

Component Reference

**A reference guide
to the components
of the RichFaces 4
(*draft*) framework**

by Sean Rogers (Red Hat)

DRAFT

1. Introduction	1
1.1. Libraries	1
2. Common Ajax attributes	3
2.1. Data processing	3
2.1.1. execute	3
2.1.2. bypassUpdates	4
2.2. Rendering	4
2.2.1. render	4
2.2.2. ajaxRendered	5
2.2.3. limitRender	5
2.3. Queuing and traffic control	6
2.3.1. requestDelay	6
2.3.2. ignoreDupResponses	6
2.4. Events and JavaScript interactions	6
2.4.1. onsubmit	6
2.4.2. onbegin	6
2.4.3. onbeforeDOMupdate	7
2.4.4. oncomplete	7
2.4.5. onerror	7
I. Ajax control components	9
3. Actions	11
3.1. <a4j:ajax>	11
3.1.1. Basic usage	11
3.1.2. Reference data	11
3.2. <a4j:param>	11
3.2.1. Basic usage	12
3.2.2. Interoperability	12
3.2.3. Passing client-side parameters	13
3.2.4. Reference data	13
3.3. <a4j:commandButton>	13
3.3.1. Basic usage	14
3.3.2. Reference data	14
3.4. <a4j:commandLink>	14
3.4.1. Basic usage	15
3.4.2. Reference data	15
3.5. <a4j:jsFunction>	15
3.5.1. Basic usage	15
3.5.2. Parameters and JavaScript	16
3.5.3. Reference data	16
3.6. <a4j:poll>	16
3.6.1. Timing options	16
3.6.2. Reference data	17
3.7. <a4j:push>	17
3.7.1. Basic usage	17

3.7.2. Handling a push notification	19
3.7.3. Reference data	19
4. Containers	21
4.1. <a4j:outputPanel>	21
4.1.1. Aiding complex Ajax rendering	21
4.1.2. Panel appearance	21
4.1.3. Reference data	21
4.2. <a4j:region>	21
4.2.1. Reference data	22
5. Validation	23
5.1. <rich:validator> client-side validation	24
5.1.1. Basic usage	24
5.1.2. Messages from client-side validators	24
5.1.3. Validation triggers	25
5.1.4. Ajax fall-backs	26
5.1.5. Reference data	26
5.2. <rich:graphValidator> object validation	26
5.2.1. Basic usage	27
5.2.2. Reference data	28
6. Processing management	29
6.1. <a4j:queue>	29
6.1.1. Basic usage	29
6.1.2. Delaying requests	29
6.1.3. Duplicate responses	29
6.1.4. Queue scopes	29
6.1.5. <a4j:queue> client-side events	30
6.1.6. Reference data	30
6.1.7. <a4j:attachQueue>	30
6.2. <a4j:log>	32
6.2.1. Basic usage	32
6.2.2. Log monitoring	32
6.2.3. Reference data	33
6.2.4. Style classes and skin parameters	34
6.3. <a4j:status>	35
6.3.1. Customizing the text	35
6.3.2. Specifying a region	36
6.3.3. JavaScript API	37
6.3.4. Reference data	37
II. User interface components	39
7. Rich inputs	41
7.1. <rich:autocomplete>	41
7.1.1. Basic usage	41
7.1.2. Submission modes	42
7.1.3. Interactivity options	42

7.1.4. Customizing the filter in client and lazyClient modes	42
7.1.5. JavaScript API	43
7.1.6. Reference data	43
7.1.7. Style classes and skin parameters	44
7.2. <rich:calendar>	45
7.2.1. Basic usage	46
7.2.2. Behavior and appearance	46
7.2.3. Time of day	46
7.2.4. Localization and formatting	47
7.2.5. Using a data model	47
7.2.6. Client-side customization	48
7.2.7. JavaScript API	49
7.2.8. Reference data	50
7.2.9. Style classes and skin parameters	50
7.3. <rich:fileUpload>	55
7.3.1. Basic usage	55
7.3.2. Upload settings	56
7.3.3. Interactivity options	56
7.3.4. <rich:fileUpload> client-side events	56
7.3.5. Reference data	57
7.3.6. Style classes and skin parameters	57
7.4. <rich:inplaceInput>	59
7.4.1. Basic usage	59
7.4.2. Interactivity options	59
7.4.3. JavaScript API	60
7.4.4. Reference data	60
7.4.5. Style classes and skin parameters	61
7.5. <rich:inplaceSelect>	62
7.5.1. Basic usage	63
7.5.2. Interactivity options	63
7.5.3. JavaScript API	64
7.5.4. Reference data	65
7.5.5. Style classes and skin parameters	65
7.6. <rich:inputNumberSlider>	67
7.6.1. Basic usage	68
7.6.2. Interactivity options	68
7.6.3. JavaScript API	68
7.6.4. Reference data	69
7.6.5. Style classes and skin parameters	69
7.7. <rich:inputNumberSpinner>	71
7.7.1. Basic usage	71
7.7.2. Interactivity options	71
7.7.3. JavaScript API	71
7.7.4. Reference data	72

7.7.5. Style classes and skin parameters	72
7.8. <rich:select>	73
7.8.1. Basic usage	73
7.8.2. Using manual input	74
7.8.3. Advanced options	74
7.8.4. JavaScript API	74
7.8.5. Reference data	75
7.8.6. Style classes and skin parameters	75
8. Panels	77
8.1. <rich:panel>	77
8.1.1. Basic usage	77
8.1.2. Adding a header	77
8.1.3. Reference data	78
8.1.4. Style classes and skin parameters	78
8.2. <rich:accordion>	78
8.2.1. Basic usage	79
8.2.2. Switching panels	79
8.2.3. <rich:accordion> client-side events	80
8.2.4. <rich:accordion> server-side events	80
8.2.5. JavaScript API	80
8.2.6. Reference data	80
8.2.7. Style classes and skin parameters	81
8.2.8. <rich:accordionItem>	82
8.3. <rich:collapsiblePanel>	83
8.3.1. Basic usage	83
8.3.2. Expanding and collapsing the panel	83
8.3.3. Appearance	84
8.3.4. <rich:collapsiblePanel> server-side events	84
8.3.5. JavaScript API	84
8.3.6. Reference data	84
8.3.7. Style classes and skin parameters	85
8.4. <rich:popupPanel>	86
8.4.1. Basic usage	86
8.4.2. Showing and hiding the pop-up	86
8.4.3. Modal and non-modal panels	87
8.4.4. Size and positioning	87
8.4.5. Header and controls	88
8.4.6. Contents of the pop-up	89
8.4.7. JavaScript API	89
8.4.8. Reference data	90
8.4.9. Style classes and skin parameters	90
8.5. <rich:tabPanel>	91
8.5.1. Switching panels	92
8.5.2. <rich:tabPanel> client-side events	92

8.5.3. <rich:tabPanel> server-side events	93
8.5.4. JavaScript API	93
8.5.5. Reference data	93
8.5.6. Style classes and skin parameters	93
8.5.7. <rich:tab>	95
8.6. <rich:togglePanel>	96
8.6.1. Basic usage	96
8.6.2. Toggling between components	97
8.6.3. JavaScript API	98
8.6.4. Reference data	98
8.6.5. <rich:toggleControl>	98
8.6.6. <rich:togglePanelItem>	100
9. Tables and grids	103
9.1. <a4j:repeat>	103
9.1.1. Basic usage	103
9.1.2. Limited views and partial updates	103
9.1.3. Reference data	104
9.2. <rich:dataTable>	105
9.2.1. Basic usage	105
9.2.2. Customizing the table	105
9.2.3. Partial updates	106
9.2.4. JavaScript API	106
9.2.5. Reference data	107
9.2.6. Style classes and skin parameters	107
9.3. <rich:column>	110
9.3.1. Basic usage	110
9.3.2. Spanning columns	111
9.3.3. Spanning rows	112
9.3.4. Reference data	113
9.4. <rich:columnGroup>	113
9.4.1. Complex headers	114
9.4.2. Reference data	115
9.5. <rich:collapsibleSubTable>	115
9.5.1. Basic usage	115
9.5.2. Expanding and collapsing the sub-table	117
9.5.3. Reference data	118
9.5.4. Style classes	118
9.5.5. <rich:collapsibleSubTableToggler>	120
9.6. <rich:extendedDataTable>	121
9.6.1. Basic usage	122
9.6.2. Table appearance	122
9.6.3. Extended features	122
9.6.4. JavaScript API	128
9.6.5. Reference data	129

