Component Reference

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

by Sean Rogers (Red Hat)

1. Introduction 1
1.1. Libraries 1
2. Common Ajax attributes 3
2.1. Rendering 3
2.1.1. render 3
2.1.2. ajaxRendered 4
2.1.3. limitRender 4
2.2. Queuing and traffic control 5
2.2.1. queue 5
2.2.2. requestDelay 5
2.2.3. ignoreDupResponses 5
2.3. Data processing 5
2.3.1. execute 5
2.3.2. immediate 6
2.3.3. bypassUpdates 6
2.4. Action and navigation 6
2.4.1. action 6
2.4.2. actionListener 6
2.5. Events and JavaScript interactions 6
2.5.1. onsubmit 6
2.5.2. onbegin 6
2.5.3. onclick 7
2.5.4. onsuccess 7
2.5.5. oncomplete 7
2.5.6. onerror 7
2.5.7. data 7
3. Common features 9
3.1. Positioning and appearance of components 9
3.2. Calling available JavaScript methods 9
I. a4j tag library 11
4. Actions 13
4.1. <a4j:actionParam> 13
4.2. <a4j:commandButton> 14
4.3. <a4j:commandLink> 15
4.4. <rich:componentControl> 15
4.5. <a4j:hashParam> 17
4.6. <a4j:jsFunction> 17
4.7. <a4j:poll> 18
4.8. <a4j:push> 19
4.9. <a4j:repeat> 19
4.10. <a4j:ajax> 20
5. Resources 23
5.1. <a4j:mediaOutput> 23
6. Containers 27

6.1. <a4j:outputPanel> 27
7. Processing management 29
7.1. <a4j:queue> 29
7.2. <a4j:log> 30
7.3. <a4j:status> 30
II. rich tag library 33
8. Tables and grids 35
8.1. <rich:column> 35
8.2. <rich:columnGroup> 38
8.3. <rich:dataScroller> 40
8.4. <rich:dataTable> 40
8.5. <rich:extendedDataTable> 40
8.6. <rich:subTable> 40
8.7. Table filtering 40
8.8. Table sorting 40
9. Functions 41
9.1. rich:clientId 41
9.2. rich:component 41
9.3. rich:element 41
9.4. rich:findComponent 41
9.5. rich:isUserInRole 41
A. Revision History 43

Introduction

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

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

1.1. Libraries

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

Ajax support

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

Common Ajax attributes

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

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

2.1. Rendering

2.1.1. `render`

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

Example 2.1. render example

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

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

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

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

`rendered` attributes

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

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

2.1.2. `ajaxRendered`

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

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

2.1.3. `limitRender`

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

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

Example 2.2. Rendering example

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

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

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

2.2. Queuing and traffic control

2.2.1. `queue`

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

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

2.2.2. `requestDelay`

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

2.2.3. `ignoreDupResponses`

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

2.3. Data processing

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

2.3.1. `execute`

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

2.3.2. `immediate`

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

2.3.3. `bypassUpdates`

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

2.4. Action and navigation

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

2.4.1. `action`

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

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

2.4.2. `actionListener`

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

2.5. Events and JavaScript interactions

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

2.5.1. `onsubmit`

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

2.5.2. `onbegin`

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

2.5.3. `onclick`

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

2.5.4. `onsuccess`

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

2.5.5. `oncomplete`

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

Reference consistency

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

2.5.6. `onerror`

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

2.5.7. `data`

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

Example 2.3. Data reference example

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

Common features

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

3.1. Positioning and appearance of components

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

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

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

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

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

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

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

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

3.2. Calling available JavaScript methods

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

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

Part I. a4j tag library

Actions

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

4.1. `<a4j:actionParam>`

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

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

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

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

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

Example 4.1. `<a4j:actionParam>` example

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

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

   public String getName() {
      return name;
```

```
    }

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

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

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

Validation failure

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

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

4.2. `<a4j:commandButton>`

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

Figure 4.1. `<a4j:commandButton>`

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

The `<a4j:commandButton>` requires only the `value` and `render` attributes to function. The `value` attribute specifies the text of the button and the `render` attribute specifies which areas are to be

updated. The <a4j:commandButton> uses the onclick event instead of the onsubmit event, but otherwise uses all common Ajax attributes as listed in *Chapter 2, Common Ajax attributes*.

Set disabledDefault="true"

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

4.3. <a4j:commandLink>

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

Figure 4.2. <a4j:commandLink>

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

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

4.4. <rich:componentControl>

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

- component-type: org.richfaces.ComponentControl
- component-family: org.richfaces.ComponentControl
- component-class: org.richfaces.component.html.HtmlComponentControl
- renderer-type: org.richfaces.ComponentControlRenderer

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

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

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

Example 4.2. rich:componentControl basic usage

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

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

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

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

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

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

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

Example 4.4. Using parameters

The `params` attribute

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

<f:param> elements

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

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

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

immediate
: attached during execution of the script.

onavailable
: attached after the target component is initialized.

onload
: attached after the page is loaded.

4.5. <a4j:hashParam>

Incomplete

4.6. <a4j:jsFunction>

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

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

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

Example 4.5. `<a4j:jsFunction>` example

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

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

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

4.7. `<a4j:poll>`

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

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

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

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

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

4.8. <a4j:push>

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

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

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

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

4.9. <a4j:repeat>

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

REVIEW COMMENT - a4j:repeat is not an action, should be moved. The `<a4j:repeat>` component is used to iterate changes through a repeated collection of components. It allows specific rows of items to be updated without sending Ajax requests for the entire collection. The `<a4j:repeat>` component forms the basis for many of the tabular components detailed in *Chapter 8, Tables and grids*.

