Component Reference

A reference guide to the components of the RichFaces 4 *(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>`

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

4.1.1. Basic usage

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.

Example 4.1. `<a4j:actionParam>` example

```
<h:form id="form">
    <a4j:commandButton value="Set name to Alex" reRender="rep">

<a4j:actionparamname="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;
    }
```

```
}
```

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.

4.1.2. Interoperability

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.

4.1.3. Reference data

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

4.2. `<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 submits an Ajax form, and when a response is received the command button can be dynamically rendered.

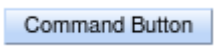

Figure 4.1. `<a4j:commandButton>`

4.2.1. Basic usage

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.

4.2.2. Reference data

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

4.3. <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.

Figure 4.2. <a4j:commandLink>

4.3.1. Basic usage

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*.

4.3.2. Reference data

- *component-type*: org.ajax4jsf.CommandLink
- *component-class*: org.ajax4jsf.component.html.HtmlAjaxCommandLink
- *component-family*: javax.faces.Command

- *renderer-type*: `org.ajax4jsf.components.AjaxCommandLinkRenderer`

4.4. `<rich:componentControl>`

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.

4.4.1. Basic usage

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`.

4.4.2. Attaching to a component

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`.

4.4.3. Parameters and JavaScript

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="menu" 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.

4.4.4. Timing

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.

4.4.5. Reference data

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

4.5. <a4j:jsFunction>

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.

4.5.1. Basic usage

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

4.5.2. Parameters and JavaScript

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.

4.5.3. Reference data

- *component-type*: org.ajax4jsf.Function
- *component-class*: org.ajax4jsf.component.html.HtmlajaxFunction

- *component-family*: `org.ajax4jsf.components.ajaxFunction`
- *renderer-type*: `org.ajax4jsf.components.ajaxFunctionRenderer`

4.6. `<a4j:poll>`

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.

4.6.1. Timing options

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.

4.6.2. Reference data

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

4.7. `<a4j:push>`

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.

4.7.1. Timing options

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.

4.7.2. Reference data

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

4.8. `<a4j:ajax>`

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 4.6. `<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>
```

4.8.1. Reference data

- *component-type*: `org.ajax4jsf.Ajax`
- *component-class*: `org.ajax4jsf.component.html.HtmlAjaxSupport`
- *component-family*: `org.ajax4jsf.Ajax`

- *renderer-type*: `org.ajax4jsf.components.AjaxSupportRenderer`

Resources

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

5.1. `<a4j:keepAlive>`

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 (`IllegalArgumentException`). The `<a4j:keepAlive>` component avoids this by maintaining the state of the whole bean object for subsequent requests.

5.1.1. Basic usage

The `beanName` attribute defines the request-scope managed bean name to keep alive.

Example 5.1. `<a4j:keepAlive>` example

```
<a4j:keepAlive beanName="testBean" />
```

5.1.2. Non-Ajax requests

The `ajaxOnly` attribute determines whether or not the value of the bean should be available during non-Ajax 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.

5.1.3. Reference data

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

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:include>`

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.

6.1.1. Basic usage

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>
```

6.1.2. Reference data

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

6.2. `<a4j:outputPanel>`

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.

6.2.1. Panel appearance

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.

6.2.2. Reference data

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

6.3. `<a4j:region>`

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.

6.3.1. Reference data

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

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>`

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>
```

7.1.1. Custom validators

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;
```

```
    }
}
```

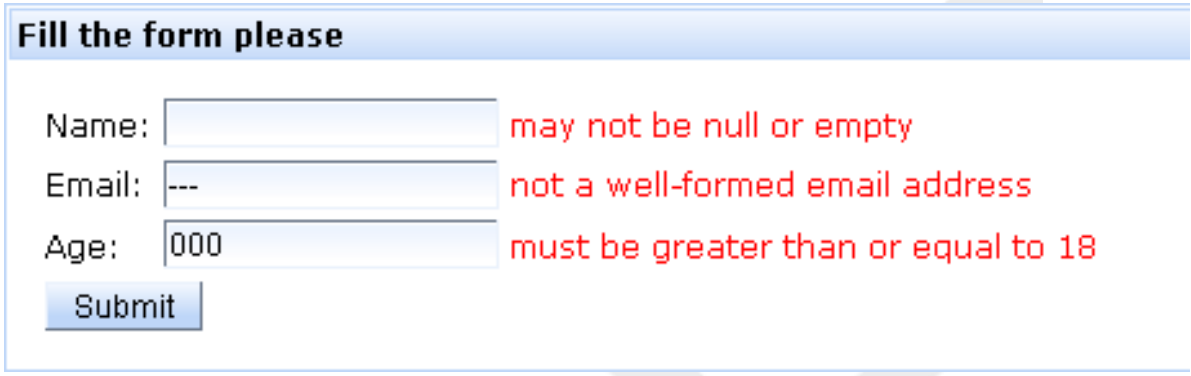

Figure 7.1. `<rich:ajaxValidator>` example result

7.1.2. Reference data

- *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`

7.2. `<rich:beanValidator>`

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.

7.2.1. Basic usage

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:commandButton value="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;
    }
}
```

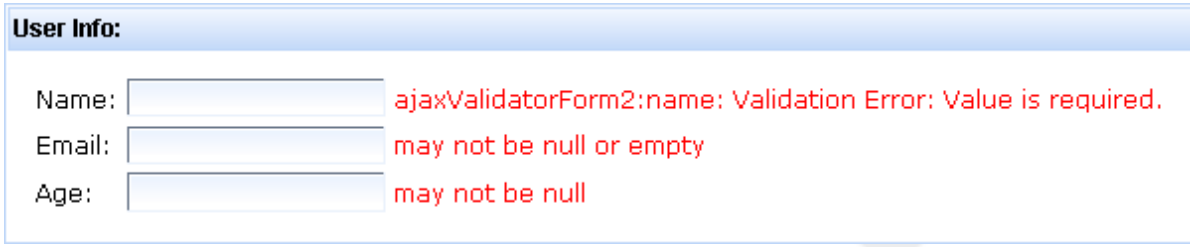

Figure 7.2. `<rich:beanValidator>` example result

7.2.2. Reference data

- *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`

7.3. `<rich:graphValidator>`

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.

7.3.1. Basic usage

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;
    }
}
```

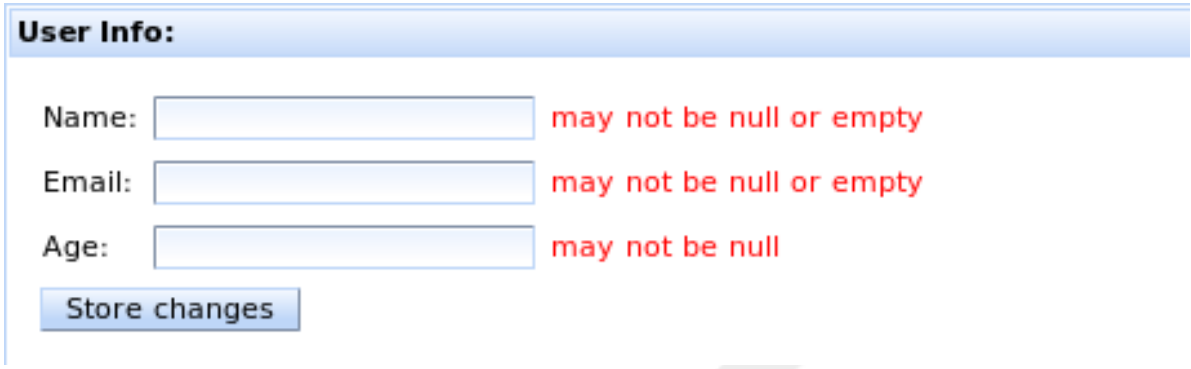

Figure 7.3. `<rich:graphValidator>` example result

7.3.2. Bean values

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>
```

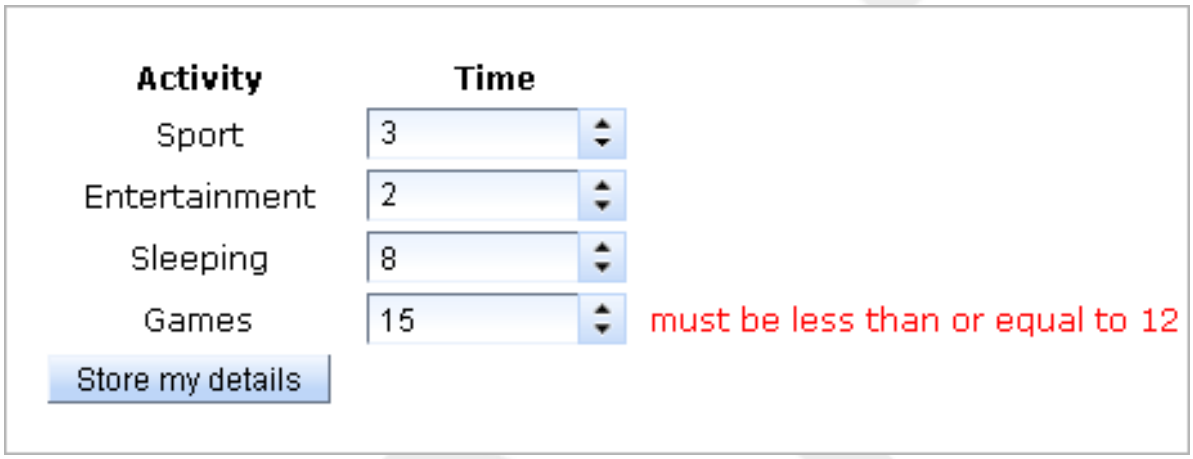

Figure 7.4. Result from using the `value` attribute

7.3.3. Reference data

- *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`

Processing management

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

8.1. `<a4j:queue>`

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

8.1.1. Queue size

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.

8.1.2. `<a4j:queue>` client-side events

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 but before the DOM is updated on the client side.

8.1.3. Reference data

- *`renderer-type`*: `org.ajax4jsf.QueueRenderer`

- *component-class*: org.ajax4jsf.component.html.HtmlQueue
- *component-family*: org.ajax4jsf.Queue
- *tag-class*: org.ajax4jsf.taglib.html.jsp.QueueTag

8.2. <a4j:log>

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

8.2.1. Log monitoring

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.

8.2.2. Reference data

- *component-type*: org.ajax4jsf.Log
- *component-class*: org.ajax4jsf.component.html.AjaxLog

- *component-family*: org.ajax4jsf.Log
- *renderer-type*: org.ajax4jsf.LogRenderer

8.3. <a4j:status>

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

8.3.1. Customizing the text

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" />
```