9.6.6. Style classes and skin parameters	129
9.7. <rich:dataGrid>	131
9.7.1. Basic usage	132
9.7.2. Customizing the grid	132
9.7.3. Partial updates	133
9.7.4. Reference data	133
9.7.5. Style classes and skin parameters	134
9.8. <rich:list>	135
9.8.1. Basic usage	135
9.8.2. Type of list	136
9.8.3. Bullet and numeration appearance	137
9.8.4. Customizing the list	137
9.8.5. Reference data	138
9.8.6. Style classes and skin parameters	139
9.9. <rich:dataScroller>	139
9.9.1. Basic usage	140
9.9.2. Appearance and interactivity	140
9.9.3. JavaScript API	141
9.9.4. Reference data	142
9.9.5. Style classes and skin parameters	142
9.10. Table filtering	143
9.10.1. Basic filtering	143
9.10.2. External filtering	145
9.11. Table sorting	146
9.11.1. External sorting	146
10. Trees	149
10.1. <rich:tree>	149
10.1.1. Basic usage	149
10.1.2. Appearance	150
10.1.3. Expanding and collapsing tree nodes	151
10.1.4. Selecting tree nodes	152
10.1.5. Identifying nodes with the rowKeyConverter attribute	152
10.1.6. Event handling	152
10.1.7. Reference data	152
10.1.8. Style classes	153
10.1.9. <rich:treeNode>	153
10.2. Tree adaptors	156
10.2.1. <rich:treeModelAdaptor>	156
10.2.2. <rich:treeModelRecursiveAdaptor>	158
11. Menus and toolbars	161
11.1. <rich:dropDownMenu>	161
11.1.1. Basic usage	161
11.1.2. Menu content	161
11.1.3. Appearance	161

11.1.4. Expanding and collapsing the menu	162
11.1.5. Reference data	162
11.1.6. Style classes and skin parameters	163
11.2. Menu sub-components	164
11.2.1. <rich:menuItem>	164
11.2.2. <rich:menuGroup>	166
11.2.3. <rich:menuSeparator>	167
11.3. <rich:panelMenu>	167
11.3.1. Basic usage	168
11.3.2. Interactivity options	168
11.3.3. Appearance	168
11.3.4. Submission modes	170
11.3.5. <rich:panelMenu> server-side events	170
11.3.6. JavaScript API	170
11.3.7. Reference data	170
11.3.8. Style classes and skin parameters	171
11.3.9. <rich:panelMenuGroup>	175
11.3.10. <rich:panelMenuItem>	176
11.4. <rich:toolbar>	177
11.4.1. Basic usage	178
11.4.2. Appearance	178
11.4.3. Grouping items	179
11.4.4. Reference data	179
11.4.5. Style classes and skin parameters	179
11.4.6. <rich:toolbarGroup>	180
12. Output and messages	183
12.1. <rich:message>	183
12.1.1. Basic usage	183
12.1.2. Appearance	183
12.1.3. Reference data	184
12.1.4. Style classes and skin parameters	184
12.2. <rich:messages>	185
12.2.1. Basic usage	185
12.2.2. Appearance	185
12.2.3. Reference data	187
12.2.4. Style classes and skin parameters	187
12.3. <rich:progressBar>	187
12.3.1. Basic usage	187
12.3.2. Customizing the appearance	188
12.3.3. Using set intervals	189
12.3.4. Update mode	190
12.3.5. JavaScript API	190
12.3.6. Reference data	190
12.3.7. Style classes and skin parameters	191

12.4. <rich:tooltip>	191
12.4.1. Basic usage	191
12.4.2. Attaching the tool-tip to another component	192
12.4.3. Appearance	193
12.4.4. Update mode	193
12.4.5. <rich:tooltip> client-side events	194
12.4.6. JavaScript API	194
12.4.7. Reference data	194
12.4.8. Style classes and skin parameters	195
13. Drag and drop	197
13.1. <rich:dragSource>	197
13.1.1. Basic usage	197
13.1.2. Dragging an object	197
13.1.3. Reference data	197
13.2. <rich:dropTarget>	197
13.2.1. Basic usage	198
13.2.2. Handling dropped data	198
13.2.3. Reference data	198
13.2.4. Style classes	198
13.3. <rich:dragIndicator>	199
13.3.1. Basic usage	199
13.3.2. Styling the indicator	199
13.3.3. Reference data	199
13.3.4. Style classes	200
14. Layout and appearance	201
14.1. <rich:jQuery>	201
14.1.1. Basic usage	201
14.1.2. Defining a selector	201
14.1.3. Event handlers	202
14.1.4. Timed queries	202
14.1.5. Named queries	203
14.1.6. Dynamic rendering	204
14.1.7. Reference data	204
15. Functions	205
15.1. <rich:componentControl>	205
15.1.1. Basic usage	205
15.1.2. Passing parameters to API methods	205
15.1.3. Reference data	206
15.2. rich:clientId	206
15.3. rich:component	206
15.4. rich:element	206
15.5. rich:findComponent	206
15.6. rich:isUserInRole	207

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*, *JS Documentation*, and *VDL Documentation* available from the RichFaces website.

For further examples for each component, refer to the *RichFaces Showcase* at <http://richfaces-showcase.appspot.com/>.

1.1. Libraries

The RichFaces framework is made up of two tag libraries: the `a4j` library and the `rich` library.

`a4j` library

The `a4j` tag library provides core Ajax and utility components.

`rich` library

The `rich` tag library provides ready-made, self-contained, rich user-interface components. These components have built-in Ajax support. By default, the components don't require additional configuration in order to send requests or update, but can also be customized by plugging in utility behaviors.

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. Data processing

The RichFaces Ajax script is built on a base of the JSF 2 Ajax script. 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.

2.1.1. `execute`

The `execute` attribute allows JSF processing to be limited to defined components. The `execute` attribute can point to an `id` identifier of a specific component to process. Components can also be identified through the use of Expression Language (EL).

Alternatively, the `execute` attribute accepts the following keywords:

`@all`

Every component is processed.

`@none`

No components are processed.

`@this`

The requesting component with the `execute` attribute is processed. This is the default behavior for components.

`@form`

The form that contains the requesting component is processed.

`@region`

The region that contains the requesting component is processed. Use the `<a4j:region>` component as a wrapper element to specify regions.

Some components make use of additional keywords. These are detailed under the relevant component entry in this book.

2.1.2. `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. This is the opposite functionality to the `execute` attribute in RichFaces.

2.2. Rendering

2.2.1. `render`

The `render` attribute provides a reference to one or more components 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` identifiers as a reference. Components can be referenced by their `id` identifier alone, or by their `clientId` identifier 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 `clientId` 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>

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

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.



Differences between JSF Ajax and RichFaces Ajax

JSF evaluates all parameters during page rendering. As such, during a request from a page, all `execute` and `render` values are passed to the server from the client. In contrast, RichFaces evaluates these options at the server side during the current request.

This means that with JSF, making changes during a request to a `render` value defined with EL will not influence the request. RichFaces, however, will always use the newer values.

The RichFaces approach additionally increases data integrity. Parameters that are changed from the client side are re-evaluated on the server, where they cannot be changed.



Conditionally-rendered component updates

A common problem with using `render` occurs when the referenced component is conditionally rendered via the `rendered` attribute. If a component is not initially rendered, it does not have any HTML representation in the Document Object Model (DOM). As such, when RichFaces renders the component via Ajax, 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>` component. The `<a4j:outputPanel>` component will receive the update and render the component as required.

2.2.2. `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.

The `ajaxRendered` attribute's functionality is the basis for the `<a4j:outputPanel>` component. The `<a4j:outputPanel>` component is designed to mark parts of the page for automatic updating. Refer to [Section 4.1, “<a4j:outputPanel>”](#) for details.

