup" reRender="out" status="commonstatus" />
          </h:inputText>

          <h:outputText value="Job" />
          <h:inputText id="job" value="#{userBean.job}">
            <a4j:support event="onkeyup" reRender="out" status="commonstatus" />
          </h:inputText>

          <h:panelGroup />

        </h:panelGrid>
      </a4j:region>
      <a4j:region>
        <br />
        <rich:spacer height="5" />
        <b><h:outputText id="out"
          value="Name: #{userBean.name}, Job: #{userBean.job}" /></b>
        <br />
        <rich:spacer height="5" />
        <br />
        <a4j:commandButton ajaxSingle="true" value="Clean Up Form"
          reRender="name, job, out" status="commonstatus">
          <a4j:actionparam name="n" value="" assignTo="#{userBean.name}" />
          <a4j:actionparam name="j" value="" assignTo="#{userBean.job}" />
        </a4j:commandButton>
      </a4j:region>

    </h:form>
</a4j:region>
```

Part II. rich tag library

Tables and grids

Documentation in development

Some concepts covered in this chapter may refer to the previous version of Richfaces , version 3.3.3. This chapter is scheduled for review to ensure all information is up to date.

This chapter covers all components related to the display of tables and grids.

9.1. `<rich:column>`

- component-type: `org.richfaces.Column`
- component-class: `org.richfaces.component.html.HtmlColumn`
- component-family: `org.richfaces.Column`
- renderer-type: `org.richfaces.renderkit.CellRenderer`
- tag-class: `org.richfaces.taglib.ColumnTag`

The `<rich:column>` component facilitates columns in a table or other `UIData` component. It supports merging columns and rows, sorting, filtering, and customized skinning.

In general usage, the `<rich:column>` component is used in the same was as the JavaServer Faces (JSF) `<h:column>` component. It requires no extra attributes for basic usage, as shown in *Example 9.1, "Basic column example"*.

Example 9.1. Basic column example

```
<rich:dataTable value="#{capitalsBean.capitals}" var="cap" rows="5">
  <rich:column>
    <f:facet name="header">State Flag</f:facet>
    <h:graphicImage value="#{cap.stateFlag}"/>
  </rich:column>
  <rich:column>
    <f:facet name="header">State Name</f:facet>
    <h:outputText value="#{cap.state}"/>
  </rich:column>
  <rich:column >
    <f:facet name="header">State Capital</f:facet>
    <h:outputText value="#{cap.name}"/>
```

```
  </rich:column>
  <rich:column>
    <f:facet name="header">Time Zone</f:facet>
    <h:outputText value="#{cap.timeZone}"/>
  </rich:column>
</rich:dataTable>
```

State Flag	State Name	State Capital	Time Zone
	Alabama	Montgomery	GMT-6
	Alaska	Juneau	GMT-9
	Arizona	Phoenix	GMT-7
	Arkansas	Little Rock	GMT-6
	California	Sacramento	GMT-8

Figure 9.1. Basic column example

Columns can be merged by using the `colspan` attribute to specify how many normal columns to span. The `colspan` attribute is used in conjunction with the `breakBefore` attribute on the next column to determine how the merged columns are laid out. *Example 9.2, "Column spanning example"*.

Example 9.2. Column spanning example

```
<rich:dataTable value="#{capitalsBean.capitals}" var="cap" rows="5">
  <rich:column colspan="3">
    <h:graphicImage value="#{cap.stateFlag}"/>
  </rich:column>
  <rich:column breakBefore="true">
    <h:outputText value="#{cap.state}"/>
  </rich:column>
  <rich:column >
    <h:outputText value="#{cap.name}"/>
  </rich:column>
  <rich:column>
    <h:outputText value="#{cap.timeZone}"/>
  </rich:column>
</rich:dataTable>
```

Alabama	Montgomery	GMT-6
Alaska	Juneau	GMT-9
Arizona	Phoenix	GMT-7
Arkansas	Little Rock	GMT-6
California	Sacramento	GMT-8

Figure 9.2. Column spanning example

Similarly, the `rowspan` attribute can be used to merge and span rows. Again the `breakBefore` attribute needs to be used on related `<rich:column>` components to define the layout. *Example 9.3, "Row spanning example"* and the resulting *Figure 9.4, "Complex headers using column groups"* show the first column of the table spanning three rows.

Example 9.3. Row spanning example

```
<rich:dataTable value="#{capitalsBean.capitals}" var="cap" rows="5">
  <rich:column rowspan="3">
    <f:facet name="header">State Flag</f:facet>
    <h:graphicImage value="#{cap.stateFlag}"/>
  </rich:column>
  <rich:column>
    <f:facet name="header">State Info</f:facet>
    <h:outputText value="#{cap.state}"/>
  </rich:column>
  <rich:column breakBefore="true">
    <h:outputText value="#{cap.name}"/>
  </rich:column>
  <rich:column breakBefore="true">
    <h:outputText value="#{cap.timeZone}"/>
  </rich:column>
</rich:dataTable>
```

State Flag	State Info
	Alabama
	Montgomery
	GMT-6
	Alaska
	Juneau
	GMT-9
	Arizona
	Phoenix
	GMT-7
	Arkansas
	Little Rock
	GMT-6
	California
	Sacramento
	GMT-8

Figure 9.3. Row spanning example

For details on filtering and sorting columns, refer to *Section 9.11, "Table filtering"* and *Section 9.12, "Table sorting"*.

9.2. `<rich:columnGroup>`

- component-type: `org.richfaces.ColumnGroup`
- component-class: `org.richfaces.component.html.HtmlColumnGroup`
- component-family: `org.richfaces.ColumnGroup`
- renderer-type: `org.richfaces.ColumnGroupRenderer`
- tag-class: `org.richfaces.taglib.ColumnGroupTag`

The `<rich:columnGroup>` component combines multiple columns in a single row to organize complex parts of a table. The resulting effect is similar to using the `breakBefore` attribute of the `<rich:column>` component, but is clearer and easier to follow in the source code.

The `<rich:columnGroup>` can also be used to create complex headers in a table. *Example 9.4, "Complex headers using column groups"* and the resulting *Figure 9.4, "Complex headers using column groups"* demonstrate how complex headers can be achieved.

Example 9.4. Complex headers using column groups