8.3.2. Specifying a region

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>
```

8.3.3. Reference data

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

Part II. rich tag library

Rich inputs

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 details those components which act as panels and containers to hold groups of other components.

9.1. `<rich:autocomplete>`

The `<rich:autocomplete>` component is an auto-completing input-box with built-in Ajax capabilities. It supports client-side suggestions, browser-like selection, and customization of the look and feel.

To attach an auto-completion behavior to other components, use the `<rich:autocompleteBehavior>` behavior. Refer to *???* for full details on the `<rich:autocompleteBehavior>` behavior.

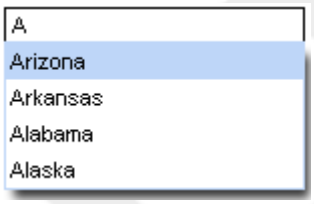

Figure 9.1. `<rich:autocomplete>`

9.1.1. Basic usage

The `value` attribute stores the text entered by the user for the auto-complete box. Suggestions shown in the auto-complete list can be specified using the `autocompleteMethod` attribute, which points to a collection of suggestions.

Example 9.1. Defining suggestion values

```
<rich:autocomplete value="#{bean.state}" autocompleteMethod="#{bean.suggestions}" />
```

9.1.2. Interactivity options

Users can type into the combo-box's text field to enter a value, which also searches through the suggestion items in the drop-down box. By default, the first suggestion item is selected as the user types. This behavior can be deactivated by setting `selectFirst="false"`.

Setting `autoFill="true"` causes the combo-box to fill the text field box with a matching suggestion as the user types.

9.1.3. Reference data

- *`component-type`*: `org.richfaces.autocomplete`
- *`component-class`*: `org.richfaces.component.html.HtmlAutocomplete`
- *`component-family`*: `org.richfaces.autocomplete`
- *`renderer-type`*: `org.richfaces.renderkit.autocompleteRenderer`
- *`tag-class`*: `org.richfaces.taglib.autocompleteTag`

9.2. `<rich:inplaceInput>`

The `<rich:inplaceInput>` component allows information to be entered in-line in blocks of text, improving readability of the text. Multiple input regions can be navigated with keyboard navigation. The component has three functional states: the "view" state, where the component displays its initial setting, such as "click to edit"; the "edit" state, where the user can input text; and the "changed" state, where the new value for the component has been confirmed but can be edited again if required.

9.2.1. Basic usage

Basic usage requires the `value` attribute to point to the expression for the current value of the component.

9.2.2. Interactivity options

When in the initial "view" state, the starting label can be set using the `defaultLabel` attribute. Once the user has entered text, the label is stored in the model specified by the `value` attribute. The use of the default label and value is shown in *Example 9.2, “Default label and value”*.

Example 9.2. Default label and value

```
<rich:inplaceInput value="#{bean.value}" defaultLabel="click to edit"/>
```

By default, the event to switch the component to the "edit" state is a single mouse click. This can be changed using the `editEvent` attribute to specify a different event.

The user can confirm and save their input by pressing the **Enter** key or cancel by pressing the **Esc** key. Alternatively, buttons for confirming or canceling can be added to the component by setting `showControls="true"`.

9.2.3. Reference data

- *`component-type`*: `org.richfaces.inplaceInput`
- *`component-class`*: `org.richfaces.component.html.HtmlInplaceInput`
- *`component-family`*: `org.richfaces.inplaceInput`
- *`renderer-type`*: `org.richfaces.renderkit.inplaceInputRenderer`
- *`tag-class`*: `org.richfaces.taglib.inplaceInputTag`

9.3. `<rich:inputNumberSlider>`

The `<rich:inputNumberSlider>` component provides a slider for changing numerical values. Optional features include control arrows to step through the values, a tool-tip to display the value while sliding, and a text field for typing the numerical value which can then be validated against the slider's range.

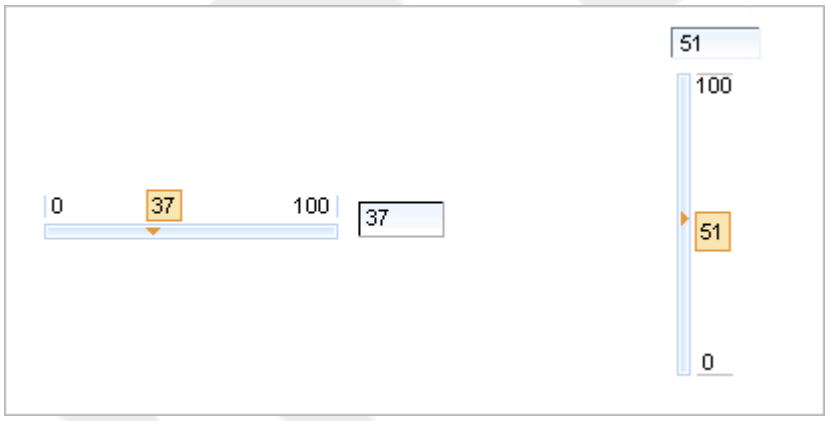

Figure 9.2. `<rich:inputNumberSlider>`

9.3.1. Basic usage

Basic use of the component with no attributes specified will render a slider with a minimum value of 0, a maximum of 100, and a gradient step of 1, together with a text field for typing the desired numerical value. The slider is labeled with the minimum and maximum boundary values, and a tool-tip showing the current value is shown while sliding the slider. The `value` attribute is used for storing the currently selected value of the slider.

9.3.2. Interactivity options

The text field can be removed by setting `showInput="false"`.

The properties of the slider can be set with the attributes `minValue`, `maxValue`, and `step`.

The minimum and maximum labels on the slider can be hidden by setting `showBoundaryValues="false"`. The tool-tip showing the current value can be hidden by setting `showToolTip="false"`.

Arrow controls can be added to either side of the slider to adjust the value incrementally by setting `showArrows="true"`. Clicking the arrows move the slider indicator in that direction by the gradient step, and clicking and holding the arrows moves the indicator continuously. The time delay for each step when updating continuously can be defined using the `delay` attribute.

9.3.3. Reference data

- *`component-type`*: `org.richfaces.inputNumberSlider`
- *`component-class`*: `org.richfaces.component.html.HtmlInputNumberSlider`
- *`component-family`*: `org.richfaces.inputNumberSlider`
- *`renderer-type`*: `org.richfaces.renderkit.inputNumberSliderRenderer`
- *`tag-class`*: `org.richfaces.taglib.inputNumberSliderTag`

9.4. `<rich:inputNumberSpinner>`

The `<rich:inputNumberSpinner>` component is a single-line input field with buttons to increase and decrease a numerical value. The value can be changed using the corresponding directional keys on a keyboard, or by typing into the field.

Figure 9.3. `<rich:inputNumberSpinner>`

9.4.1. Basic usage

Basic use of the component with no attributes specified will render a number spinner with a minimum value of 1, a maximum value of 100, and a gradient step of 1.

These default properties can be re-defined with the attributes `minValue`, `maxValue`, and `step` respectively. The starting value of the spinner is the minimum value unless otherwise specified with the `value` attribute.

9.4.2. Interactivity options

When changing the value using the buttons, raising the value above the maximum or cause the spinner to restart at the minimum value. Likewise, when lowering below the minimum value the spinner will reset to the maximum value. This behavior can be deactivated by setting `cycled="false"`, which will cause the buttons to stop responding when the reach the maximum or minimum value.

The ability to change the value by typing into the text field can be disabled by setting `enableManualInput="false"`.

9.4.3. Reference data

- *`component-type`*: `org.richfaces.inputNumberSpinner`
- *`component-class`*: `org.richfaces.component.html.HtmlInputNumberSpinner`
- *`component-family`*: `org.richfaces.inputNumberSpinner`
- *`renderer-type`*: `org.richfaces.renderkit.inputNumberSpinnerRenderer`
- *`tag-class`*: `org.richfaces.taglib.inputNumberSpinnerTag`

Panels and containers

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 details those components which act as panels and containers to hold groups of other components.

10.1. `<rich:accordion>`

The `<rich:accordion>` is a series of panels stacked on top of each other, each collapsed such that only the header of the panel is showing. When the header of a panel is clicked, it is expanded to show the content of the panel. Clicking on a different header will collapse the previous panel and epand the selected one. Each panel contained in a `<rich:accordion>` component is a `<rich:accordionItem>` component.

Figure 10.1. A `<rich:accordion>` component containing three `<rich:accordionItem>` components

10.1.1. Basic usage

The `<rich:accordion>` component requires no attributes for basic usage. The component can contain any number of `<rich:accordionItem>` components as children. The headers of the `<rich:accordionItem>` components control the expanding and collapsing when clicked. Only a single `<rich:accordionItem>` can be displayed at a time. Refer to *Section 10.2, "<rich:accordionItem>"* for details on the `<rich:accordionItem>` component.

10.1.2. Switching panels

The switching mode for performing submissions is determined by the `switchType` attribute, which can have one of the following three values:

`server`
: The default setting. Activation of a `<rich:accordionItem>` component causes the parent `<rich:accordion>` component to perform a common submission, completely re-rendering the page. Only one panel at a time is uploaded to the client side.

`ajax`
: Activation of a `<rich:accordionItem>` component causes the parent `<rich:accordion>` component to perform an Ajax form submission, and the content of the panel is rendered. Only one panel at a time is uploaded to the client side.

`client`
: Activation of a `<rich:accordionItem>` component causes the parent `<rich:accordion>` component to update on the client side. JavaScript changes the styles such that one panel component becomes hidden while the other is shown.

10.1.3. `<rich:accordion>` client-side events

In addition to the standard Ajax events and HTML events, the `<rich:accordion>` component uses the client-side events common to all switchable panels:

- The `onitemchange` event points to the function to perform when the switchable item is changed.
- The `onbeforeitemchange` event points to the function to perform when before the switchable item is changed.

10.1.4. `<rich:accordion>` server-side events

The `<rich:accordion>` component uses the server-side events common to all switchable panels:

- The `ItemChangeEvent` event occurs on the server side when an item is changed through Ajax using the `server` mode. It can be processed using the `ItemChangeListener` attribute.

10.1.5. Reference data

- *`component-type`*: `org.richfaces.accordion`
- *`component-class`*: `org.richfaces.component.html.HtmlAccordion`
- *`component-family`*: `org.richfaces.accordion`
- *`renderer-type`*: `org.richfaces.accordionRenderer`

- *tag-class*: `org.richfaces.taglib.accordionTag`

10.2. `<rich:accordionItem>`

The `<rich:accordionItem>` component is a panel for use with the `<rich:accordion>` component. Refer to *Section 10.1, "<rich:accordion>"* for details on the `<rich:accordion>` component.

Figure 10.2. A `<rich:accordion>` component containing three `<rich:accordionItem>` components

10.2.1. Basic usage

Basic usage of the `<rich:accordionItem>` component requires the `label` attribute, which provides the text on the panel header. The panel header is all that is visible when the accordion item is collapsed.

Alternatively the `header` facet could be used in place of the `label` attribute. This would allow for additional styles and custom content to be applied to the tab.

10.2.2. `<rich:accordionItem>` client-side events

In addition to the standard HTML events, the `<rich:accordionItem>` component uses the client-side events common to all switchable panel items:

- The `onenter` event points to the function to perform when the mouse enters the panel.
- The `onleave` attribute points to the function to perform when the mouse leaves the panel.

10.2.3. Reference data

- *component-type*: `org.richfaces.accordionItem`
- *component-class*: `org.richfaces.component.html.HtmlAccordionItem`

- *component-family*: org.richfaces.accordionItem
- *renderer-type*: org.richfaces.accordionItemRenderer
- *tag-class*: org.richfaces.taglib.accordionItemTag

10.3. <rich:panel>

The <rich:panel> component is a bordered panel with an optional header.

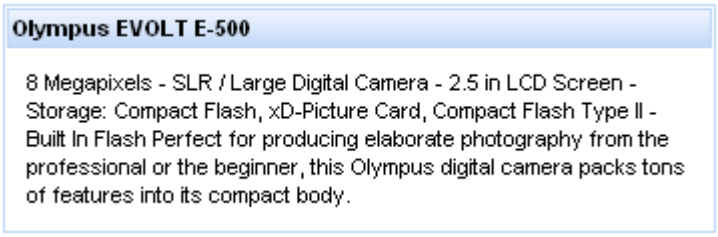

Figure 10.3. <rich:panel>

10.3.1. Basic usage

No attributes need to be listed for basic usage. a <rich:panel> without any attributes defined renders a bordered region with no header.

10.3.2. Adding a header

To add a header to the panel, use the header attribute to specify the text to appear in the header. Alternatively the header can be constructed using a header facet. *Example 10.1, "Adding a header"* demonstrates the two different approaches.

Example 10.1. Adding a header