Automatic rendering of such components can be repressed by adding `limitRender="true"` to the requesting component, as described in [Section 2.2.3, “limitRender”](#).

2.2.3. `limitRender`

RichFaces Ajax-enabled components and Ajax behaviors 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>
```

2.3. Queuing and traffic control

2.3.1. 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 replaced with the new one.

2.3.2. 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.

2.4. Events and JavaScript interactions

JSF provides global `jsf.ajax.onError` and `jsf.ajax.onEvent` events to define handlers (the `jsf.ajax.onEvent` event is used for all `begin`, `success`, and `complete` events). RichFaces adds event-specific attributes at the component level.

2.4.1. onsubmit

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

2.4.2. onbegin

The `onbegin` event attribute invokes the event listener *after* the Ajax request is sent.

2.4.3. onbeforedomupdate

The `onbeforedomupdate` event attribute invokes the event listener after the Ajax response has been returned but *before* the DOM tree of the browser has been updated.

2.4.4. oncomplete

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

2.4.4.1. data

The `data` attribute allows additional data to be handled with the `oncomplete` event. Use JSF Expression Language (EL) 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 `event.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" oncomplete="showTheName(event.data.name)" data="#{userBean.name}" />
```

2.4.5. onerror

The `onerror` event attribute invokes the event listener when an error has occurred during Ajax communications.

Part I. Ajax control components

DRAFT

DRAFT

Actions

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

3.1. <a4j:ajax>

The <a4j:ajax> behavior allows Ajax capability to be added to a non-Ajax component. The non-Ajax component must implement the `ClientBehaviorHolder` interface for all the event attributes that support behavior rendering.

3.1.1. Basic usage

The <a4j:ajax> behavior is placed as a direct child to the component that requires Ajax support.

Point the `event` attribute to the standard JSF event that triggers the behavior. If the `event` attribute is not defined, the behavior is triggered on the event that normally provides interaction behavior for the parent component.

Example 3.1. <a4j:ajax> example

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

3.1.2. 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`

3.2. <a4j:param>

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

3.2.1. Basic usage

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

- The `value` attribute is the initial value of the parameter.
- The `assignTo` attribute defines the bean property. The property is updated if the parent command component performs an action event during the *Process Request* phase.

Example 3.2, “<a4j:param> example” shows a simple implementation along with the accompanying managed bean.

Example 3.2. `<a4j:param>` example

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

```
public class ParamBean {
    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.

3.2.2. Interoperability

The `<a4j:param>` tag can be used with non-Ajax components in addition to Ajax components. This includes components which are working through the `GET` request, such as the `<h:link>` and `<h:button>` components. In this way, data model values can also be updated without any Java 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.

3.2.3. Passing client-side parameters

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.

Example 3.3. Passing client-side parameters

```
<h:form>
  <a4j:commandButton value="Show Screen Size" render="infoPanel">
    <a4j:param name="w" value="screen.width"
               assignTo="#{paramBean.screenWidth}" noEscape="true" />
    <a4j:param name="h" value="screen.height"
               assignTo="#{paramBean.screenHeight}" noEscape="true" />
  </a4j:commandButton>

  <h:panelGrid columns="2" id="infoPanel">
    <h:outputText value="Width:" />
    <h:outputText value="#{paramBean.screenWidth}" />
    <h:outputText value="Height:" />
    <h:outputText value="#{paramBean.screenHeight}" />
  </h:panelGrid>
</h:form>
```

The command button triggers the `<a4j:param>` behaviors and renders the output text. The `<a4j:param>` behaviors pass client-side parameters for the screen width and height through the backing bean. These parameters are then used to populate the output text values.

3.2.4. Reference data

- *component-type*: `org.ajax4jsf.parameter`
- *component-class*: `org.ajax4jsf.component.html.HTMLParameter`

3.3. `<a4j:commandButton>`

The `<a4j:commandButton>` component is similar to the JavaServer Faces (JSF) `<h:commandButton>` component, but additionally includes Ajax support.

**Figure 3.1.** `<a4j:commandButton>`

The `<a4j:commandButton>` component executes the complete form

Button controls are typically used to perform complete form submissions for data storing. As a consequence, the `<a4j:commandButton>` component has the `execute="@form"` setting by default. To limit rendering to a different scope, redefine the `execute` attribute.

3.3.1. Basic usage

The `<a4j:commandButton>` requires only the `value` attribute to function. Use the `value` attribute to specify the text of the button.

By default, the `<a4j:commandButton>` uses the `click` event instead of the `submit` event.

3.3.2. 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`

3.4. `<a4j:commandLink>`

The `<a4j:commandLink>` component is similar to the JavaServer Faces (JSF) `<h:commandLink>` component, except that it includes plugged-in Ajax behavior.

[Command Link](#)

Figure 3.2. `<a4j:commandLink>`

The `<a4j:commandLink>` component executes the complete form

Link controls are typically used to perform complete form submissions for data storing. As a consequence, the `<a4j:commandLink>` component has the

`execute="@form"` setting by default. To limit rendering to a different scope, redefine the `execute` attribute.

3.4.1. Basic usage

The `<a4j:commandLink>` requires only the `value` attribute to function. Use the `value` attribute to specify the text of the link.

The `<a4j:commandLink>` uses the `click` event instead of the `submit` event.

3.4.2. 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`

3.5. `<a4j:jsFunction>`

The `<a4j:jsFunction>` component performs Ajax requests directly from JavaScript code and retrieves server-side data. The server-side data is returned in JavaScript Object Notation (JSON) format prior to the execution of any JavaScript code defined using the `oncomplete` attribute.

3.5.1. Basic usage

The `<a4j:jsFunction>` component requires the `data` attribute. Use the `data` attribute to define where the retrieved server-side data is stored.

[Example 3.4, “`<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 3.4. `<a4j:jsFunction>` example

```
<table width="400">
  <tbody>
    <tr>
      <td>
        <span onmouseover="updateName('Kate')"
              onmouseout="updateName('')">Kate</span>
      </td>
      <td>
        <span onmouseover="updateName('John')"
              onmouseout="updateName('')">John</span>
      </td>
    </tr>
  </tbody>
</table>
```

```

        </td>
        <td>
            <span onmouseover="updateName('Alex')"
                onmouseout="updateName(' ')">Alex</span>
        </td>
    </tr>
    <tr>
        <td colspan="3">
            Name: <b><h:outputText id="showname" value="#{functionBean.text}" /
></b>
        </td>
    </tr>
</tbody>
</table>
<h:form>
    <a4j:jsFunction name="updateName" render="showname">
        <a4j:param name="name" assignTo="#{functionBean.text}" />
    </a4j:jsFunction>
</h:form>

```

The output text for the name is changed depending on which table cell the user hovers over with the mouse. The `<a4j:jsFunction>` component manages the updating and display of the name.

3.5.2. Parameters and JavaScript

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

3.5.3. 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`

3.6. <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.

3.6.1. Timing options

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

The `<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.

3.6.2. 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`

3.7. `<a4j:push>`

The `<a4j:push>` component performs real-time updates on the client side from events raised at the server side. The events are pushed out to the client through the Java Message Service (JMS). When the `<a4j:push>` component is triggered by a server event, it can in turn cause Ajax updates and changes.

The `<a4j:push>` component uses the Comet model for pushing data to the client.

3.7.1. Basic usage

RichFaces requires the use of the `jquery-atmosphere.js` client plug-in for push functionality. Add the server-side Atmosphere framework as a build dependency to your project.

The JMS instance on the back-end must be configured to work with your `<a4j:push>` components. Refer to the *JBoss Application Server Administration Console Guide* for details on managing JBoss Application Server through the Administration Console.

Example 3.5. JMS server configuration

This example describes the JMS server configuration required for a simple chat room. The chat room requires topics on the JMS server for the push functionality to check for new messages. Create a new JMS topic using the following settings:

- **Name:** chat
- **JNDI name:** /topic/chat
- Use the default settings for other options.

Add a single role for the topic in the same form using the following settings:

- **Name:** guest

- **Send:** true
- **Consume:** true
- **Create subscriber:** true
- **Delete subscriber:** true
- **Create durable subscriber:** true
- **Delete durable subscriber:** true

Ensure the **Create durable subscriber** and **Delete durable subscriber** options are set to **true** for push functionality. Durable subscriptions receive all events, including those events which were sent while the push component was not connected.

Refer to *JMS Documentation* for details on configuring the JMS Server.

With the JMS server configured, use the `<a4j:push>` component's `address` attribute to reference the topic on the JMS server that contains the pushed messages.

Example 3.6. Basic usage