```
<rich:dataTable value="#{capitalsBean.capitals}" var="cap" rows="5" id="sublist">
  <f:facet name="header">
    <rich:columnGroup>
      <rich:column rowspan="2">
  <h:outputText value="State Flag"/>
      </rich:column>
      <rich:column colspan="3">
  <h:outputText value="State Info"/>
      </rich:column>
      <rich:column breakBefore="true">
  <h:outputText value="State Name"/>
      </rich:column>
      <rich:column>
  <h:outputText value="State Capital"/>
      </rich:column>
      <rich:column>
  <h:outputText value="Time Zone"/>
      </rich:column>
    </rich:columnGroup>
  </f:facet>
  <rich:column>
    <h:graphicImage value="#{cap.stateFlag}"/>
      </rich:column>
      <rich:column>
    <h:outputText value="#{cap.state}"/>
      </rich:column>
      <rich:column>
    <h:outputText value="#{cap.name}"/>
      </rich:column>
      <rich:column>
    <h:outputText value="#{cap.timeZone}"/>
  </rich:column>
</rich:dataTable>
```

State Flag	State Info		
	State Name	State Capital	Time Zone
	Alabama	Montgomery	GMT-6
	Alaska	Juneau	GMT-9
	Arizona	Phoenix	GMT-7
	Arkansas	Little Rock	GMT-6
	California	Sacramento	GMT-8

Figure 9.4. Complex headers using column groups

9.3. `<rich:columns>`

- component-type: `org.richfaces.Column`
- tag-class: `org.richfaces.taglib.ColumnsTagHandler`

The `<rich:columns>` component allows for dynamic sets of columns for tables. Columns and rows can be merged, and the look and feel can be highly customized. The component gets a list from a data model and creates a corresponding set of columns in a `<rich:dataTable>` component. The `<rich:columns>` component also supports `header` and `footer` facets.

Basic usage of the `<rich:columns>` component requires the `value` attribute, which points to the data model; the `var` attribute, which holds the current variable for the collection of data; and the `index` attribute, which holds the current counter. The `id` attribute is used for when the individuals rows require identifiers for Ajax events.

Example 9.5. Basic columns example

```
<rich:dataTable value="#{dataTableScrollerBean.model}" var="model" width="750">
  <rich:columns value="#{dataTableScrollerBean.columns}" var="columns" index="ind" id="column#{ind}">
    <f:facet name="header">
      <h:outputText value="#{columns.header}" />
    </f:facet>
    <h:outputText value="#{model[ind].model} " />
    <h:outputText value="#{model[ind].mileage} miles " />
    <h:outputText value="#{model[ind].price}$" />
  </rich:columns>
</rich:dataTable>
```

The output can be customized to display specific columns and rows. The `columns` attribute specifies the number of columns. The `rowspan` attribute specifies the number of rows to display; if the attribute is set to `0`, all remaining rows in the table are displayed. The `begin` and `end` attributes are used to specify the first and last zero-based iteration items to display in the table. Columns can be adjusted using the `colspan`, `rowspan`, and `breakBefore` attributes the same as with the `<rich:column>` component.

The `<rich:columns>` component can be used alongside `<rich:column>` components.

Example 9.6. Using `<rich:columns>` and `<rich:column>` together

```
<rich:dataTable value="#{dataTableScrollerBean.model}" var="model" width="500px" rows="5">
  <f:facet name="header">
    <h:outputText value="Cars Available"></h:outputText>
  </f:facet>
  <rich:columns value="#{dataTableScrollerBean.columns}" var="columns" index="ind">
    <f:facet name="header">
      <h:outputText value="#{columns.header}" />
    </f:facet>
    <h:outputText value="#{model[ind].model} " />
  </rich:columns>
  <rich:column>
    <f:facet name="header">
      <h:outputText value="Price" />
    </f:facet>
    <h:outputText value="Price" />
  </rich:column>
  <rich:columns value="#{dataTableScrollerBean.columns}" var="columns" index="ind">
    <f:facet name="header">
      <h:outputText value="#{columns.header}" />
    </f:facet>
    <h:outputText value="#{model[ind].mileage}$" />
  </rich:columns>
</rich:dataTable>
```

For details on filtering and sorting columns, refer to *Section 9.11, "Table filtering"* and *Section 9.12, "Table sorting"*.

9.4. `<rich:dataDefinitionList>`

- component-type: `org.richfaces.DataDefinitionList`
- component-class: `org.richfaces.component.html.HtmlDataDefinitionList`

- component-family: `org.richfaces.DataDefinitionList`
- renderer-type: `org.richfaces.DataDefinitionListRenderer`
- tag-class: `org.richfaces.taglib.DataDefinitionListTag`

The `<rich:dataDefinitionList>` component renders a list of items with definitions. The component uses a data model for managing the list items, which can be updated dynamically.

The `var` attribute names a variable for iterating through the items in the data model. The items to iterate through are determined with the `value` attribute by using EL (Expression Lanugage). The `first` attribute specifies which item in the data model to start from, and the `rows` attribute specifies the number of items to list. The `title` attribute is used for a floating tool-tip. *Example 9.7, "<rich:dataDefinitionList> example"* shows a simple example using the `<rich:dataDefinitionList>` component.

Example 9.7. `<rich:dataDefinitionList>` example

```
<h:form>

 <rich:dataDefinitionList var="car" value="#{dataTableScrollerBean.allCars}" rows="5" first="4" title="Cars">
     <f:facet name="term">
        <h:outputText value="#{car.make} #{car.model}"></h:outputText>
     </f:facet>
     <h:outputText value="Price:" styleClass="label"></h:outputText>
     <h:outputText value="#{car.price}" /><br/>
     <h:outputText value="Mileage:" styleClass="label"></h:outputText>
     <h:outputText value="#{car.mileage}" /><br/>
  </rich:dataDefinitionList>
</h:form>
```

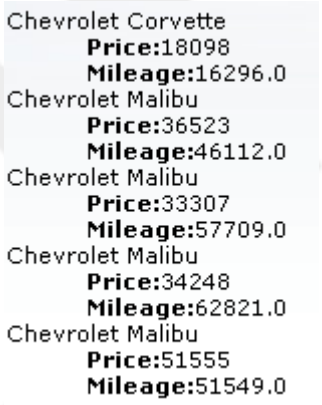


Figure 9.5. `<rich:dataDefinitionList>` example

9.5. `<rich:dataFilterSlider>`

- component-type: `org.richfaces.DataFilterSlider`
- component-class: `org.richfaces.component.html.HtmlDataFilterSlider`
- component-family: `org.richfaces.DataFilterSlider`
- renderer-type: `org.richfaces.DataFilterSliderRenderer`
- tag-class: `org.richfaces.taglib.dataFilterSliderTag`

The `<rich:dataFilterSlider>` components is a slider control that can be used for filtering data in a table. The range and increment of the slider control are defined using the `startRange`, `endRange`, and `increment` attributes.

The slider is bound to a `UIData` component specified with the `for` attribute. The `forValRef` attribute refers to the `value` attribute used by the target component, and the `filterBy` attribute specifies which object member to filter according to the slider.

The `handleValue` attribute keeps the current handle position on the slider control; filtering is based on this value. The `storeResults` attribute allows the `<rich:dataFilterSlider>` component to keep the target `UIData` component in session.

The event defined with the `submitOnSlide` attribute is triggered when the handle value on the slider is changed.

Example 9.8. `<rich:dataFilterSlider>` example