```
<rich:panel header="This is the panel header">
    <h:outputText value="This is the panel content" />
</rich:panel>
```

```
<rich:panel>
    <f:facet name="header">
        <h:outputText value="This is the panel header">
    </f:facet>
    <h:outputText value="This is the panel content" />
</rich:panel>
```

Both the examples render an identical panel.

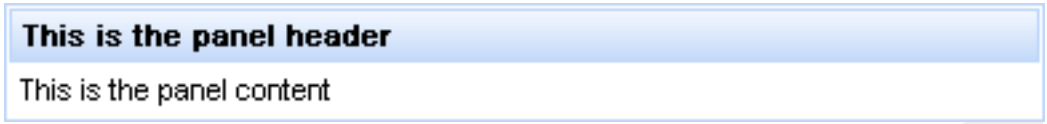

Figure 10.4. Adding a header

10.3.3. Reference data

- *component-type*: `org.richfaces.panel`
- *component-class*: `org.richfaces.component.html.HtmlPanel`
- *component-family*: `org.richfaces.panel`
- *renderer-type*: `org.richfaces.panelRenderer`
- *tag-class*: `org.richfaces.taglib.panelTag`

10.4. `<rich:popupPanel>`

The `<rich:popupPanel>` component provides a pop-up panel or window that appears in front of the rest of the application. The `<rich:popupPanel>` component functions either as a modal window which blocks interaction with the rest of the application while active, or as a non-modal window. It can be positioned on the screen, dragged to a new position by the user, and re-sized.

10.4.1. Basic usage

The `<rich:popupPanel>` does not require any compulsory attributes, though certain use cases require different attributes.

10.4.2. Showing and hiding the pop-up

If `show="true"` then the pop-up panel will display when the page is first loaded.

The `<rich:popupPanel>` component can be shown and hidden manually using the `show()` and `hide()` methods from the JavaScript API. These can be implemented using two different approaches:

- Using the `<rich:componentControl>` component. For details on the component, refer to *Section 4.4, "<rich:componentControl>"*.
- Using the `rich:component` function. For details on the function, refer to *Section 14.2, "rich:component"*.

For explicit referencing when using the functions, the component can be given an `id` identifier. The component can, however, be referenced using other means, such as through a selector.

Example 10.2, "<rich:popupPanel> example" demonstrates basic use of both the `<rich:componentControl>` component and the `rich:component` function to show and hide the `<rich:popupPanel>` component.

Example 10.2. `<rich:popupPanel>` example

```
<h:commandButton value="Show the panel">
    <rich:componentControl target="popup" operation="show" />
</h:commandButton>
...
<a4j:form>
    <rich:popupPanel id="popup">

   <p><a href="#" onclick="#{rich:component('popup')}.hide()">Hide the panel</
a></p>
    </rich:popupPanel>
</a4j:form>
```

Placement

The `<rich:popupPanel>` component should usually be placed outside the original form, and include its own form if performing submissions. An exception to this is when using the `domElementAttachment` attribute, as described in *Section 10.4.4, "Size and positioning"*.

10.4.3. Modal and non-modal panels

By default, the `<rich:popupPanel>` appears as a modal window that blocks interaction with the other objects on the page. To implement a non-modal window instead, set `modal="false"`. This will allow interaction with other objects outside the pop-up panel.

10.4.4. Size and positioning

The pop-up panel can be both re-sized and re-positioned by the user. The minimum possible size for the panel can be set with the `minWith` and `minHeight` attributes. These abilities can be deactivated by setting `resizable` or `movable` to `false` as necessary.

The pop-up panel can be automatically sized when it is shown if the `autosized` attribute is set to `true`.

The `<rich:popupPanel>` component is usually rendered in front of any other objects on the page. This is achieved by attaching the component to the `<body>` element of the page, and setting a very high *"z-index"* (the stack order of the object). This approach is taken because relatively-positioned elements could still overlap the pop-up panel if they exist at higher levels of the DOM hierarchy,

even if their z-index is less than the `<rich:popupPanel>` component. However, to avoid form limitation of the pop-up panel on pages where no such elements exist, the `<rich:popupPanel>` component can be reattached to its original DOM element by setting `domElementAttachment` to either `parent` or `form`.

Embedded objects inserted into the HTML with the `<embed>` tag will typically be rendered in front of a `<rich:popupPanel>` component. The `<rich:popupPanel>` component can be forcibly rendered in front of these objects by setting `overlapEmbedObjects="true"`.

Using `overlapEmbedObjects`

Due to the additional script processing required when using the `overlapEmbedObjects` attribute, applications can suffer from decreased performance. As such, `overlapEmbedObjects` should only be set to `true` when `<embed>` tags are being used. Do not set it to `true` for applications that do not require it.

10.4.5. Contents of the pop-up

The `<rich:popupPanel>` component can contain any other rich component just like a normal panel.

Contents of the `<rich:popupPanel>` component which are positioned relatively may be trimmed if they extend beyond the borders of the pop-up panel. For certain in-line controls this behavior may be preferable, but for other dynamic controls it could be undesirable. If the `trimOverlayedElements` attribute is set to `false` then child components will not be trimmed if they extend beyond the borders of the pop-up panel.

10.4.6. Header and controls

A panel header and associated controls can be added to the `<rich:popupPanel>` component through the use of facets. The `header` facet displays a title for the panel, and the `controls` facet can be customized to allow window controls such as a button for closing the pop-up. *Example 10.3, "Header and controls"* demonstrates the use of the facets.

Example 10.3. Header and controls