```
<rich:list value="#{chatBean.users}" var="user" id="users" type="unordered">
    #{user.nick}
</rich:list>
...
<a4j:push address="#{chatBean.listSubtopic}@chat"
    onerror="alert(event.rf.data)">
    <a4j:ajax event="dataavailable" render="users" execute="@none" />
</a4j:push>
```

```
public String getListSubtopic() {
    return this.getUserName() + SUBTOPIC_SEPARATOR + channelName + "List";
}

@Override
protected void onUserList(String channel, User[] users) {
    try {
        getTopicsContext().publish(new TopicKey("chat", getListSubtopic()), null);
    } catch (MessageException e) {
        LOGGER.error(e.getMessage(), e);
    }
}

@Override
protected void onJoin(String channel, String sender, String login, String hostname) {
```

```
getTopicsContext().publish(new TopicKey("chat", getListSubtopic()), null);
Message messageObject = new Message("joined channel", sender,
    DateFormat.getInstance().format(new Date()));
getTopicsContext().publish(new TopicKey("chat", getMessagesSubtopic()), messageObject);
} catch (MessageException e) {
    LOGGER.error(e.getMessage(), e);
}
}
```

The example demonstrates a simple use of the `<a4j:push>` component to manage a list of users in a chat room. The `<a4j:push>` component refers to the `#{chatBean.listSubtopic}@chat` address, which has been created on the JMS server. It then uses the sub-topics to separate messages across different topics.

When a new message arrives, the `<a4j:ajax>` behavior causes the user list to update. If an error occurs, the user is alerted.

3.7.2. Handling a push notification

A push notification sent to the `<a4j:push>` behavior will cause it to trigger any event handlers defined using the `ondataavailable` event attribute.

The `<a4j:push>` component should also include the `onerror` event attribute to inform the user when an error has occurred with the push notifications.

Example 3.7. Handling a push notification

```
<a4j:push address="#{chatBean.messagesSubtopic}@chat"
    onerror="alert(event.rf.data)"
    ondataavailable="jQuery('<div/>').
        prependTo('.#{chatBean.channelName}Output').text(
            getMessageString(event.rf.data))" />
```

The example uses the `dataavailable` event attribute with JavaScript to update messages in a chat room. The `event.rf.data` parameter contains JMS message data serialized to JavaScript.

3.7.3. 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`

Containers

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

4.1. `<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.

4.1.1. Aiding complex Ajax rendering

Use the `<a4j:outputPanel>` component to wrap behaviors when using complex Ajax rendering. Parent components may not render correctly when attached behaviors trigger updates. Point the behaviors to the wrapping `<a4j:outputPanel>` component instead of the parent components. The `<a4j:outputPanel>` component is properly encoded to ensure the wrapped components are correctly rendered.

4.1.2. 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 `` 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.

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.

4.1.3. 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`

4.2. `<a4j:region>`

The `<a4j:region>` component specifies a part of the JSF component tree to be processed on the server. The region causes all the `a4j` and `rich` Ajax controls to execute: decoding, validating,

and updating the model. The region causes these components to execute even if not explicitly declared. As such, processing areas can more easily be marked using a declarative approach.

Regions can be nested, in which case only the parent region of the component initiating the request will be processed.

4.2.1. 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

JavaServer Faces 2 provides built-in support for bean validation as per the Java Specification Request JSR-303 standard. As such, containers must validate model objects. Validation is performed at different application tiers according to annotation-based constraints. Refer to <http://jcp.org/en/jsr/detail?id=303> for further details on the JSR-303 specification.

Example 5.1, “JSR-303 validation annotations” shows an example JSF managed bean. The bean includes JSR-303 annotations for validation. Validation annotations defined in this way are registered on components bound to the bean properties, and validation is triggered in the *Process Validation* phase.

Example 5.1. JSR-303 validation annotations

```
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;

@ManagedBean
@RequestScoped
public class UserBean {

    @Size(min=3, max=12)
    private String name = null;

    @Pattern(regexp = "^[\\w\\-]{1}\\.[\\w\\-]{1}@([\\w\\-]{1}\\.){1}[a-zA-Z]{2,4}$" , message="Bad email")
    private String email = null;

    @Min(value = 18)
    @Max(value = 99)
    private Integer age;

    //...
    //Getters and Setters
}
```



Requirements

Bean validation in both JavaServer Faces and RichFaces requires the *JSR-303* implementation. The implementation is bundled with JEE 6 Application Server.

If using Tomcat or another simple servlet container, add the `validation-api` Java Archive and a validation provider (such as Hibernate Validator) to your application libraries.

5.1. <rich:validator> client-side validation

The validation built in to JavaServer Faces 2 occurs on the server side. The `<rich:validator>` behavior adds client-side validation to a control based on registered server-side validators. It provides this validation without the need to reproduce the server-side annotations. The `<rich:validator>` behavior triggers all client validator annotations listed in the relevant managed bean.

5.1.1. Basic usage

The `<rich:validator>` behavior is added as a child element to any input control. The value of the input control must reference a managed bean. The content of the input control validates on the client-side based on registered server-side validators included in the managed bean.

Example 5.2. Basic usage

```
<h:inputText value="#{userBean.name}">
  <rich:validator/>
</h:inputText>
```



JSF validation tags

JSF validation tags, such as `<f:validateLength>` and `<f:validateDoubleRange>` tags, can be declared alongside `<rich:validator>` behaviors. However, because this duplicates the validation processes at both the view and model level, it is not recommended.

5.1.2. Messages from client-side validators

Use the `<rich:message>` and `<rich:messages>` components to display validation messages. The `for` attribute of the `<rich:message>` component references the `id` identifier of the input control being validated.

Example 5.3. Messages

```
<rich:panel header="User information">
  <h:panelGrid columns="3">
```

```

<h:outputText value="Name:" />
<h:inputText value="#{validationBean.name}" id="name">
    <rich:validator />
</h:inputText>
<rich:message for="name" />

<h:outputText value="Email" />
<h:inputText value="#{validationBean.email}" id="email">
    <rich:validator />
</h:inputText>
<rich:message for="email" />

<h:outputText value="Age" />
<h:inputText value="#{validationBean.age}" id="age">
    <rich:validator />
</h:inputText>
<rich:message for="age" />

<h:outputText value="I agree the terms" />
<h:selectBooleanCheckbox value="#{validationBean.agree}" id="agree">
    <rich:validator/>
</h:selectBooleanCheckbox>
<rich:message for="agree" />

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

```

Failed validation checks are reported using `<rich:message>` components. The validation annotations in the managed bean are outlined in [Example 5.1, “JSR-303 validation annotations”](#).

User information		
Name:	<input type="text" value="i"/>	✖ size must be between 3 and 12
Email	<input type="text" value="i"/>	✖ Bad email
Age	<input type="text" value="i"/>	✖ j_idt134:age: 'i' must be a number
I agree the terms	<input type="checkbox"/>	✖ must be true

5.1.3. Validation triggers

Use the `event` attribute to specify which event on the input control triggers the validation process. By default, the `<rich:validator>` behavior triggers validation when the input control is changed (`event="change"`).

Example 5.4. Validation triggers

```

<h:inputText value="#{userBean.name}" />

```

```
<rich:validator event="keyup" />
</h:inputText>
```

The `event` attribute is changed to the `keyup` event, such that validation takes place after each key press.

5.1.4. Ajax fall-backs

If no client-side validation method exists for a registered server-side validator, Ajax fall-back is used. The `<rich:validator>` behavior invokes all available client-side validators. If all the client-side validators return valid, RichFaces performs an Ajax request to invoke the remaining validators on the server side.

5.1.5. Reference data

- `component-type`: `org.richfaces.validator`
- `component-class`: `org.richfaces.component.html.Htmlvalidator`
- `component-family`: `org.richfaces.validator`
- `renderer-type`: `org.richfaces.validatorRenderer`
- `tag-class`: `org.richfaces.taglib.validatorTag`

5.2. `<rich:graphValidator>` object validation

The `<rich:graphValidator>` component is used to wrap a set of input components related to one object. The object defined by the `<rich:graphValidator>` component can then be completely validated. The validation includes all object properties, even those which are not bound to the individual form components. Validation performed in this way allows for cross-field validation in complex forms.



Validation without model updates

The `<rich:graphValidator>` component performs a `clone()` method on the referenced bean instance during the validation phase. The cloned object is validated and triggers any required validation messages. As such, the model object remains clean, and the lifecycle is interrupted properly after the *Process Validations* phase.

Ensure the referenced object implements the `Cloneable` interface, and allows a deep clone if required.

5.2.1. Basic usage

The `<rich:graphValidator>` element must wrap all the input controls that are required to validate the object. The `value` attribute names the bean for the validating object.

Example 5.5. Basic usage

The example demonstrates a simple form for changing a password. The two entered passwords must match, so a `<rich:graphValidator>` component is used for cross-field validation.