```
<rich:dataFilterSlider sliderListener="#{mybean.doSlide}"
                       startRange="0"
                       endRange="50000"
                       increment="10000"
                       handleValue="1"
                       for="carIndex"
                       forValRef="inventoryList.carInventory"
                       filterBy="getMileage" />

<h:dataTable id="carIndex">
   ...
</h:dataTable>
```

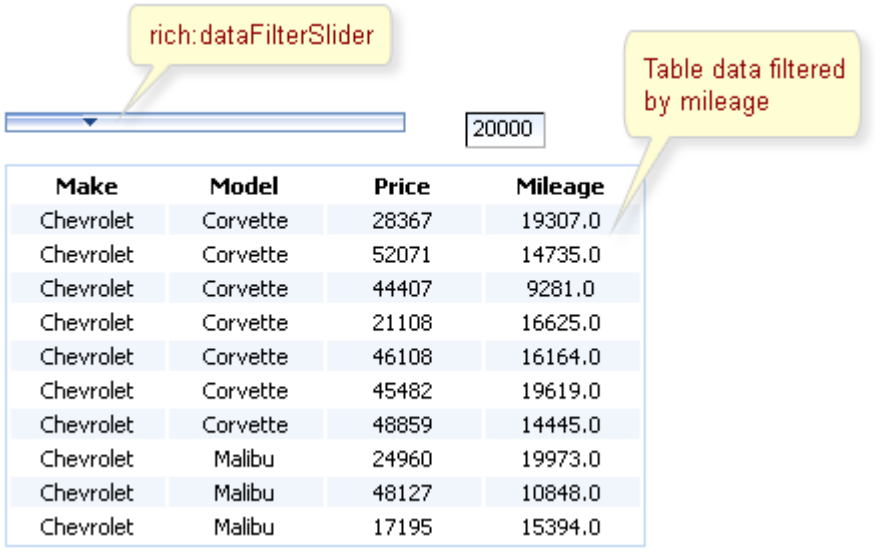


Figure 9.6. `<rich:dataFilterSlider>` example

9.6. `<rich:dataGrid>`

- component-type: `org.richfaces.DataGrid`
- component-class: `org.richfaces.component.html.HtmlDataGrid`
- component-family: `org.richfaces.DataGrid`
- renderer-type: `org.richfaces.DataGridRenderer`
- tag-class: `org.richfaces.taglib.DataGridTag`

The `<rich:dataGrid>` component is used to arrange data objects in a grid. Values in the grid can be updated dynamically from the data model, and Ajax updates can be limited to specific rows. The component supports `header`, `footer`, and `caption` facets.

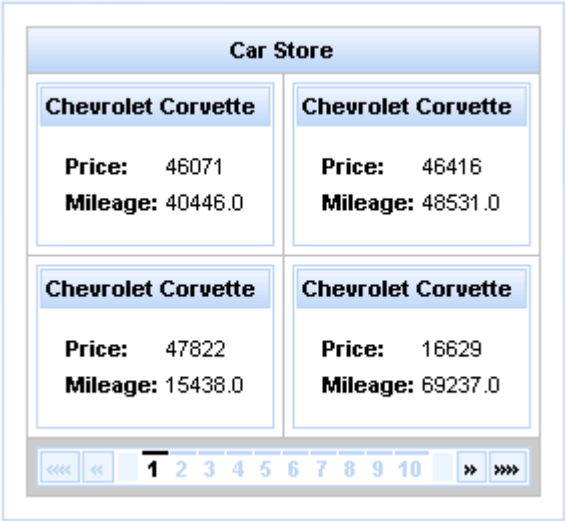


Figure 9.7. The `<rich:dataGrid>` component

The `<rich:dataGrid>` component requires the `value` attribute, which points to the data model, and the `var` attribute, which holds the current variable for the collection of data. The number of columns for the grid is specifed with the `columns` attribute, and the number of elements to layout among the columns is determined with the `elements` attribute. The `first` attribute references the zero-based element in the data model from which the grid starts.

Example 9.9. `<rich:dataGrid>` example

```
<rich:panel style="width:150px;height:200px;">
 <h:form>

 <rich:dataGrid value="#{dataTableScrollerBean.allCars}" var="car" columns="2" elements="4" first="1">
      <f:facet name="header">
       <h:outputText value="Car Store"></h:outputText>
    </f:facet>
    <rich:panel>
       <f:facet name="header">
        <h:outputText value="#{car.make} #{car.model}"></h:outputText>
       </f:facet>
       <h:panelGrid columns="2">
        <h:outputText value="Price:" styleClass="label"></h:outputText>
        <h:outputText value="#{car.price}"/>
        <h:outputText value="Mileage:" styleClass="label"></h:outputText>
        <h:outputText value="#{car.mileage}"/>
       </h:panelGrid>
    </rich:panel>
    <f:facet name="footer">
```

```
        <rich:datascroller></rich:datascroller>
      </f:facet>
    </rich:dataGrid>
  </h:form>
</rich:panel>
```

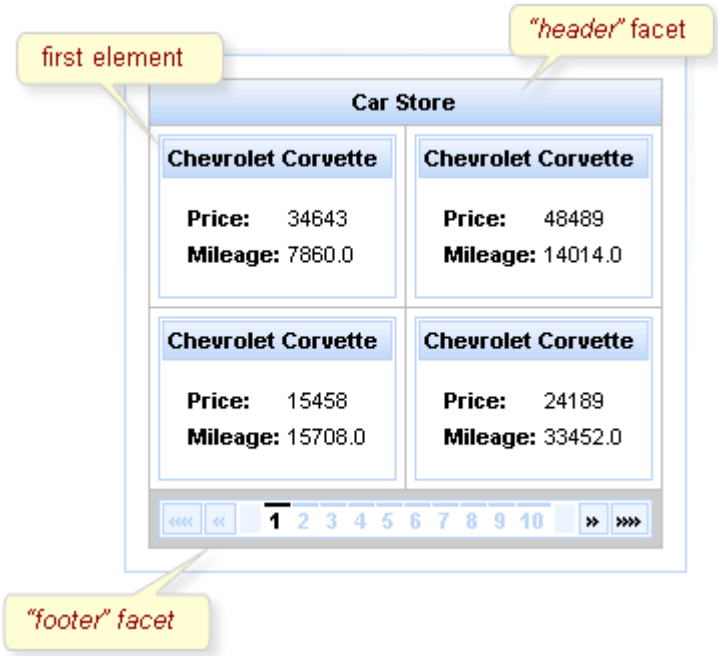


Figure 9.8. `<rich:dataGrid>` example

As the component is based on the `<a4j:repeat>` component, it can be partially updated with Ajax. The `ajaxKeys` attribute allows row keys to be defined, which are updated after an Ajax request.

Example 9.10. `ajaxKeys` example