```
<h:commandLink value="Show pop-up">
    <rich:componentControl target="popup" operation="show" />
</h:commandLink>
...
<a4j:form>

  <rich:popupPanel id="popup" modal="false" autosized="true" resizeable="false">
        <f:facet name="header">
```

```
            <h:outputText value="The title of the panel" />
        </f:facet>
        <f:facet name="controls">
                                                <h:graphicImage    value="/pages/
close.png" style="cursor:pointer" onclick="#{rich:component('popup')}.hide()" /
>
        </f:facet>
        <p>
            This is the content of the panel.
        </p>
    </rich:popupPanel>
</a4j:form>
```

Figure 10.5. Header and controls

10.4.7. Reference data

- *component-type*: `org.richfaces.popupPanel`
- *component-class*: `org.richfaces.component.html.HtmlpopupPanel`
- *component-family*: `org.richfaces.popupPanel`

- *renderer-type*: `org.richfaces.popupPanelRenderer`
- *tag-class*: `org.richfaces.taglib.popupPanelTag`

10.5. `<rich:collapsiblePanel>`

The `<rich:collapsiblePanel>` component is a collapsible panel that shows or hides content when the header bar is activated. It is a simplified version of `<rich:togglePanel>` component.

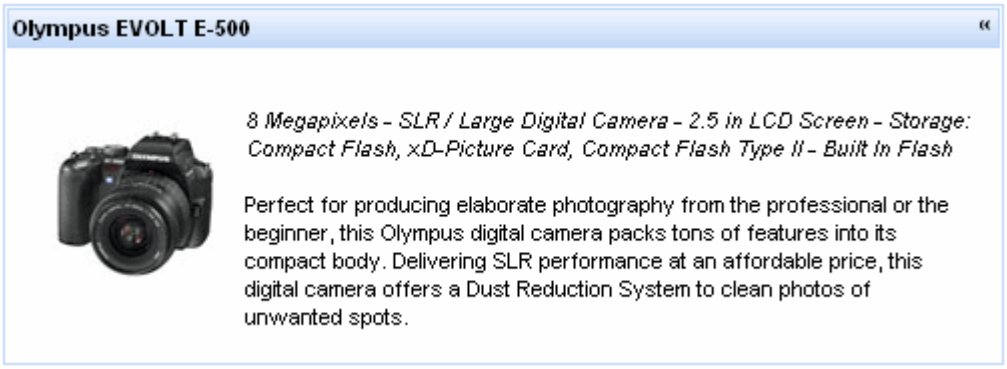

Figure 10.6. `<rich:collapsiblePanel>`

10.5.1. Basic usage

Basic usage requires the `header` attribute to be specified, which provides the title for the header element. Additionally the panel requires content to display when it is expanded. Content is added as child elements like a standard panel.

10.5.2. Expanding and collapsing the panel

The switching mode for performing submissions is determined by the `switchType` attribute, which can have one of the following three values:

`server`
: This is the default setting. The `<rich:collapsiblePanel>` component performs a common submission, completely re-rendering the page. Only one panel at a time is uploaded to the client side.

`ajax`
: The `<rich:collapsiblePanel>` component performs an Ajax form submission, and only the content of the panel is rendered. Only one panel at a time is uploaded to the client side.

`client`
: The `<rich:collapsiblePanel>` component updates on the client side, re-rendering itself and any additional components listed with the `render` attribute.

10.5.3. Appearance

The appearance of the `<rich:collapsiblePanel>` component can be customized using facets. The `headerExpanded` and `headerCollapsed` facets are used to style the appearance of the panel when it is expanded and collapsed respectively. The `expandControl` facet styles the control in the panel header used for expanding, and the `collapseControl` facet styles the control for collapsing.

10.5.4. `<rich:collapsiblePanel>` server-side events

The `<rich:collapsiblePanel>` component uses the following unique server-side events:

- The `ChangeExpandEvent` event occurs on the server side when the `<rich:collapsiblePanel>` component is expanded or collapsed through Ajax using the `server` mode. It can be processed using the `ChangeExpandListener` attribute.

10.5.5. Reference data

- *`component-type`*: `org.richfaces.collapsiblePanel`
- *`component-class`*: `org.richfaces.component.html.HtmlcollapsiblePanel`
- *`component-family`*: `org.richfaces.collapsiblePanel`
- *`renderer-type`*: `org.richfaces.collapsiblePanelRenderer`
- *`tag-class`*: `org.richfaces.taglib.collapsiblePanelTag`

10.6. `<rich:tab>`

The `<rich:tab>` component represents an individual tab inside a `<rich:tabPanel>` component, including the tab's content. Clicking on the tab header will bring its corresponding content to the front of other tabs. Refer to *Section 10.7, "<rich:tabPanel>"* for details on the `<rich:tabPanel>` component.

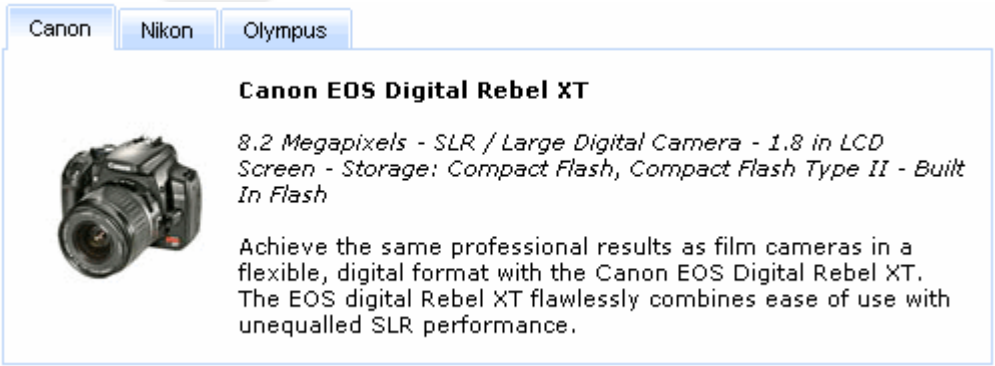

Figure 10.7. A `<rich:tabPanel>` component containing three `<rich:tab>` components

10.6.1. Basic usage

Basic usage of the `<rich:tab>` component requires the `name` attribute to uniquely identify the tab within the parent `<rich:tabPanel>` component. As the tabs are switched, the `name` identifier of the currently selected tab is stored in the `activeItem` attribute of the parent `<rich:tabPanel>` component.

10.6.2. Header labeling

In addition to the `name` identifier, the `header` attribute must be defined. The `header` attribute provides the text on the tab header. The content of the tab is then detailed inside the `<rich:tab>` tags.

Alternatively, the `header` facet could be used in place of the `header` attribute. This would allow for additional styles and custom content to be applied to the tab. The component also supports three facets to customize the appearance depending on the current state of the tab:

`headerActive` facet
: This facet is used when the tab is the currently active tab.

`headerInactive` facet
: This facet is used when the tab is not currently active.

`headerDisabled` facet
: This facet is used when the tab is disabled.

The `header` facet is used in place of any state-based facet that has not been defined.

10.6.3. Switching tabs

The switching mode for performing submissions can be inherited from the `switchType` attribute of the parent `<rich:tabPanel>` component, or set individually for each `<rich:tab>` component. Refer to *Section 10.7, “<rich:tabPanel>”* for details on the `switchType` attribute.

An individual tab can be disabled by setting `disabled="true"`. Disabled tabs cannot be activated or switched to.

10.6.4. `<rich:tab>` client-side events

In addition to the standard HTML events, the `<rich:tab>` component uses the client-side events common to all switchable panel items:

- The `onenter` event points to the function to perform when the mouse enters the tab.
- The `onleave` attribute points to the function to perform when the mouse leaves the tab.

10.6.5. Reference data

- *component-type*: org.richfaces.tab
- *component-class*: org.richfaces.component.html.HtmlTab
- *component-family*: org.richfaces.tab
- *renderer-type*: org.richfaces.tabRenderer
- *tag-class*: org.richfaces.taglib.tabTag

10.7. <rich:tabPanel>

The <rich:tabPanel> component provides a set of tabbed panels for displaying one panel of content at a time. The tabs can be highly customized and themed. Each tab within a <rich:tabPanel> container is a <rich:tab> component. Refer to *Section 10.6, "<rich:tab>"* for further details on the <rich:tab> component.

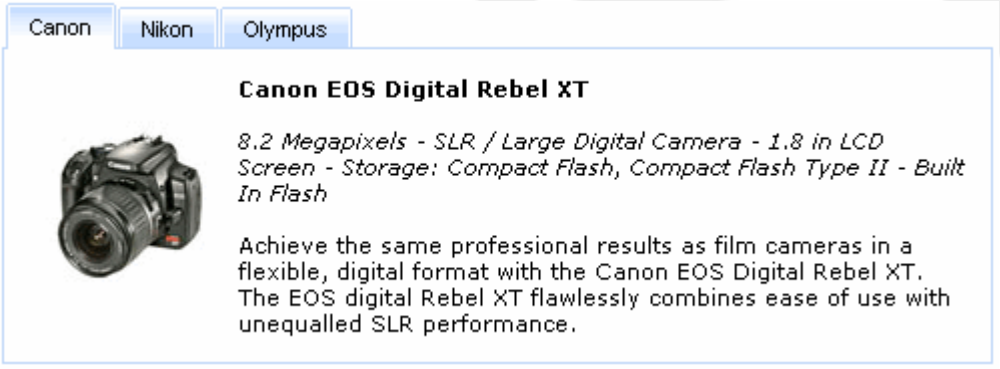

Figure 10.8. A <rich:tabPanel> component containing three <rich:tab> components

Form elements required

All <rich:tabPanel> components should be wrapped in a form element so that the contents of the tab are processed correctly during a tab change in either ajax or server mode.

Alternatively, the contents of a <rich:tab> component within the <rich:tabPanel> component could be wrapped in a form element, such that they will be processed using the inner submitting component only. In this case, the <rich:tabPanel> component will automatically add form tags around the tab's contents, and the contents will not be processed during switching.

10.7.1. Switching panels

The `activeItem` attribute holds the active tab name. This name is a reference to the `name` identifier of the active child `<rich:tab>` component.

The switching mode for performing submissions is determined by the `switchType` attribute, which can have one of the following three values:

`server`
: The default setting. Activation of a `<rich:tab>` component causes the parent `<rich:tabPanel>` component to perform a common submission, completely re-rendering the page. Only one tab at a time is uploaded to the client side.

`ajax`
: Activation of a `<rich:tab>` component causes the parent `<rich:tabPanel>` component to perform an Ajax form submission, and the content of the tab is rendered. Only one tab at a time is uploaded to the client side.

`client`
: Activation of a `<rich:tab>` component causes the parent `<rich:tabPanel>` component to update on the client side. JavaScript changes the styles such that one tab becomes hidden while the other is shown.

10.7.2. `<rich:tabPanel>` client-side events

In addition to the standard Ajax events and HTML events, the `<rich:tabPanel>` component uses the client-side events common to all switchable panels:

- The `onitemchange` event points to the function to perform when the switchable item is changed.
- The `onbeforeitemchange` event points to the function to perform when before the switchable item is changed.

10.7.3. `<rich:tabPanel>` server-side events

The `<rich:tabPanel>` component uses the server-side events common to all switchable panels:

- The `ItemChangeEvent` event occurs on the server side when an item is changed through Ajax using the `server` mode. It can be processed using the `ItemChangeListener` attribute.

10.7.4. Reference data

- *`component-type`*: `org.richfaces.tabPanel`
- *`component-class`*: `org.richfaces.component.html.HtmlTabPanel`
- *`component-family`*: `org.richfaces.tabPanel`

- *renderer-type*: `org.richfaces.tabPanelRenderer`
- *tag-class*: `org.richfaces.taglib.tabPanelTag`

10.8. `<rich:toggleControl>`

The `<rich:toggleControl>` behavior can be attached to any interface component. It works with a `<rich:togglePanel>` component to switch between different `<rich:togglePanelItem>` components.

Refer to *Section 10.9, "<rich:togglePanel>"* and *Section 10.10, "<rich:togglePanelItem>"* for details on how to use the components together.

10.8.1. Basic usage

The `<rich:toggleControl>` can be used to switch through `<rich:togglePanelItem>` components in a `<rich:togglePanel>` container. If the `<rich:toggleControl>` component is positioned inside a `<rich:togglePanel>` component, no attributes need to be defined, as the control is assumed to switch through the `<rich:togglePanelItem>` components of its parent.

A `<rich:toggleControl>` component can be located outside the `<rich:togglePanel>` component it needs to switch. Where this is the case, the `<rich:togglePanel>` is identified using the `activePanel` attribute. the Cycling through components requires the `for` attribute, which points to the `id` identifier of the `<rich:togglePanel>` that it controls.

10.8.2. Specifying the next state

The `<rich:toggleControl>` component will cycle through `<rich:togglePanelItem>` components in the order they are defined within the view. However, the next item to switch to can be explicitly defined by including a `<rich:toggleControl>` component within a `<rich:togglePanelItem>` and using the `targetItem` attribute. The `targetItem` attribute points to the `<rich:togglePanelItem>` to switch to when the state is next changed. *Example 10.4, "<rich:toggleControl> example"* demonstrates how to specify the next switchable state in this way.

Example 10.4. `<rich:toggleControl>` example