```
<h:form>
  <rich:graphValidator value="#{userBean}">
    <rich:panel header="Change password">
      <rich:messages/>
      <h:panelGrid columns="3">
        <h:outputText value="Enter new password:" />
        <h:inputSecret value="#{userBean.password}" id="pass"/>
        <rich:message for="pass"/>
        <h:outputText value="Confirm the new password:" />
        <h:inputSecret value="#{userBean.confirm}" id="conf"/>
        <rich:message for="conf"/>
      </h:panelGrid>
      <a4j:commandButton value="Store changes"
        action="#{userBean.storeNewPassword}" />
    </rich:panel>
  </rich:graphValidator>
</h:form>
```

The input controls validate against the following bean:

```
@ManagedBean
@RequestScoped
public class UserBean implements Cloneable {

    @Size(min = 5, max = 15, message="Wrong size for password")
    private String password;
    @Size(min = 5, max = 15, message="Wrong size for confirmation")
    private String confirm;
    private String status = "";

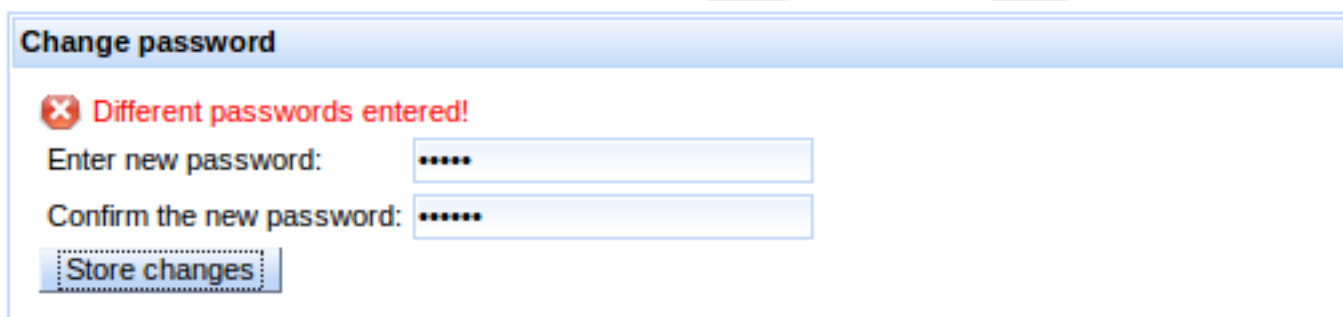
    @AssertTrue(message = "Different passwords entered!")
    public boolean isPasswordsEquals() {
        return password.equals(confirm);
    }

    public void storeNewPassword() {
```


```
FacesContext.getCurrentInstance().addMessage(new FacesMessage(FacesMessage.SEVERITY_INFO, "Successfully  
changed!", "Successfully changed!"));  
}  
  
...  
}
```

When validation occurs, the whole object is validated against the annotation constraints. The `@AssertTrue` annotation relies on the `isPasswordsEqual()` function to check whether the two entered passwords are equal.

If the entered passwords do not match, an error message is displayed:



Change password

 **Different passwords entered!**

Enter new password:

Confirm the new password:

5.2.2. 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.

6.1. <a4j:queue>

The <a4j:queue> component manages the JSF queue of Ajax requests. It provides additional options for a finer control of request processing.

6.1.1. Basic usage

The <a4j:queue> component works in the same basic way as the standard JSF queue. It can be enabled and disabled through the `enabled` attribute.



Requests from other libraries

The <a4j:queue> component does not handle standard JSF requests or requests from component libraries other than RichFaces.

6.1.2. Delaying requests

Use the `requestDelay` attribute to add a delay between each request in the queue. Set the `requestDelay` attribute to the number of milliseconds to wait in between each request. Delaying requests avoids unnecessary processing for actions that would otherwise cause multiple requests, such as typing. Similar requests in the queue are combined while waiting for the request delay.

Example 6.1. Delaying requests

```
<a4j:queue requestDelay="1500"/>
```

The queue delays each request by 1500 milliseconds.

6.1.3. Duplicate responses

The client side can update unnecessarily if duplicate responses require similar updates. Set `ignoreDupResponses="true"` to ignore duplicate responses. With this setting, the client will not update from a request if a similar request is in the queue.

6.1.4. Queue scopes

Define the queue scope to make it the default queue for all requests in that scope. The scope depends on the placement of the queue and any naming identifiers.

- An unnamed `<a4j:queue>` component placed outside any forms becomes the default queue for all requests on the page.
- An unnamed `<a4j:queue>` component placed inside a form becomes the default queue for all requests within that form.
- Use the `name` identifier attribute to name an `<a4j:queue>` component. Named queues can be accessed with the `<a4j:attachQueue>` behavior to act as a queue for specific components and behaviors. Refer to [Section 6.1.7, “<a4j:attachQueue>”](#) for details.

Example 6.2. Queue scopes

```
<a4j:queue name="viewQueue" requestDelay="2000"/>
<h:form>
    <a4j:queue name="formQueue" requestDelay="1500"/>
    ...
</h:form>
```

The queue outside the form is scoped to the view. The queue inside the form is scoped only to that form.

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

The `<a4j:queue>` component features several events relating to queuing actions in addition to the common JSF events:

- The `complete` event 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 `requestqueue` event is fired after a new request has been added to the queue.
- The `requestdequeue` event is fired after a request has been removed from the queue.

6.1.6. 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`

6.1.7. `<a4j:attachQueue>`

The `<a4j:attachQueue>` behavior is used together with a `<a4j:queue>` component to further customize queuing for particular components and behaviors. The `<a4j:attachQueue>` behavior

can override the scope-wide queue settings for an individual component, or attach specific requests to a queue.

6.1.7.1. Overriding scope settings

Queues can be scoped to various levels as described in [Section 6.1.4, “Queue scopes”](#). Use an `<a4j:attachQueue>` behavior in the same scope as a queue to override the queue settings for a particular control.

Example 6.3. Overriding scope settings

```
<a4j:queue requestDelay="2000" />
<h:form>
  <rich:panel>
    <h:inputText>
      <a4j:ajax event="keyup" />
    </h:inputText>
    <a4j:commandButton value="submit">
      <a4j:attachQueue requestDelay="0" />
    </a4j:commandButton>
  </rich:panel>
</h:form>
```

The request delay is overridden by the `<a4j:attachQueue>` behavior on the submit button.

6.1.7.2. Using a named queue

Name an `<a4j:queue>` component using the `name` attribute. It can then be used by specific components through the `<a4j:attachQueue>` behavior. Use the `name` attribute of the `<a4j:attachQueue>` behavior to identify the name of the destination queue.

Example 6.4. Using a named queue

```
<a4j:queue name="viewQueue" />
<h:form>
  <a4j:queue name="formQueue" />
  <rich:panel>
    <a4j:commandButton value="submit">
      <a4j:attachQueue name="viewQueue" />
    </a4j:commandButton>
  </rich:panel>
</h:form>
```

The requests from the button are attached to the `viewQueue` queue, rather than the `formQueue` queue.

6.1.7.3. Grouping requests

Use grouping to process multiple requests together. Specify a grouping identifier with the `requestGroupId` attribute. Requests from multiple `<a4j:attachQueue>` behaviors can use the same identifier to group requests together.

Example 6.5. Grouping requests