```
<rich:dataGrid value="#{dataTableScrollerBean.allCars}" var="car" ajaxKeys="#{listBean.list}" binding="#{listBean
  ...
</rich:dataGrid>
...
<a4j:commandButton action="#{listBean.action}" reRender="grid" value="Submit"/>
```

9.7. `<rich:dataList>`

- component-type: `org.richfaces.DataList`
- component-class: `org.richfaces.component.html.HtmlDataList`

- component-family: `org.richfaces.DataList`
- renderer-type: `org.richfaces.DataListRenderer`
- tag-class: `org.richfaces.taglib.DataListTag`

The `<rich:dataList>` component renders an unordered list of items. The component uses a data model for managing the list items, which can be updated dynamically.

The `type` attribute refers to the appearance of the bullet points. The values of the attribute correspond to the `type` parameter for the `<ul>` HTML element:

`circle`
: Displays an unfilled circle as a bullet point.

`disc`
: Displays a filled disc as a bullet point.

`square`
: Displays a square as a bullet point.

The `var` attribute names a variable for iterating through the items in the data model. The items to iterate through are determined with the `value` attribute by using EL (Expression Lanugage). The `first` attribute specifies which item in the data model to start from, and the `rows` attribute specifies the number of items to list. The `title` attribute is used for a floating tool-tip. *Example 9.7, "<rich:dataDefinitionList> example"* shows a simple example using the `<rich:dataDefinitionList>` component.

Example 9.11. `<rich:dataList>` example

```
<h:form>

<rich:dataListvar="car"value="#{dataTableScrollerBean.allCars}"rows="5"type="disc"title="Car
 Store">
      <h:outputText value="#{car.make} #{car.model}"/><br/>
      <h:outputText value="Price:" styleClass="label"></h:outputText>
      <h:outputText value="#{car.price} "/><br/>
      <h:outputText value="Mileage:" styleClass="label"></h:outputText>
      <h:outputText value="#{car.mileage} "/><br/>
   </rich:dataList>
</h:form>
```

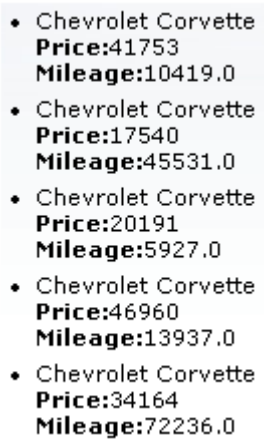


Figure 9.9. `<rich:dataList>` example

9.8. `<rich:dataOrderedList>`

- component-type: `org.richfaces.DataOrderedList`
- component-class: `org.richfaces.component.html.HtmlDataOrderedList`
- component-family: `org.richfaces.DataOrderedList`
- renderer-type: `org.richfaces.DataOrderedListRenderer`
- tag-class: `org.richfaces.taglib.DataOrderedListTag`

The `<rich:dataOrderedList>` component renders an ordered list of items from a data model. Specific rows can be updated dynamically without updating the entire list.

The `type` attribute defines the appearance of the numerating list markers for the list. The possible values for the `type` attribute are as follows:

`A`
: Numerates the list items as *A*, *B*, *C*, etc.

`a`
: Numerates the list items as *a*, *b*, *c*, etc.

`I`
: Numerates the list items as *I*, *II*, *III*, etc.

`i`
: Numerates the list items as *i*, *ii*, *iii*, etc.

`1`
: Numerates the list items as *1*, *2*, *3*, etc.

The `first` attribute specifies which item in the data model to start from, and the `rows` attribute specifies the number of items to list. The `title` attribute defines a pop-up title. To only update a sub-set of the rows in the list, use the `ajaxKeys` attribute, which points to an object that contains the specified rows.

Example 9.12. `<rich:dataOrderedList>` example

```
<h:form>

<rich:dataOrderedList var="car" value="#{dataTableScrollerBean.allCars}" rows="5" type="1" title="Car
  Store">
      <h:outputText value="#{car.make} #{car.model}"/><br/>
      <h:outputText value="Price:" styleClass="label"></h:outputText>
      <h:outputText value="#{car.price}" /><br/>
      <h:outputText value="Mileage:" styleClass="label"></h:outputText>
      <h:outputText value="#{car.mileage}" /><br/>
    </rich:dataOrderedList>
</h:form>
```

1. Chevrolet Corvette
   Price:16080
   Mileage:55773.0
2. Chevrolet Corvette
   Price:49936
   Mileage:72356.0
3. Chevrolet Corvette
   Price:52167
   Mileage:30749.0
4. Chevrolet Corvette
   Price:21148
   Mileage:55447.0
5. Chevrolet Corvette
   Price:18098
   Mileage:16296.0

Figure 9.10. `<rich:dataOrderedList>` example

9.9. `<rich:dataTable>`

- component-type: `org.richfaces.DataTable`
- component-class: `org.richfaces.component.html.HtmlDataTable`
- component-family: `org.richfaces.DataTable`
- renderer-type: `org.richfaces.DataTableRenderer`
- tag-class: `org.richfaces.taglib.DataTableTag`

The `<rich:dataTable>` component is used to render a table, including the table's header and footer. It works in conjunction with the `<rich:column>`, `<rich:columnGroup>`, and `<rich:columns>` components to list the contents of a data model.

`<rich:extendedDataTable>`

The `<rich:dataTable>` component does not include extended table features, such as data scrolling, row selection, and column reordering. These features are available as part of the `<rich:extendedDataTable>` component; refer to *Section 9.10, "<rich:extendedDataTable>"* for further details.

The `value` attribute points to the data model, and the `var` attribute specifies a variable to use when iterating through the data model. The `first` attribute specifies which item in the data model to start from, and the `rows` attribute specifies the number of items to list. The `header`, `footer`, and `caption` facets can be used to display text, and to customize the appearance of the table through skinning. demonstrates a simple table implementation.

Example 9.13. `<rich:dataTable>` example