```
<rich:togglePanel id="layout" activeItem="short">
     <rich:togglePanelItem id="short">
          //content
          <h:commandButton>
           <rich:toggleControl targetItem="details"> // switches to details state
          </h:commandButton>
     </rich:togglePanelItem>
     <rich:togglePanelItem id="details">
          //content
          <h:commandButton>
              <rich:toggleControl targetItem="short"> //switches to short state
```

```
          </h:commandButton>
     <rich:togglePanelItem>
</rich:togglePanel>
<h:commandButton>
     <rich:toggleControl activePanel="layout"/> // cycles through the states
</h:commandButton>
```

10.8.3. Reference data

- *component-type*: `org.richfaces.ToggleControl`
- *component-class*: `org.richfaces.component.html.HtmlToggleControl`
- *component-family*: `org.richfaces.ToggleControl`
- *renderer-type*: `org.richfaces.ToggleControlRenderer`
- *tag-class*: `org.richfaces.taglib.ToggleControlTag`

10.9. `<rich:togglePanel>`

The `<rich:togglePanel>` component is a wrapper for multiple `<rich:togglePanelItem>` components. Each child component is displayed after being activated with the `<rich:toggleControl>` behavior.

Refer to *Section 10.8, "<rich:toggleControl>"* and *Section 10.9, "<rich:togglePanel>"* for details on how to use the components together.

The `<rich:togglePanel>` component is used as a base for the other switchable components, the `<rich:accordion>` component and the `<rich:tabPanel>` component. It provides an abstract switchable component without any associated markup. As such, the `<rich:togglePanel>` component could be customized to provide a switchable component when neither an accordion component or a tab panel component is appropriate.

10.9.1. Basic usage

The initial state of the component can be configured using the `activeItem` attribute, which points to a child component to display. Alternatively, if no `activeItem` attribute is defined, the initial state will be blank until the user activates a child component using the `<rich:toggleControl>` component.

The child components are shown in the order in which they are defined in the view.

10.9.2. Toggling between components

The switching mode for performing submissions is determined by the `switchType` attribute, which can have one of the following three values:

`server`
: The default setting. Activation of a child component causes the parent `<rich:togglePanel>` component to perform a common submission, completely re-rendering the page. Only one child at a time is uploaded to the client side.

`ajax`
: Activation of a child component causes the parent `<rich:togglePanel>` component to perform an Ajax form submission, and the content of the child is rendered. Only one child at a time is uploaded to the client side.

`client`
: Activation of a child component causes the parent `<rich:togglePanel>` component to update on the client side. JavaScript changes the styles such that one child component becomes hidden while the other is shown.

10.9.3. Reference data

- *component-type*: `org.richfaces.TogglePanel`
- *component-class*: `org.richfaces.component.html.HtmlTogglePanel`
- *component-family*: `org.richfaces.TogglePanel`
- *renderer-type*: `org.richfaces.TogglePanelRenderer`
- *tag-class*: `org.richfaces.taglib.TogglePanelTag`

10.10. `<rich:togglePanelItem>`

The `<rich:togglePanelItem>` component is a switchable panel for use with the `<rich:togglePanel>` component. Switching between `<rich:togglePanelItem>` components is handled by the `<rich:toggleControl>` behavior.

Refer to *Section 10.8, "<rich:toggleControl>"* and *Section 10.9, "<rich:togglePanel>"* for details on how to use the components together.

10.10.1. Reference data

- *component-type*: `org.richfaces.TogglePanelItem`
- *component-class*: `org.richfaces.component.html.HtmlTogglePanelItem`
- *component-family*: `org.richfaces.TogglePanelItem`
- *renderer-type*: `org.richfaces.TogglePanelItemRenderer`
- *tag-class*: `org.richfaces.taglib.TogglePanelItemTag`

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.

11.1. `<a4j:repeat>`

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 11, Tables and grids*.

11.1.1. Basic usage

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. *Example 11.1, "<a4j:repeat> example"* shows how to use `<a4j:repeat>` to maintain a simple table.

Example 11.1. `<a4j:repeat>` example

```
<table>
    <tbody>
        <a4j:repeat value="#{repeatBean.items}" var="item">
            <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.

11.1.2. Limited views and partial updates

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.

Specific cells, rows, and columns can be updated without sending Ajax requests for the entire collection. Components that cause the change can specify which part of the table to update through the `render` attribute. The `render` attribute specifies which part of a table to update:

`render=`*`cellId`*
: Update the cell with an identifier of *`cellId`* within the row that contains the current component.

  Instead of a specific identifier, the *`cellId`* reference could be a variable: `render=#{`*`bean.cellToUpdate`*`}`.

`render=`*`tableId`*`:`*`rowId`*
: Update the row with an identifier of *`rowId`* within the table with an identifier of *`tableId`*. Alternatively, if the current component is contained within the table, use `render=`*`rowId`*.

  Instead of a specific identifier, the *`tableId`* of *`rowId`* references could be variables: `render=`*`tableId`*`:#{`*`bean.rowToUpdate`*`}`.

`render=`*`tableId`*`:`*`rowId`*`:`*`cellId`*
: Update the cell with an identifier of *`cellId`*, within the row with and identifier of *`rowId`*, within the table with an identifier of *`tableId`*.

  Instead of a specific identifier, any of the references could be variables: `render=`*`tableId`*`:#{`*`bean.rowToUpdate`*`}:`*`cellId`*.

Alternatively, keywords can be used with the `render` attribute:

`render=@column`
: Update the column that contains the current component.

`render=@row`
: Update the row that contains the current component.

`render=`*`tableId`*`:@body`
: Update the body of the table with the identifier of *`tableId`*. Alternatively, if the current component is contained within the table, use `render=@body` instead.

`render=`*`tableId`*`:@header`
: Update the header of the table with the identifier of *`tableId`*. Alternatively, if the current component is contained within the table, use `render=@header` instead.