```
<h:form>
  <a4j:queue requestDelay="2000"/>
  <h:inputText id="input1" value="#{queueBean.text1}">
    <a4j:attachQueue requestGroupId="registrationForm"/>
  </h:inputText>
  <h:inputText id="input2" value="#{queueBean.text2}">
    <a4j:attachQueue requestGroupId="registrationForm"/>
  </h:inputText>
</h:form>
```

Requests from both the text input boxes are grouped together with the `registrationForm` identifier.

6.1.7.4. Reference data

- *renderer-type*: `org.ajax4jsf.AttachQueueRenderer`
- *component-class*: `org.ajax4jsf.component.html.HtmlAttachQueue`
- *component-family*: `org.ajax4jsf.AttachQueue`
- *tag-class*: `org.ajax4jsf.taglib.html.jsp.AttachQueueTag`

6.2. `<a4j:log>`

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

6.2.1. Basic usage

The `<a4j:log>` component doesn't require any additional attributes for basic functionality.

6.2.2. Log monitoring

The `mode` attribute determines how the log appears on the page.

- Set `mode="inline"` to place the logging data in-line on the current page. This is the default setting.

- Set `mode="popup"` to present the logging data in a new pop-up window. 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:

- Set `level="ERROR"` to log all errors.
- Set `level="FATAL"` to log only fatal messages.
- Set `level="INFO"` to log only informational messages.
- Set `level="WARN"` to log only warning messages.
- Set `level="ALL"` to log all data. This is the default setting.

Example 6.6. `<a4j:log>` example

```
<a4j:log level="ALL" mode="inline" />
```

```
debug[09:57:20.732]: New request added to queue. Queue requestGroupId changed to
j_idt718:j_idt720
debug[09:57:20.732]: Queue will wait 0ms before submit
debug[09:57:20.733]: richfaces.queue: will submit request NOW
info [09:57:20.736]: Received 'begin' event from <input id=j_idt718:j_idt720 ...>
info [09:57:21.383]: Received 'beforeDOMupdate' event from <input id=j_idt718:j_idt720 ...>
debug[09:57:21.384]: Server returned responseText: <?xml version='1.0' encoding='UTF-8'?>
<partial-response><changes><update id="javax.faces.ViewState"><![
CDATA[H4sIAAAAAAAAAANVabWwcxRmeO9/FX4lj0zgfjI10QFUvGeHODi2KnKxndjgJJbPdfmQep27ndzteW93mZk
```



Log renewal

The log is automatically renewed after each Ajax request. It does not need to be explicitly re-rendered. To clear previous requests, implement a **Clear** button or similar functionality.

6.2.3. Reference data

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

- `renderer-type: org.ajax4jsf.LogRenderer`

6.2.4. Style classes and skin parameters

The `<a4j:log>` component is intended primarily for debugging during development. However it is still possible to style the component if desired.

Table 6.1. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-log</code> This class defines styles for the log.	<code>generalTextColor</code>	<code>color</code>
<code>.rf-log-popup</code> This class defines styles for the log when it appears as a pop-up.	No skin parameters.	
<code>.rf-log-popup-cnt</code> This class defines styles for the content of the log pop-up.	No skin parameters.	
<code>.rf-log-inline</code> This class defines styles for the log when it appears in-line.	No skin parameters.	
<code>.rf-log-contents</code> This class defines styles for the log contents.	No skin parameters.	
<code>.rf-log-entry-lbl</code> This class defines styles for a label in the log.	No skin parameters.	
<code>.rf-log-entry-lbl-debug</code> This class defines styles for the debug label in the log.	No skin parameters.	
<code>.rf-log-entry-lbl-info</code> This class defines styles for the information label in the log.	No skin parameters.	
<code>.rf-log-entry-lbl-warn</code> This class defines styles for the warning label in the log.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-log-entry-lbl-error</code> This class defines styles for the error label in the log.	No skin parameters.	
<code>.rf-log-entry-msg</code> This class defines styles for a message in the log.	No skin parameters.	
<code>.rf-log-entry-msg-debug</code> This class defines styles for the debug message in the log.	No skin parameters.	
<code>.rf-log-entry-msg-info</code> This class defines styles for the information message in the log.	No skin parameters.	
<code>.rf-log-entry-msg-warn</code> This class defines styles for the warning message in the log.	No skin parameters.	
<code>.rf-log-entry-msg-error</code> This class defines styles for the error message in the log.	No skin parameters.	
<code>.rf-log-entry-msg-xml</code> This class defines styles for an XML message in the log.	No skin parameters.	

6.3. <a4j:status>

The <a4j:status> component displays the status of current Ajax requests. The status can be either in progress, complete, or an error is shown after a failed request.

6.3.1. Customizing the text

The text display can be customized depending on the current status.

- The `startText` attribute defines the text shown after the request has been started and is currently in progress. Set the styles for the text with the `startStyle` and `startStyleClass` attributes. Alternatively, use the `start` facet to customize the text appearance.

- The `stopText` attribute defines the text shown once the request is complete. Set the styles for the text with the `stopStyle` and `stopStyleClass` attributes. Alternatively, use the `stop` facet to customize the text appearance.

If the `stopText` attribute is not defined, and no facet exists for the stopped state, the complete status is simply not shown. In this way, only the progress of the request is displayed to the user, along with any errors.

- The `errorText` attribute defines the text shown when an error has occurred. Set the styles for the text with the `errorStyle` and `errorStyleClass` attributes. Alternatively, use the `error` facet to customize the text appearance.

Example 6.7. Basic `<a4j:status>` usage

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

6.3.2. Specifying a region

The `<a4j:status>` component monitors the status of the region relevant to where it is placed.

- If unnamed and placed outside any forms, it monitors the status at the view level.
- If unnamed and placed inside a form, it monitors the status at the form level.

However, if identified with the `name` attribute, the `<a4j:status>` component can monitor any Ajax component or behavior. Use the `status` attribute on the Ajax component or behavior to reference the `name` identifier of the `<a4j:status>` component.

Example 6.8. Updating a referenced `<a4j:status>` component

```
<rich:panel>
  <f:facet name="header">
    <h:outputText value="User Details Panel" />
  </f:facet>
  <h:panelGrid columns="3">
    <h:outputText value="User name:" />
    <h:inputText value="#{userBean.name}">
      <a4j:ajax status="nameStatus" event="keyup" />
    </h:inputText>
    <a4j:status name="nameStatus">
      <f:facet name="start">
        <h:graphicImage value="/images/ai.gif" />
      </f:facet>
    </a4j:status>
    <h:outputText value="Address:" />
  </h:panelGrid>
</rich:panel>
```