```
<rich:dataTable value="#{capitalsBean.capitals}" var="cap" rows="5">
   <f:facet name="caption">
      <h:outputText value="United States Capitals" />
   </f:facet>
   <f:facet name="header">
      <h:outputText value="Capitals and States Table" />
   </f:facet>
   <rich:column>
      <f:facet name="header">State Flag</f:facet>
         <h:graphicImage value="#{cap.stateFlag}"/>
      <f:facet name="footer">State Flag</f:facet>
   </rich:column>
   <rich:column>
      <f:facet name="header">State Name</f:facet>
         <h:outputText value="#{cap.state}"/>
      <f:facet name="footer">State Name</f:facet>
   </rich:column>
   <rich:column >
      <f:facet name="header">State Capital</f:facet>
         <h:outputText value="#{cap.name}"/>
      <f:facet name="footer">State Capital</f:facet>
   </rich:column>
   <rich:column>
      <f:facet name="header">Time Zone</f:facet>
```

```
        <h:outputText value="#{cap.timeZone}"/>
      <f:facet name="footer">Time Zone</f:facet>
    </rich:column>
    <f:facet name="footer">
      <h:outputText value="Capitals and States Table" />
    </f:facet>
</rich:dataTable>
```

United States Capitals

Capitals and States Table			
State Flag	Capital Name	State Name	TimeZone
	Montgomery	Alabama	GMT-6
	Juneau	Alaska	GMT-9
	Phoenix	Arizona	GMT-7
	Little Rock	Arkansas	GMT-6
	Sacramento	California	GMT-8
State Flag	Capital Name	State Name	TimeZone
Capitals and States Table			

Figure 9.11. `<rich:dataTable>` example

The `<rich:dataTable>` component is based on the `<a4j:repeat>` component, and as such it can be partially updated using Ajax. The `ajaxKeys` attribute defines the rows to be updated during an Ajax request.

For details on filtering and sorting data tables, refer to *Section 9.11, "Table filtering"* and *Section 9.12, "Table sorting"*.

9.10. `<rich:extendedDataTable>`

- component-type: `org.richfaces.ExtendedDataTable`
- component-class: `org.richfaces.component.html.HtmlExtendedDataTable`
- component-family: `org.richfaces.ExtendedDataTable`
- renderer-type: `org.richfaces.ExtendedDataTableRenderer`
- tag-class: `org.richfaces.taglib.ExtendedDataTableTag`

The `<rich:extendedDataTable>` component builds on the functionality of the `<rich:dataTable>` component, adding features such as data scrolling, row and column selection, and rearranging of columns.

The `<rich:extendedDataTable>` component includes the following attributes not included in the `<rich:dataTable>` component:

`activeClass`	`height`	`selection`
`activeRowKey`	`noDataLabel`	`selectionMode`
`enableContextMenu`	`onselectionchange`	`tableState`
`groupingColumn`	`selectedClass`	

The `<rich:extendedDataTable>` component does *not* include the following attributes available with the `<rich:dataTable>` component:

- `columns`
- `columnsWidth`
- `onRowContextMenu`

Basic use of the `<rich:extendedDataTable>` component requires the `value` and `var` attributes, the same as with the `<rich:dataTable>` component. Refer to *Section 9.9, "<rich:dataTable>"* for details.

The `height` attribute is mandatory, and defines the height of the table on the page. This is set to `500px` by default. The width of the table can be set by using the `width` attribute. As with the `<rich:dataTable>` component, the look of the `<rich:extendedDataTable>` component can be customized and skinned using the `header`, `footer`, and `caption` facets.

Example 9.14. `<rich:extendedDataTable>` example

```
<rich:extendedDataTable id="edt" value="#{extendedDT.dataModel}" var="edt" width="500px" height="500px" sel

 <rich:column id="id" headerClass="dataTableHeader" width="50" label="Id" sortable="true" sortBy="#{edt.id}" s
     <f:facet name="header">
        <h:outputText value="Id" />
     </f:facet>
     <h:outputText value="#{edt.id}" />
  </rich:column>

 <rich:column id="name" width="300" headerClass="dataTableHeader" label="Name" sortable="true" sortBy="#
     <f:facet name="header">
        <h:outputText value="Name" />
     </f:facet>
     <h:outputText value="#{edt.name}" />
  </rich:column>

 <rich:column id="date" width="100" headerClass="dataTableHeader" label="Date" sortable="true" comparator=
```

```
        <f:facet name="header">
            <h:outputText value="Date" />
        </f:facet>
       <h:outputText value="#{edt.date}"><f:convertDateTime pattern="yyyy-MM-dd HH:mm:ss" />
        </h:outputText>
    </rich:column>