`render=tableId:@footer`

Update the footer of the table with the identifier of *`tableId`*. Alternatively, if the current component is contained within the table, use `render=@footer` instead.

11.1.3. Reference data

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

11.2. `<rich:column>`

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

11.2.1. Basic usage

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 11.2, "Basic column example"*.

Example 11.2. 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 11.1. Basic column example

11.2.2. Spanning columns

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 11.3, "Column spanning example"*.

Example 11.3. 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 11.2. Column spanning example

11.2.3. Spanning rows

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 11.4, "Row spanning example"* and the resulting *Figure 11.4, "Complex headers using column groups"* show the first column of the table spanning three rows.

Example 11.4. 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>
```

Figure 11.3. Row spanning example

For details on filtering and sorting columns, refer to *Section 11.8, "Table filtering"* and *Section 11.9, "Table sorting"*.

11.2.4. Reference data

- *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`

11.3. `<rich:columnGroup>`

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.

11.3.1. Complex headers

The `<rich:columnGroup>` can also be used to create complex headers in a table. *Example 11.5, "Complex headers using column groups"* and the resulting *Figure 11.4, "Complex headers using column groups"* demonstrate how complex headers can be achieved.

Example 11.5. 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 11.4. Complex headers using column groups

11.3.2. Reference data

- *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`

11.4. `<rich:dataGrid>`

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.

The `<rich:dataGrid>` component is similar in function to the JavaServer Faces `<h:panelGrid>` component. However, the `<rich:dataGrid>` component additionally allows iteration through the data model rather than just aligning child components in a grid layout.

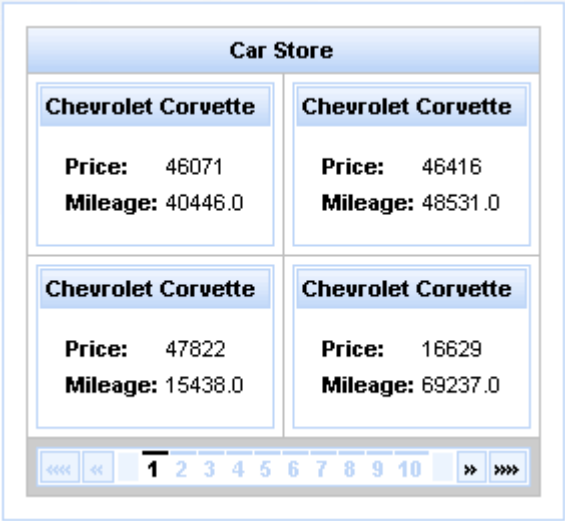

Figure 11.5. The `<rich:dataGrid>` component

11.4.1. Basic usage

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.

11.4.2. Customizing the grid

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 11.6. `<rich:dataGrid>` example

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

  <rich:dataGrid value="#{dataTableScrollerBean.allCars}" var="car" columns="2" elements="4" fi
        <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>
```

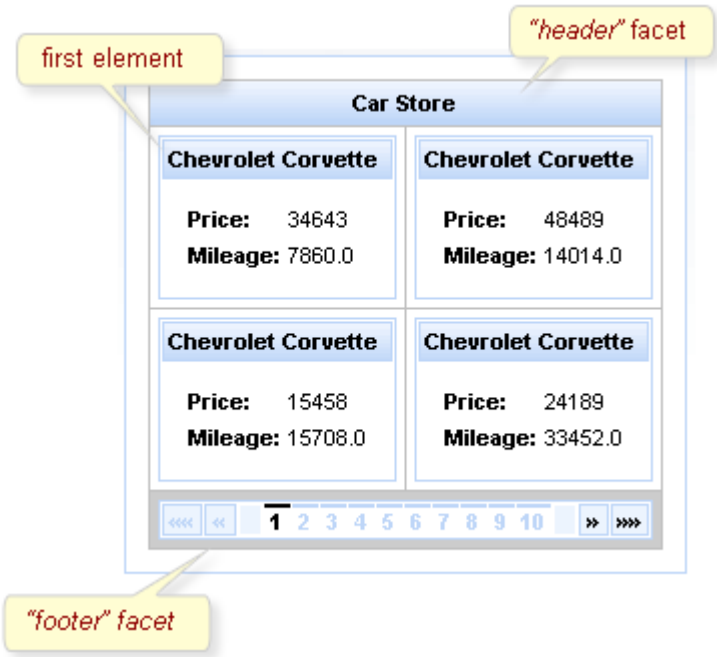

Figure 11.6. `<rich:dataGrid>` example

11.4.3. Patial updates

As `<rich:dataGrid>` the component is based on the `<a4j:repeat>` component, it can be partially updated with Ajax. Refer to *Section 11.1.2, "Limited views and partial updates"* for details on partially updating the `<rich:dataGrid>` component.

11.4.4. Reference data

- *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`

11.5. <rich:dataTable>

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> and <rich:columnGroup> 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 11.6, "<rich:extendedDataTable>"* for further details.

11.5.1. Basic usage

The value attribute points to the data model, and the var attribute specifies a variable to use when iterating through the data model.

11.5.2. Customizing the table

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 11.7. <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 11.7. `<rich:dataTable>` example

For details on filtering and sorting data tables, refer to *Section 11.8, "Table filtering"* and *Section 11.9, "Table sorting"*.

11.5.3. Partial updates

As `<rich:dataTable>` the component is based on the `<a4j:repeat>` component, it can be partially updated with Ajax. Refer to *Section 11.1.2, "Limited views and partial updates"* for details on partially updating the `<rich:dataTable>` component.

11.5.4. Reference data

- *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

11.6. <rich:extendedDataTable>

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:

frozenColumns	onselectionchange	selectionMode
height	selectedClass	tableState
noDataLabel	selection	

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

- columns
- columnsWidth

11.6.1. Basic usage

Basic use of the <rich:extendedDataTable> component requires the value and var attributes, the same as with the <rich:dataTable> component. Refer to *Section 11.5, "<rich:dataTable>"* for details.

11.6.2. Table appearance

The height attribute defines the height of the table on the page. This is set to 100% 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.

11.6.3. Extended features

Example 11.8. <rich:extendedDataTable> example

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

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

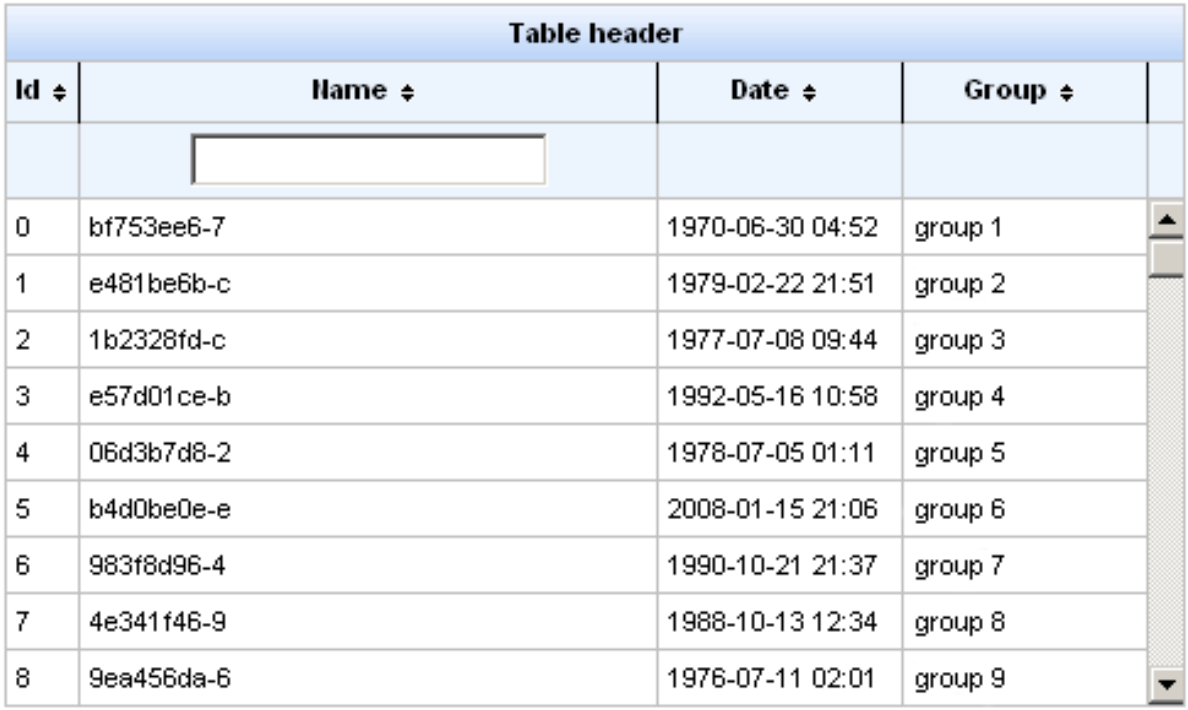

Figure 11.8. `<rich:extendedDataTable>` example

Example 11.8, "<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.

11.6.3.1. Row selection

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 11.9, "Selecting multiple rows"* shows the table from the example with multiple rows selected.