The contents of the collection are determined using Expression Language (EL). The data model for the contents is specified with the `value` attribute. The `var` attribute names the object to use when iterating through the collection. This object is then referenced in the relevant child components. After an Ajax request, only the rows specified with the `ajaxKeys` attribute are updated rather than

the entire collection. *Example 4.6, "<a4j:repeat> example"* shows how to use `<a4j:repeat>` to maintain a simple table.

Example 4.6. `<a4j:repeat>` example

```
<table>
   <tbody>

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

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

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

4.10. `<a4j:ajax>`

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

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

Attaching JavaScript functions

When attaching the `<a4j:ajax>` component to non-Ajax JavaServer Faces command components, such as `<h:commandButton>` and `<h:commandLink>`, it is

important to set `disabledDefault="true"`. If this attribute is not set, a non-Ajax request is sent after the Ajax request and the page is refreshed unexpectedly.

Example 4.7. `<a4j:ajax>` example

```
<h:panelGrid columns="2">
   <h:inputText id="myinput" value="#{userBean.name}">
      <a4j:ajax event="onkeyup" reRender="outtext" />
   </h:inputText>
   <h:outputText id="outtext" value="#{userBean.name}" />
</h:panelGrid>
```

Resources

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

5.1. `<a4j:mediaOutput>`

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

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

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

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

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

Example 5.1. `<a4j:mediaOutput>` example

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

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

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

```
package demo;

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

public class MediaBean {

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

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

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

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

```
package demo;
```

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

public class MediaData implements Serializable {

   private static final long serialVersionUID = 1L;

   Integer Width=110;
   Integer Height=50;

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

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

   /* Corresponding getters and setters */
   ...

}
```

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

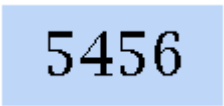

Figure 5.1. `<a4j:mediaOutput>` example result

`Serializable` interface

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

Containers

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

6.1. `<a4j:outputPanel>`

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

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

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

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

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

Processing management

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

7.1. `<a4j:queue>`

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

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

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

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

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

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

7.2. <a4j:log>

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

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

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

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

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

Example 7.1. <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.

7.3. <a4j:status>

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

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

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

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

Example 7.2. Basic `<a4j:status>` usage

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

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

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

Example 7.3. Updating a common `<a4j:status>` component

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

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

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

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

          <h:panelGroup />

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

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

Part II. rich tag library

Tables and grids

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

8.1. `<rich:column>`

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

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

In general usage, the `<rich:column>` component is used in the same was as the JavaServer Faces (JSF) `<h:column>` component. It requires no extra attributes for basic usage, as shown in *Example 8.1, "Basic column example"*.

Example 8.1. Basic column example

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

```
</rich:dataTable>
```

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

Figure 8.1. Basic column example

Columns can be merged by using the `colspan` attribute to specify how many normal columns to span. The `colspan` attribute is used in conjunction with the `breakBefore` attribute on the next column to determine how the merged columns are laid out. *Example 8.2, "Column spanning example"*.

Example 8.2. Column spanning example

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

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

Figure 8.2. Column spanning example

Similarly, the `rowspan` attribute can be used to merge and span rows. Again the `breakBefore` attribute needs to be used on related `<rich:column>` components to define the layout. *Example 8.3, "Row spanning example"* and the resulting *Figure 8.4, "Complex headers using column groups"* show the first column of the table spanning three rows.

Example 8.3. Row spanning example

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

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

Figure 8.3. Row spanning example

For details on filtering and sorting columns, refer to *Section 8.7, "Table filtering"* and *Section 8.8, "Table sorting"*.

8.2. `<rich:columnGroup>`

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

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

The `<rich:columnGroup>` can also be used to create complex headers in a table. *Example 8.4, "Complex headers using column groups"* and the resulting *Figure 8.4, "Complex headers using column groups"* demonstrate how complex headers can be achieved.

Example 8.4. Complex headers using column groups

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

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

Figure 8.4. Complex headers using column groups

8.3. `<rich:dataScroller>`

Incomplete

8.4. `<rich:dataTable>`

Incomplete

8.5. `<rich:extendedDataTable>`

* Combined with rich:scrollableDataTable

8.6. `<rich:subTable>`

Incomplete

8.7. Table filtering

Incomplete

8.8. Table sorting

Incomplete

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.

9.1. `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.

9.2. `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.

9.3. `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.

9.4. `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 9.1. `rich:findComponent` example

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

9.5. `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 9.2. `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')}" />
```

Appendix A. Revision History

Revision History
Revision 0.1 Fri Nov 20 2009 SeanRogers<serogers@redhat.com>
Outline
Revision 0.2 Thu May 06 2010 SeanRogers<serogers@redhat.com>
First draft of *Chapter 1, Introduction*
First draft of *Chapter 2, Common Ajax attributes*
First draft of *Chapter 3, Common features*
First draft of *Chapter 4, Actions*
First draft of *Chapter 5, Resources*
First draft of *Chapter 6, Containers*
First draft of *???*
First draft of *Chapter 7, Processing management*
First draft of *Chapter 9, Functions*