  <rich:column id="group" width="50" headerClass="dataTableHeader" label="Group" sortable="true" sortBy="#{
        <f:facet name="header">
            <h:outputText value="Group" />
        </f:facet>
        <h:outputText value="#{edt.group}" />
    </rich:column>
</rich:extendedDataTable>
```

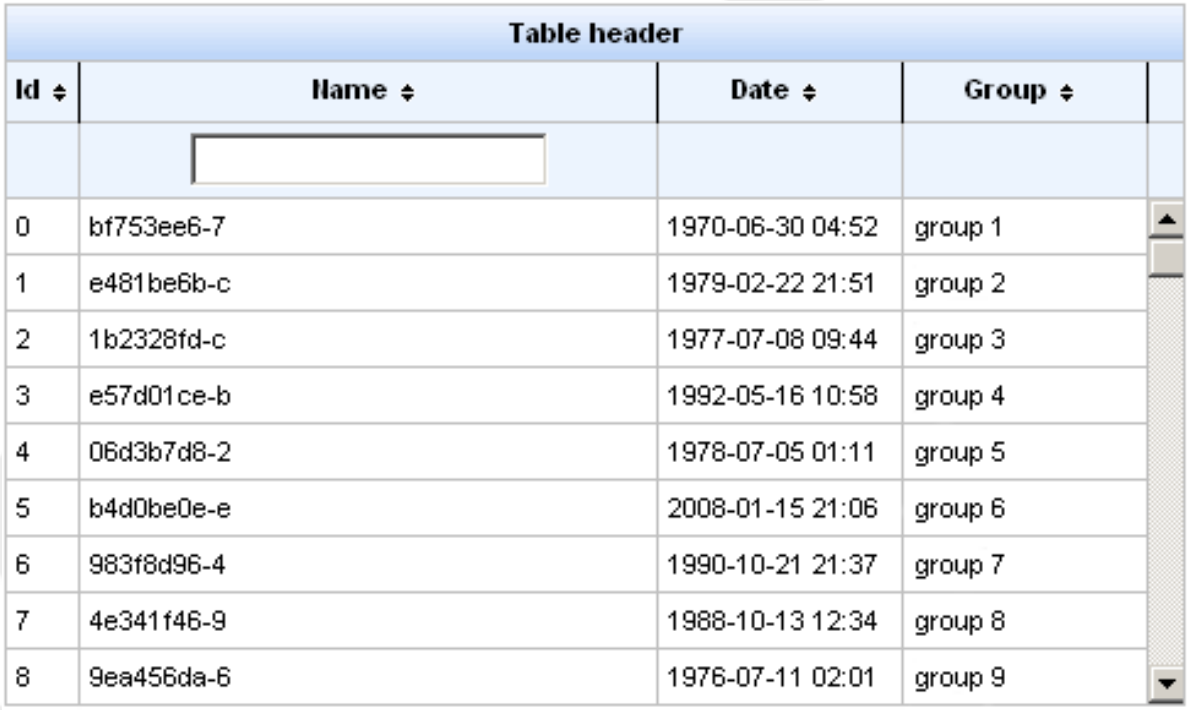


Table header			
Id	Name	Date	Group
0	bf753ee6-7	1970-06-30 04:52	group 1
1	e481be6b-c	1979-02-22 21:51	group 2
2	1b2328fd-c	1977-07-08 09:44	group 3
3	e57d01ce-b	1992-05-16 10:58	group 4
4	06d3b7d8-2	1978-07-05 01:11	group 5
5	b4d0be0e-e	2008-01-15 21:06	group 6
6	983f8d96-4	1990-10-21 21:37	group 7
7	4e341f46-9	1988-10-13 12:34	group 8
8	9ea456da-6	1976-07-11 02:01	group 9

Figure 9.12. `<rich:extendedDataTable>` example

Example 9.14, "<rich:extendedDataTable> example" shows an example extended data table. The implementation features a scrolling data table, selection of one or more rows, sorting by columns, grouping by column, and a filter on the **Name** column.

Row selection is determined by the `selectionMode` attribute. Setting the attribute to `none` allows for no row selection capability. Setting the `selectionMode` attribute to `single` allows the user to select a single row at a time using the mouse. With the `selectionMode` attribute set to `multi`, the user can select multiple rows by holding down the **Shift** or **Ctrl** keys while clicking. The `selection`

attribute points to the object that tracks which rows are selected. *Figure 9.13, "Selecting multiple rows"* shows the table from the example with multiple rows selected.

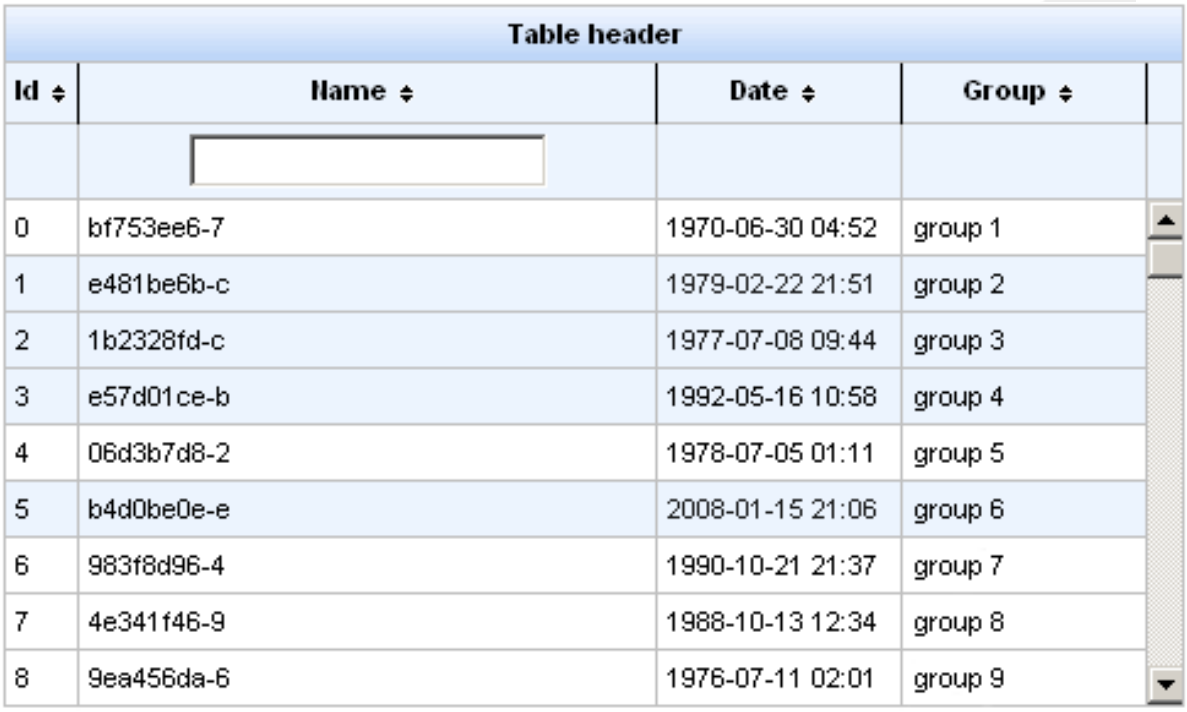


Table header			
Id	Name	Date	Group
0	bf753ee6-7	1970-06-30 04:52	group 1
1	e481be6b-c	1979-02-22 21:51	group 2
2	1b2328fd-c	1977-07-08 09:44	group 3
3	e57d01ce-b	1992-05-16 10:58	group 4
4	06d3b7d8-2	1978-07-05 01:11	group 5
5	b4d0be0e-e	2008-01-15 21:06	group 6
6	983f8d96-4	1990-10-21 21:37	group 7
7	4e341f46-9	1988-10-13 12:34	group 8
8	9ea456da-6	1976-07-11 02:01	group 9

Figure 9.13. Selecting multiple rows