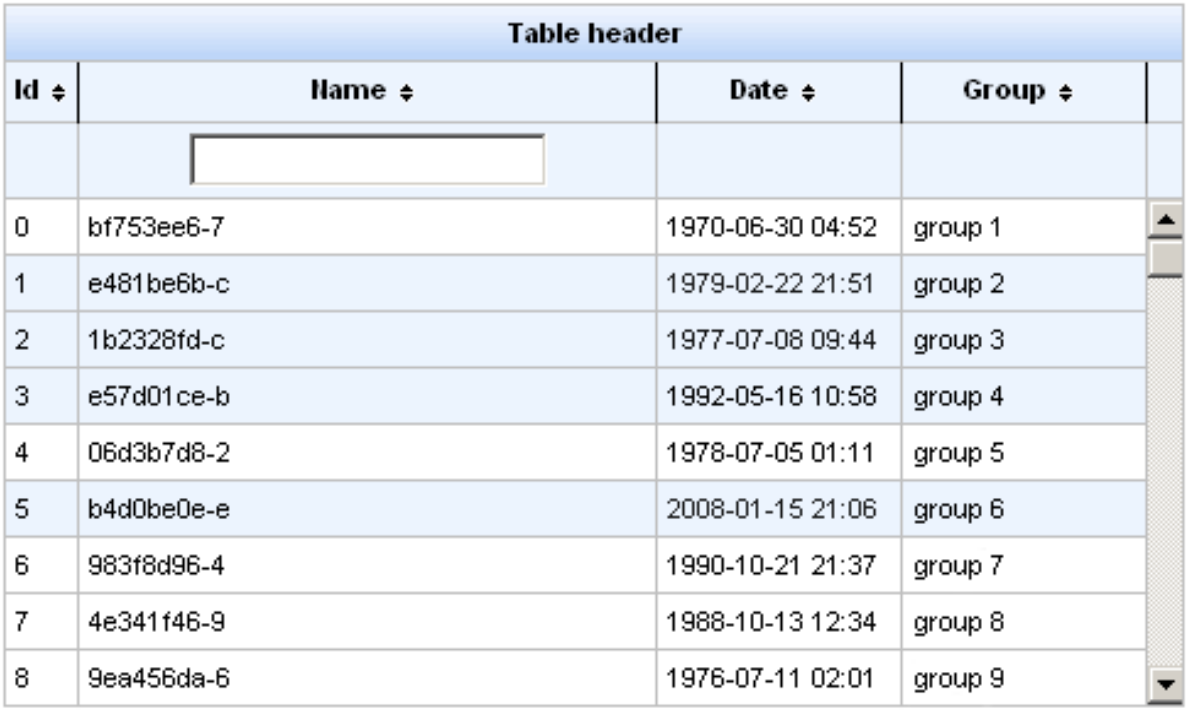

Table header			
Id	Name	Date	Group
0	bf753ee6-7	1970-06-30 04:52	group 1
1	e481be6b-c	1979-02-22 21:51	group 2
2	1b2328fd-c	1977-07-08 09:44	group 3
3	e57d01ce-b	1992-05-16 10:58	group 4
4	06d3b7d8-2	1978-07-05 01:11	group 5
5	b4d0be0e-e	2008-01-15 21:06	group 6
6	983f8d96-4	1990-10-21 21:37	group 7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

Figure 11.9. Selecting multiple rows

11.6.3.2. Filtering

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 11.8, "Table filtering"*.

11.6.3.3. Sorting

Each column can be used to sort the contents of the table. The value of the data model to sort by is specified with the `sortBy` attribute. 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 11.9, "Table sorting"*.

11.6.3.4. Rearranging 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

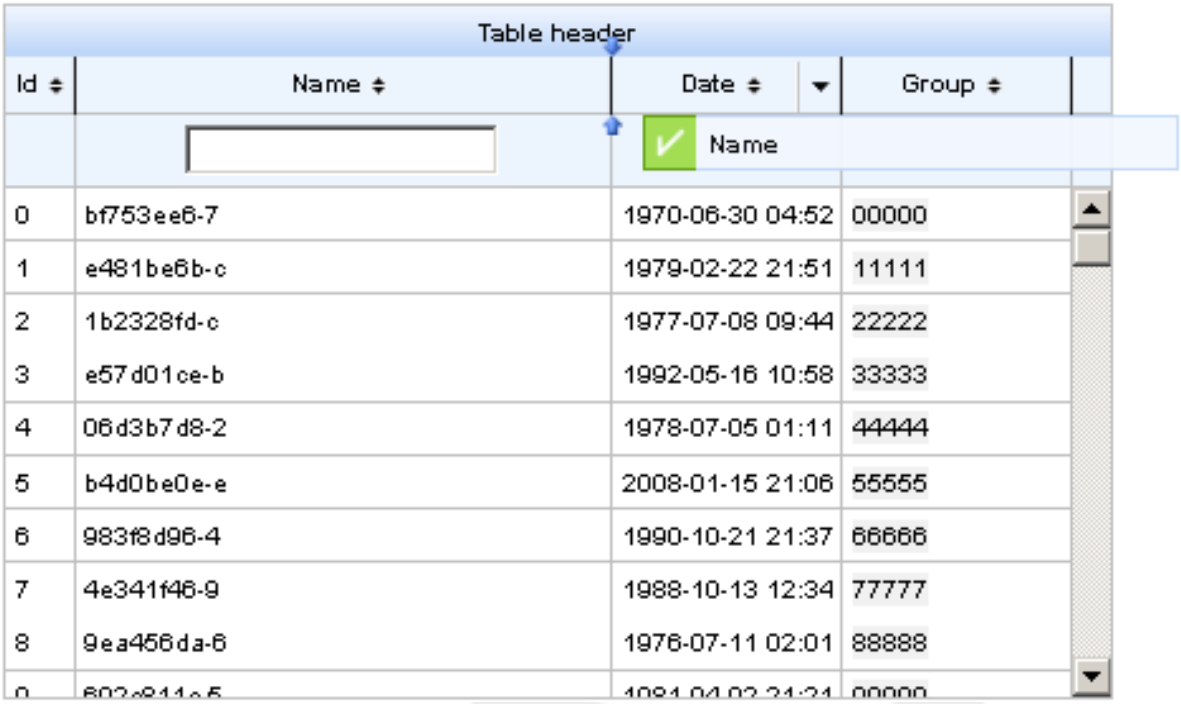

Figure 11.10. Dragging columns

11.6.3.5. Saving the state

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.

11.6.4. Reference data

- *`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`

11.7. `<rich:list>`

The `<rich:list>` component renders a list of items. The list can be an numerically ordered list, an unordered bullet-point list, or a data definition list. The component uses a data model for managing the list items, which can be updated dynamically.

11.7.1. Basic usage

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).

11.7.2. Type of list

By default, the list is displayed as an unordered bullet-point list. The `type` attribute is used to specify different list types:

`unordered`
: The default presentation. The list is presented as a series of bullet-points, similar to the `<ul>` HTML element.

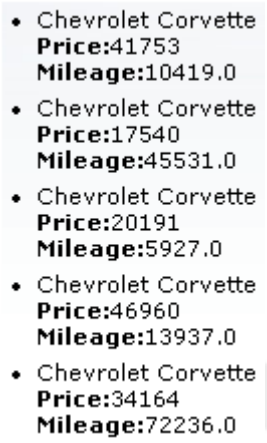

Figure 11.11. Unordered list

`ordered`
: The list is presented as a numbered series of items, similar to the `<ol>` HTML element.

Figure 11.12. Ordered list

`definitions`

The list is presented as a series of data definitions. Part of the data model, specified as the term, is listed prominently. The other associated data is listed after each term.

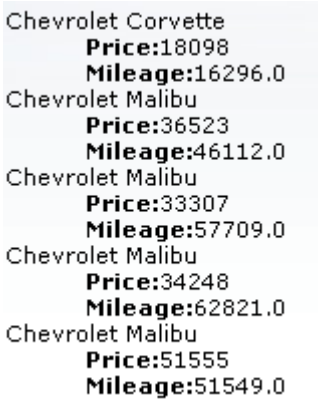

Figure 11.13. Data definition list

The term is marked using the `term` facet. The facet is required for all definition lists. Use of the facet is shown in *Example 11.9, "Data definition list"*.

Example 11.9. Data definition list

```
<h:form>

  <rich:list var="car" value="#{dataTableScrollerBean.allCars}" type="definitions" rows="5"
        <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:list>
</h:form>
```

11.7.3. Bullet and numeration appearance

The appearance of bullet points for unordered lists or numeration for ordered lists can be customized through CSS, using the list-style-type property.

11.7.4. Customizing the list

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 11.10, "<rich:list> example"* shows a simple example using the `<rich:list>` component.

Example 11.10. `<rich:list>` example

```
<h:form>

<rich:list var="car" value="#{dataTableScrollerBean.allCars}" rows="5" type="unordered" 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:list>
</h:form>
```

- Chevrolet Corvette
  Price:41753
  Mileage:10419.0
- Chevrolet Corvette
  Price:17540
  Mileage:45531.0
- Chevrolet Corvette
  Price:20191
  Mileage:5927.0
- Chevrolet Corvette
  Price:46960
  Mileage:13937.0
- Chevrolet Corvette
  Price:34164
  Mileage:72236.0

Figure 11.14. `<rich:list>` example

11.7.5. Reference data

- *component-type*: `org.richfaces.List`
- *component-class*: `org.richfaces.component.html.HtmlList`
- *component-family*: `org.richfaces.List`
- *renderer-type*: `org.richfaces.ListRenderer`
- *tag-class*: `org.richfaces.taglib.ListTag`

11.8. Table filtering

Under development.

11.9. Table sorting

Under development.

Output and messages

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.

Read this chapter for details on components that display messages and other feedback to the user.

12.1. `<rich:progressBar>`

The `<rich:progressBar>` component displays a progress bar to indicate the status of a process to the user. It can update either through Ajax or on the client side, and the look and feel can be fully customized.

Figure 12.1. `<rich:progressBar>`

12.1.1. Basic usage

Basic usage of the `<rich:progressBar>` component requires only the `value` attribute, which points to the method that provides the current progress.