```
<h:inputText value="#{userBean.address}">
  <a4j:ajax status="addressStatus" event="keyup" />
</h:inputText>
<a4j:status name="addressStatus">
  <f:facet name="start">
    <h:graphicImage value="/images/ai.gif" />
  </f:facet>
</a4j:status>
</h:panelGrid>
</rich:panel>
```

6.3.3. JavaScript API

The `<a4j:status>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`start()`

Switches status to the `start` state.

`stop()`

Switches status to the `stop` state.

`error()`

Switches status to the `error` state.

6.3.4. 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. User interface components

DRAFT

DRAFT

Rich inputs

This chapter details rich components for user input and interaction.

7.1. <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.

The auto-complete box is a standard JSF `UIInput` control with added validation.



Figure 7.1. <rich:autocomplete>

7.1.1. 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 one of two different methods:

- The `autocompleteMethod` attribute points to a method which returns a list of suggestions according to a supplied prefix.



`client` and `lazyClient` modes

The prefix is normally ignored in `client` and `lazyClient` modes. In these modes, the component requests the suggestion list once only, and performs filtering on the client.

- The `autocompleteList` attribute points to a collection of suggestions.

Example 7.1. Defining suggestion values

Using the `autocompleteMethod` attribute

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

The `<rich:autocomplete>` component uses the `bean.autocomplete` method to provide suggestions, based on the entered prefix.

Using the `autocompleteList` attribute

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

The `<rich:autocomplete>` component retrieve the suggestion list from `bean.suggestions`.

7.1.2. Submission modes

Use the `mode` attribute to determine how the suggestion list is requested:

- The `client` setting pre-loads data to the client and uses the input to filter the possible suggestions.
- The `ajax` setting fetches suggestions with every input change using Ajax requests.
- The `lazyClient` setting pre-loads data to the client and uses the input to filter the possible suggestions. The filtering does not start until the input length matches a minimum value. Set the minimum value with the `minChars` attribute.
- The `cachedAjax` setting pre-loads data via Ajax requests when the input length matches a minimum value. Set the minimum value with the `minChars` attribute. All suggestions are handled on the client until the input prefix is changed, at which point a new request is made based on the new input prefix.

7.1.3. Interactivity options

Users can type into the 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.

7.1.4. Customizing the filter in `client` and `lazyClient` modes

The `<rich:autocomplete>` component uses the JavaScript `startsWith()` method to create the list of suggestions. The filtering is performed on the client side. Alternatively, use the `clientFilter` attribute to specify a custom filtering function. The custom function must accept two parameters: the `subString` parameter is the filtering value as typed into the text box by the user, and the `value` parameter is an item in the list of suggestions against which the `subString` must be checked. Each item is iterated through and passed to the function as the `value` parameter. The custom function must return a boolean value indicating whether the passed item meets the conditions of the filter, and the suggestion list is constructed from successful items.

Example 7.2. Customizing the filter

This example demonstrates how to use a custom filter with the `clientFilter` attribute. The custom filter determines if the sub-string is contained anywhere in the suggestion item, instead of just at the start.

```
<script>
function customFilter(subString, value){
    if(subString.length>=1) {
        if(value.indexOf(subString)!=-1)
            return true;
        }else return false;
    };
</script>
<h:form>
    <rich:autocomplete mode="client" minChars="0" autofill="false"
        clientFilter="customFilter"
        autocompleteMethod="#{autocompleteBean.autocomplete}" />
</h:form>
```

7.1.5. JavaScript API

The `<rich:autocomplete>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`getValue()`

Get the current value of the text field.

`setValue(newValue)`

Set the value of the text field to the *newValue* string passed as a parameter.

`showPopup()`

Show the pop-up list of completion values.

`hidePopup()`

Hide the pop-up list.

7.1.6. 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`

7.1.7. Style classes and skin parameters

Table 7.1. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-au-fnt</code> This class defines styles for the auto-complete box font.	<code>generalTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
<code>.rf-au-inp</code> This class defines styles for the auto-complete input box.	<code>controlBackgroundColor</code>	background-color
<code>.rf-au-fld</code> This class defines styles for the auto-complete field.	<code>panelBorderColor</code>	border-color
	<code>controlBackgroundColor</code>	background-color
<code>.rf-au-fld-btn</code> This class defines styles for a button in the auto-complete field.	No skin parameters.	
<code>.rf-au-btn</code> This class defines styles for the auto-complete box button.	<code>headerBackgroundColor</code>	background-color
	<code>panelBorderColor</code>	border-left-color
<code>.rf-au-btn-arrow</code> This class defines styles for the button arrow.	No skin parameters.	
<code>.rf-au-btn-arrow-dis</code> This class defines styles for the button arrow when it is disabled.	No skin parameters.	
<code>.rf-au-lst-scr1</code> This class defines styles for the scrollbar in the auto-complete list.	No skin parameters.	
<code>.rf-au-itm</code> This class defines styles for an item in the auto-complete list.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-au-itm-sel</code> This class defines styles for a selected item in the auto-complete list.	<code>headerBackgroundColor</code>	background-color
	<code>generalTextColor</code>	border-color
<code>.rf-au-shdw</code> This class defines styles for the auto-complete box shadow.	No skin parameters.	
<code>.rf-au-shdw-t</code> , <code>.rf-au-shdw-l</code> , <code>.rf-au-shdw-r</code> , <code>.rf-au-shdw-b</code> These classes define styles for the top, left, right, and bottom part of the auto-complete box shadow.	No skin parameters.	
<code>.rf-au-tbl</code> This class defines styles for a table in the auto-complete box.	No skin parameters.	

7.2. <rich:calendar>

The <rich:calendar> component allows the user to enter a date and time through an in-line or pop-up calendar. The pop-up calendar can navigate through months and years, and its look and feel can be highly customized.



Figure 7.2. <rich:calendar>

7.2.1. Basic usage

Basic usage of the `<rich:calendar>` component requires only the `value` attribute, which holds the currently selected date. [Example 7.3, “Basic usage”](#) shows a basic declaration, with the value pointing to a bean property. The bean property holds the selected date.

Example 7.3. Basic usage

```
<rich:calendar value="#{bean.dateTest}" />
```

7.2.2. Behavior and appearance

The `<rich:calendar>` component is presented as a pop-up by default, appearing as a text field with a button to expand the full pop-up calendar. To render the calendar in-line on the page instead, set `popup="false"`. This displays the full calendar without the text field and display button.

To add keyboard support for manual input, set `enableManualInput="true"`. To disable the calendar from any user input, set `disabled="true"`.

To change the appearance of the display button from the standard calendar icon, use the `buttonIcon` and `buttonDisabledIcon` attributes to replace the icon with a specified file. Alternatively, use the `buttonLabel` attribute to display text on the button without an icon. If `buttonLabel` is specified then both the `buttonIcon` and `buttonDisabledIcon` attributes are ignored. To hide the text field box, set `showInput="false"`.

The calendar features a **Today** button for locating today's date on the calendar. This can be set to three different values using the `todayControlMode` attribute:

- `hidden`, which does not display the button;
- `select`, the default setting, which scrolls the calendar to the current month and selects the date; and
- `scroll`, which scrolls the calendar to the month but does not select the date.
- `inactive`, which displays the date but performs no action when clicked.

To make the entire calendar read-only, set `readonly="true"`. This allows months and years to be browsed through with the arrow controls, but dates and times cannot be selected.

7.2.3. Time of day

The `<rich:calendar>` component can additionally allow a time of day to be specified with the date. After selecting a date the option to set a time becomes available. The default time can be set with the `defaultTime` attribute. If the time is altered and a new date is selected, it will not reset unless `resetTimeOnDateSelect="true"` is specified.

The date selection feature is activated if the time is present in the `datePattern` attribute for the calendar.



Support for seconds

In RichFaces 4, the `<rich:calendar>` component supports times that include seconds. Previous versions of RichFaces only supported hours and minutes.

7.2.4. Localization and formatting

Date and time strings can be formatted in a set pattern. Use standard locale formatting strings specified by *ISO 8601* (for example, `d/M/yy HH:mm a`) with the `datePattern` attribute to format date and time strings.

To set the locale of the calendar, use the `locale` attribute. The calendar will render month and day names in the relevant language. For example, to set the calendar to the US locale, specify `locale="en/US"`.

Use an application resource bundle to localize the calendar control labels. Define the following strings in the resource bundle:

- The `RICH_CALENDAR_APPLY_LABEL` string is the label for the **Apply** button.
- The `RICH_CALENDAR_TODAY_LABEL` string is the label for the **Today** button.
- The `RICH_CALENDAR_CLOSE_LABEL` string is the label for the **Close** button.
- The `RICH_CALENDAR_OK_LABEL` string is the label for the **OK** button.
- The `RICH_CALENDAR_CLEAN_LABEL` string is the label for the **Clean** button.
- The `RICH_CALENDAR_CANCEL_LABEL` string is the label for the **Cancel** button.

Alternatively, use the `org.richfaces.renderkit.calendar` resource bundle with Java Archive files (JARs) defining the same properties.

7.2.5. Using a data model

The look and feel of the `<rich:calendar>` component can be customized through the use of a data model on the server side. The component supports two different ways of loading data from the server side through defining the `mode` attribute.

When the `mode` attribute is not specified, the component uses the `client` mode. The `client` mode loads an initial portion of data within a set date range. The range can be defined by using the `preloadDateRangeBegin` and `preloadDateRangeEnd` attributes. Additional data requests for months outside the range are not sent.

Alternatively, with `mode="ajax"` the `<rich:calendar>` requests portions of data from the data model every time the month is switched. The data model can be defined through the `dataModel` attribute, which points to an object that implements the `CalendarDataModel` interface. If the `dataModel` attribute is not defined or has a value of `null`, the `ajax` mode functions the same as the `client` mode.

7.2.6. Client-side customization

Instead of using a data model, the `<rich:calendar>` component can be customized on the client-side using JavaScript. Use the `dayClassFunction` attribute to reference the function that determines the CSS style class for each day cell. Use the `dayDisableFunction` to reference the function that enables or disables a day cell. [Example 7.4, “Client-side customization”](#) demonstrates how client-side customization can be used to style different days in a calendar.

Example 7.4. Client-side customization