The example uses the `filter` facet of the `<rich:column>` component to display the text field. A user can type their criteria into the text field to customize the filter of the column below. For full details on filtering tables, refer to *Section 9.11, "Table filtering"*.

When hovering the mouse over a column header, a menu button appears to the right-hand side, as shown in *Figure 9.14, "Column menu"*. This menu allows the user to sort the contents of the column, group the table by the column, or hide and show columns.

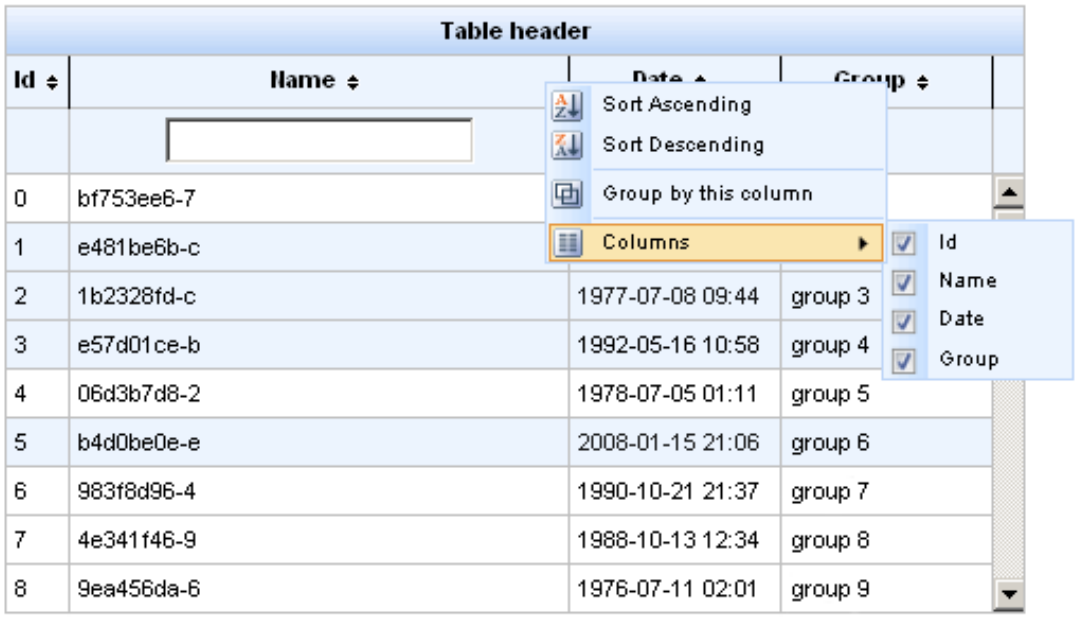


Figure 9.14. Column menu

Each column can allow sorting by setting the `<rich:column>` component's `sortable` attribute to `true`. The value of the data model to sort by is specified with the `sortBy` attribute. In addition to using the menu for sorting, columns can be quickly sorted either ascending or descending by clicking on the directional icon next to the column title. The directional icons are defined in each `<rich:column>` component with the `sortIconAscending` and `sortIconDescending` attributes, for ascending and descending icons respectively. For full details on sorting tables, refer to *Section 9.12, "Table sorting"*.

Grouping the table's entries by a column will organize the table into collapsible sections, as shown in *Figure 9.15, "Grouping table entries by column"*.

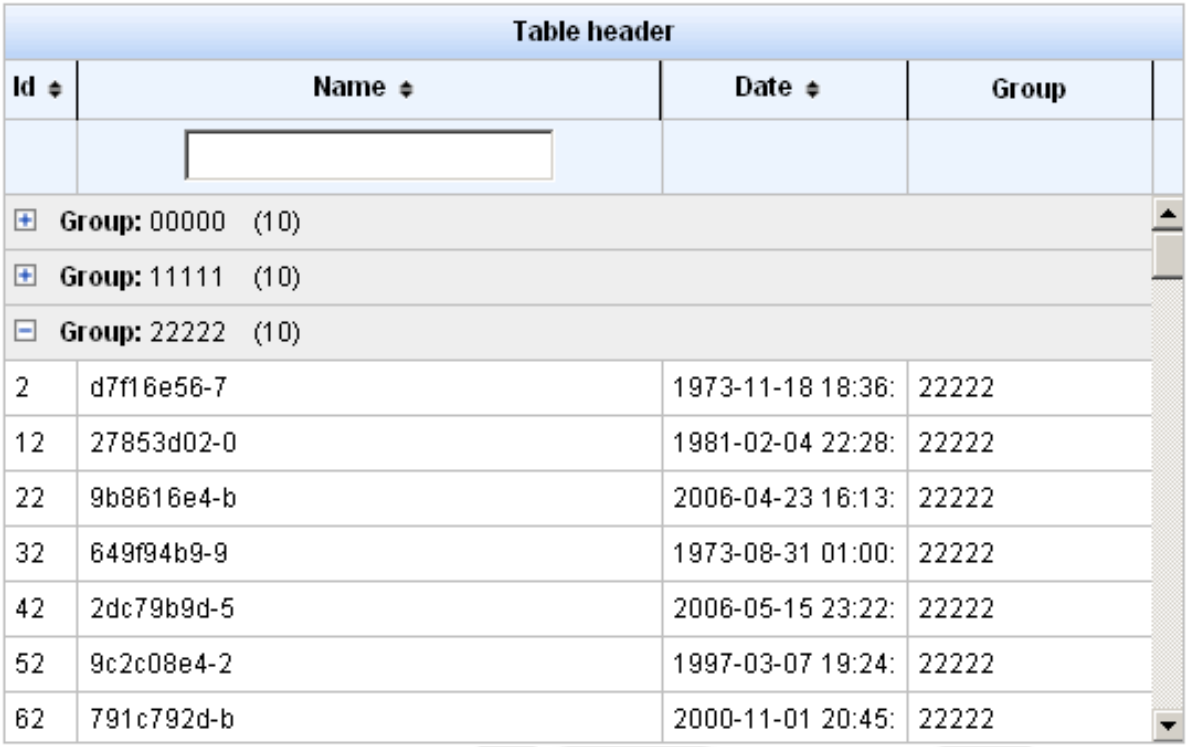


Figure 9.15. Grouping table entries by column

The checkboxes in the column menu under the **Columns** sub-menu will hide or show the respective columns, as shown in *Figure 9.16, "Hiding columns"*.

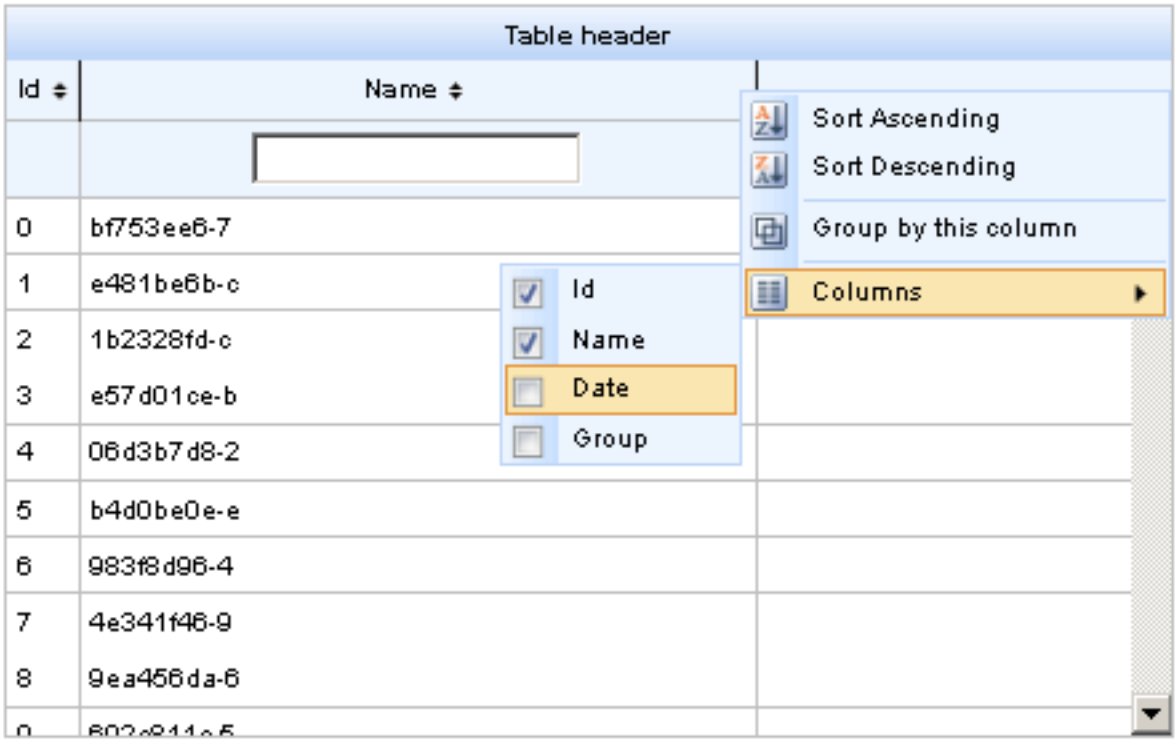


Figure 9.16. Hiding columns

Columns in a `<rich:extendedDataTable>` component can be rearranged by the user by dragging each column to a different position. The `label` attribute for the `<rich:column>` component is displayed during dragging, as shown in

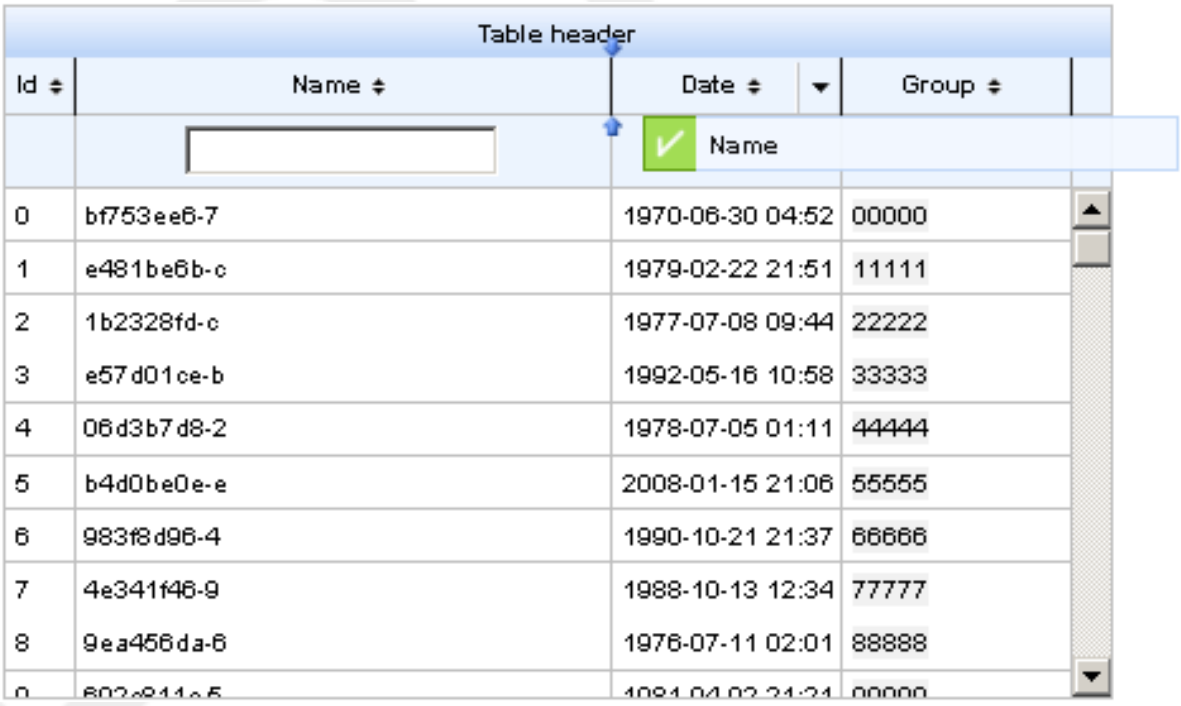


Figure 9.17. Dragging columns

Once the contents of the table have been rearranged and customized by the user, the `tableState` attribute can be used to preserve the customization so it can be restored later. The `tableState` attribute points to a backing-bean property which can in turn be saved to a database separate from standard JSF state-saving mechanisms.

9.11. Table filtering