Example 12.1. Basic usage

```
<rich:progressBar value="#{bean.incValue}" />
```

12.1.2. Customizing the appearance

By default, the minimum value of the progress bar is `0` and the maximum value of the progress bar is `100`. These values can be customized using the `minValue` and `maxValue` attributes respectively.

The progress bar can be labeled in one of two ways:

Using the `label` attribute
: The content of the `label` attribute is displayed over the progress bar.

Example 12.2. Using the `label` attribute

```
<rich:progressBarvalue="#{bean.incValue}"id="progrs"label="#{bean.incValue}"/
>
```

Using nested child components

Child components, such as the JSF `<h:outputText>` component, can be nested in the `<rich:progressBar>` component to display over the progress bar.

Example 12.3. Using nested child components

```
<rich:progressBar value="#{bean.incValue}">
    <h:outputText value="#{bean.incValue} %"/>
</rich:progressBar>
```

Macro-substitution

The following section details the use of macro-substitution paramaters in labeling. Macro-substitution may be revised and altered in future versions of RichFaces. Be aware of this when using macro-substitution in your applications.

For labeling, the `<rich:progressBar>` component recognizes three macro-substitution parameters:

`{value}`
: The current progress value.

`{minValue}`
: The minimum value for the progress bar.

`{maxValue}`
: The maximum value for the progress bar.

Example 12.4. Using macro-substitution for labeling

```
<rich:progressBar value="#{bean.incValue1}" minValue="400" maxValue="900">
    <h:outputText value="Minimum value is {minValue}, current value is {value},
 maximum value is {maxValue}"/>
</rich:progressBar>
```

Additionally, you can use the `{param}` parameter to specify any custom parameters you require. Define the parameters in the bean for the progress method, then reference it with the

`<rich:progressBar>` component's `parameters` attribute, as shown in *Example 12.5, "Using the param parameter"*.

Example 12.5. Using the `param` parameter

```
<rich:progressBar value="#{bean.incValue}" parameters="param:'#{bean.dwnlSpeed}'">
    <h:outputText value="download speed {param} KB/s"/>
</rich:progressBar>
```

To define customized initial and complete states for the progress bar, use the `initial` and `complete` facets. The `initial` facet displays when the progress value is less than or equal to the minimum value, and the `complete` facet displays when the progress value is greater than or equal to the maximum value.

Example 12.6. Initial and complete states

```
<rich:progressBar value="#{bean.incValue1}">
    <f:facet name="initial">
        <h:outputText value="Process not started"/>
    </f:facet>
    <f:facet name="complete">
        <h:outputText value="Process completed"/>
    </f:facet>
</rich:progressBar>
```

12.1.3. Using set intervals

The `<rich:progressBar>` component can be set to constantly poll for updates at a constant interval. Use the `interval` component to set the interval in milliseconds. The progress bar is updated whenever the polled value changes. Polling is only active when the `enabled` attribute is set to `true`.

Example 12.7. Using set intervals

```
<rich:progressBar value="#{bean.incValue1}" id="progress" interval="400" enabled="#{bean.enabled1}"/
>
```

12.1.4. Update mode

The mode for updating the progress bar is determined by the `mode` attribute, which can have one of the following values:

`ajax`

The progress bar updates in the same way as the `<a4j:poll>` component. The `<rich:progressBar>` component repeatedly polls the server for the current progress value.

`client`

The progress bar updates on the client side, set using the JavaScript API.

Layout and appearance

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.

Read this chapter to alter the layout and appearance of web applications using special components.

13.1. `<rich:jQuery>`

The `<rich:jQuery>` component applies styles and custom behavior to both JSF (JavaServer Faces) objects and regular DOM (Document Object Model) objects. It uses the jQuery JavaScript framework to add functionality to web applications.

13.1.1. Basic usage

The query triggered by the `<rich:jQuery>` component is specified using the `query` attribute.

With the query defined, the component is used to trigger the query as either a *timed query* or a *named query*. The query can be bound to an event to act as an *event handler*. These different approaches are covered in the following sections.

13.1.2. Defining a selector

Any objects or lists of objects used in the query are specified using the `selector` attribute. The `selector` attribute references objects using the following method:

- The `selector` attribute can refer to the `id` identifier of any JSF component or client.
- If the `selector` attribute does not match the `id` identifier attribute of any JSF components or clients on the page, it instead uses syntax defined by the World Wide Web Consortium (W3C) for the CSS rule selector. Refer to the syntax specification at *http://api.jquery.com/category/selectors/* for full details.

Because the `selector` attribute can be either an `id` identifier attribute or CSS selector syntax, conflicting values could arise. *Example 13.1, "Avoiding syntax confusion"* demonstrates how to use double backslashes to escape colon characters in `id` identifier values.

Example 13.1. Avoiding syntax confusion

```
<h:form id="form">
```

```
    <h:panelGrid id="menu">
        <h:graphicImage value="pic1.jpg" />
        <h:graphicImage value="pic2.jpg" />
    </h:panelGrid>
</h:form>
```

The `id` identifier for the `<h:panelGrid>` element is `form:menu`, which can conflict with CSS selector syntax. Double backslashes can be used to escape the colon character such that the identifier is read correctly instead of being interpreted as CSS selector syntax.

```
<rich:jQuery selector="#form\\:menu img" query="..." />
```

13.1.3. Event handlers

Queries set as event handlers are triggered when the component specified in the `selector` attribute raises an event. The query is bound to the event defined using the `event` attribute.

Use the `attachType` attribute to specify how the event-handling queries are attached to the events:

`bind`
: This is the default for attaching queries to events. The event handler is bound to all elements currently defined by the `selector` attribute.

`live`
: The event handler is bound to all current and future elements defined by the `selector` attribute.

`one`
: The event handler is bound to all elements currently defined by the `selector` attribute. After the first invocation of the event, the event handler is unbound such that it no longer fires when the event is raised.

13.1.4. Timed queries

Timed queries are triggered at specified times. This can be useful for calling simple methods when a page is rendered, or for adding specific functionality to an element. Use the `timing` attribute to specify the point at which the timed query is triggered:

`ondomready`
: This is the default behavior. The query is triggered when the document is loaded and the DOM is ready. The query is called as a `jQuery()` function.

`immediate`
: The query is triggered immediately. The query is called as an in-line script.

Example 13.2. `<rich:jQuery>` example

```
<rich:dataTable id="customList" ... >
    ...
</rich:dataTable>

<rich:jQuery                                                    selector="#customList
 tr:odd" timing="ondomready" query="addClass(odd)" />
```

In the example, the selector picks out the odd `<tr>` elements that are children of the element with an `id="customlist"` attribute. The query `addClass(odd)` is then performed on the selection during page loading (`onload`) such that the `odd` CSS class is added to the selected elements.

Make	Model	Price	Mileage
Chevrolet	Corvette	39858	64699.0
Chevrolet	Corvette	38091	38014.0
Chevrolet	Corvette	18427	64568.0
Chevrolet	Corvette	35277	79994.0
Chevrolet	Corvette	47206	19290.0
Chevrolet	Malibu	52155	5242.0
Chevrolet	Malibu	41576	73266.0
Chevrolet	Malibu	41762	16542.0

13.1.5. Named queries

Named queries are given a name such that they can be triggered by other functions or handlers. Use the `name` attribute to name the query. The query can then be accessed as though it were a JavaScript function using the specified `name` attribute as the function name.

Calls to the function must pass a direct reference (`this`) to the calling object as a parameter. This is treated the same as an item defined through the `selector` attribute.

If the function requires extra parameters itself, these are provided in JavaScript Object Notation (JSON) syntax as a second parameter in the JavaScript call. The `options` namespace is then used in the `<rich:jQuery>` query to access the passed function parameters. *Example 13.3, "Calling a <rich:jQuery> component as a function"* demonstrates the use of the `name` attribute and how to pass function parameters through the JavaScript calls.

Example 13.3. Calling a `<rich:jQuery>` component as a function

```
<h:graphicImage                    width="50"                    value="/images/
price.png"onmouseover="enlargePic(this,{width:'60px'})"onmouseout="releasePic(this)"
>
```

```
<h:graphicImage                    width="50"                    value="/images/
discount.png"                    onmouseover="enlargePic(this,{pwidth:'100px'})"                    onmouseout="releasePic(this)"
>
...
<rich:jQuery name="enlargePic" query="animate({width:options.pwidth})" />
<rich:jQuery name="releasePic" query="animate({width:'50px'})"/>
```

The example enlarges the images when the mouse moves over them. The `enlargePic` and `releasePic` components are called like ordinary JavaScript functions from the image elements.

13.1.6. Dynamic rendering

The `<rich:jQuery>` component applies style and behavioral changes to DOM objects dynamically. As such, changes applied during an Ajax response are overwritten, and will need to be re-applied once the Ajax response is complete.

Any timed queries with the `timing` attribute set to `ondomready` may not update during an Ajax response, as the DOM document is not completely reloaded. To ensure the query is re-applied after an Ajax response, include the `name` attribute in the `<rich:jQuery>` component and invoke it using JavaScript from the `oncomplete` event attribute of the component that triggered the Ajax interaction.

13.1.7. Reference data

- *`component-type`*: `org.richfaces.JQuery`
- *`component-class`*: `org.richfaces.component.html.HtmlJQuery`
- *`component-family`*: `org.richfaces.JQuery`
- *`renderer-type`*: `org.richfaces.JQueryRenderer`
- *`tag-class`*: `org.richfaces.taglib.JQueryTag`

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.

14.1. `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.

14.2. `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.

14.3. `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.

14.4. `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 14.1. `rich:findComponent` example

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

14.5. `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 14.2. `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')}" />
```