Under development.

9.12. Table sorting

Under development.

Functions

Read this chapter for details on special functions for use with particular components. Using JavaServer Faces Expression Language (JSF EL), these functions can be accessed through the `data` attribute of components. Refer to *Section 2.5.7, "data"* for details on the `data` attribute.

10.1. `rich:clientId`

The `rich:clientId('id')` function returns the client identifier related to the passed component identifier (`'id'`). If the specified component identifier is not found, `null` is returned instead.

10.2. `rich:component`

The `rich:component('id')` function is a shortcut for the equivalent `#{rich:clientId('id')}.component` code. It returns the UIComponent instance from the client, based on the passed server-side component identifier (`'id'`). If the specified component identifier is not found, `null` is returned instead.

10.3. `rich:element`

The `rich:element('id')` function is a shortcut for the equivalent `document.getElementById(#{rich:clientId('id')})` code. It returns the element from the client, based on the passed server-side component identifier. If the specified component identifier is not found, `null` is returned instead.

10.4. `rich:findComponent`

The `rich:findComponent('id')` function returns the a UIComponent instance of the passed component identifier. If the specified component identifier is not found, `null` is returned instead.

Example 10.1. `rich:findComponent` example

```
<h:inputText id="myInput">
   <a4j:support event="onkeyup" reRender="outtext"/>
</h:inputText>
<h:outputText id="outtext" value="#{rich:findComponent('myInput').value}" />
```

10.5. `rich:isUserInRole`

The `rich:isUserInRole(Object)` function checks whether the logged-in user belongs to a certain user role, such as being an administrator. User roles are defined in the `web.xml` settings file.

Example 10.2. `rich:isUserInRole` example

The `rich:isUserInRole(Object)` function can be used in conjunction with the `rendered` attribute of a component to only display certain controls to authorized users.

```
<rich:editor value="#{bean.text}" rendered="#{rich:isUserInRole('admin')}" />
```