```
<style>
    .everyThirdDay {
        background-color: gray;
    }
    .weekendBold {
        font-weight: bold;
        font-style: italic;
    }
</style>
<script type="text/javascript">
    var curDt = new Date();
    function disablementFunction(day){
        if (day.isWeekend) return false;
        if (curDt==undefined){
            curDt = day.date.getDate();
        }
        if (curDt.getTime() - day.date.getTime() < 0) return true;
        else return false;
    }
    function disabledClassesProv(day){
        if (curDt.getTime() - day.date.getTime() >= 0) return 'rf-ca-boundary-
dates';
        var res = '';
        if (day.isWeekend) res+='weekendBold ';
        if (day.day%3==0) res+='everyThirdDay';
        return res;
    }
</script>
<rich:calendar dayDisableFunction="disablementFunction"
                dayClassFunction="disabledClassesProv"
                boundaryDatesMode="scroll" />
```

7.2.7. JavaScript API

The `<rich:calendar>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`showPopup()`

Expand the pop-up calendar element.

`hidePopup()`

Collapse the pop-up calendar element.

`switchPopup()`

Invert the state of the pop-up calendar element.

`getValue()`

Return the selected date value of the calendar.

`getValueAsString()`

Return the selected date value of the calendar as a formatted string.

`setValue(newValue)`

Set the selected date value to the *newValue* date passed as a parameter. If the new date is not in the currently displayed month, a request is performed to display the correct month.

`resetValue()`

Clear the selected date value.

`today()`

Select today's date.

`getCurrentMonth()`

Return the number of the month currently being displayed.

`getCurrentYear()`

Return the number of the year currently being displayed.

`showSelectedDate()`

Show the calendar month that contains the currently selected date.

`showDateEditor()`

Show the date editor pop-up.

`hideDateEditor()`

Hide the date editor pop-up.

`showTimeEditor()`

Show the time editor pop-up.

`hideTimeEditor()`

Hide the time editor pop-up.

7.2.8. Reference data

- *component-type*: `org.richfaces.calendar`
- *component-class*: `org.richfaces.component.html.HtmlCalendar`
- *component-family*: `org.richfaces.calendar`
- *renderer-type*: `org.richfaces.renderkit.calendarRenderer`
- *tag-class*: `org.richfaces.taglib.calendarTag`

7.2.9. Style classes and skin parameters

Table 7.2. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-cal-extr</code> This class defines the styles for a pop-up calendar exterior.	<code>panelBorderColor</code>	<code>border-color</code>
<code>.rf-cal-btn</code> This class defines styles for a calendar button.	No skin parameters.	
<code>.rf-cal-hdr</code> This class defines the styles for a calendar header.	<code>panelBorderColor</code>	<code>border-bottom-color</code>
	<code>additionalBackgroundColor</code>	<code>background-color</code>
	<code>generalSizeFont</code>	<code>font-size</code>
	<code>generalFamilyFont</code>	<code>font-family</code>
<code>.rf-cal-hdr-optnl</code> This class defines the styles for an optional header.	<code>panelBorderColor</code>	<code>border-bottom-color</code>
	<code>additionalBackgroundColor</code>	<code>background-color</code>
	<code>generalSizeFont</code>	<code>font-size</code>
	<code>generalFamilyFont</code>	<code>font-family</code>
<code>.rf-cal-hdr-month</code> This class defines the styles for the month header.	<code>headerBackgroundColor</code>	<code>background-color</code>
	<code>headerSizeFont</code>	<code>font-size</code>
	<code>headerFamilyFont</code>	<code>font-family</code>
	<code>headerWeightFont</code>	<code>font-weight</code>
	<code>headerTextColor</code>	<code>color</code>
<code>.rf-cal-ftr</code> This class defines the styles for a calendar footer.	<code>panelBorderColor</code>	<code>border-right-color, border-bottom-color</code>
	<code>additionalBackgroundColor</code>	<code>background</code>
	<code>generalSizeFont</code>	<code>font-size</code>

Class (selector)	Skin Parameters	Mapped CSS properties
	<i>generalFamilyFont</i>	font-family
.rf-cal-ftr-optnl This class defines the styles for an optional footer.	<i>panelBorderColor</i>	border-right-color, border-bottom-color
	<i>additionalBackgroundColor</i>	background
	<i>generalSizeFont</i>	font-size
	<i>generalFamilyFont</i>	font-family
.rf-cal-tl This class defines the styles for calendar toolbars.	<i>headerBackgroundColor</i>	background-color
	<i>headerSizeFont</i>	font-size
	<i>headerFamilyFont</i>	font-family
	<i>headerWeightFont</i>	font-weight
	<i>headerTextColor</i>	color
.rf-cal-tl-ftr This class defines the styles for a toolbar item in the calendar footer.	<i>additionalBackgroundColor</i>	background
	<i>generalSizeFont</i>	font-size
	<i>generalFamilyFont</i>	font-family
.rf-cal-tl-btn This class defines styles for a toolbar button.	No skin parameters.	
.rf-cal-tl-btn-dis This class defines styles for a disabled toolbar button.	No skin parameters.	
.rf-cal-tl-btn-hov This class defines the styles for toolbar items when it is hovered over with the mouse cursor.	<i>calendarWeekBackgroundColor</i>	background-color
	<i>generalTextColor</i>	color
	<i>tableBackgroundColor</i>	border-color
	<i>panelBorderColor</i>	border-right-color, border-bottom-color
.rf-cal-tl-btn-press This class defines the styles for toolbar items when it is pressed.	<i>panelBorderColor</i>	border-color
	<i>panelBorderColor</i>	border-right-color, border-bottom-color
.rf-cal-tl-close This class defines styles for a Close button in a toolbar.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-cal-c This class defines the styles for regular calendar cells.	<i>panelBorderColor</i>	border-bottom-color, border-right-color
	<i>tableBackgroundColor</i>	background-color
	<i>generalSizeFont</i>	font-size
	<i>generalFamilyFont</i>	font-family
.rf-cal-c-cnt This class defines styles for the content of a cell.	No skin parameters.	
.rf-cal-today This class defines the styles for the cell representing today's date.	<i>calendarCurrentBackgroundColor</i>	background-color
	<i>calendarCurrentTextColor</i>	color
.rf-cal-sel This class defines the styles for the selected day.	<i>headerBackgroundColor</i>	background-color
	<i>headerTextColor</i>	color
.rf-cal-hov This class defines the styles for a cell when it is hovered over with the mouse cursor.	<i>calendarSpecBackgroundColor</i>	background-color
	<i>calendarSpecTextColor</i>	color
.rf-cal-week This class defines the styles for week numbers.	<i>panelBorderColor</i>	border-bottom-color, border-right-color
	<i>calendarWeekBackgroundColor</i>	background-color
	<i>generalSizeFont</i>	font-size
	<i>generalFamilyFont</i>	font-family
.rf-cal-holiday This class defines the styles for weekends and holidays.	<i>calendarHolidaysBackgroundColor</i>	background-color
	<i>calendarHolidaysTextColor</i>	color
.rf-cal-boundary-day This class defines styles for an active boundary button.	No skin parameters.	
.rf-cal-sp-inp This class defines the styles for a spinner input field in the pop-up element for time selection.	<i>buttonSizeFont</i>	font-size
	<i>buttonFamilyFont</i>	font-family

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-cal-sp-inp-cntr This class defines the styles for a wrapper <code><td></code> element for a spinner input field in the pop-up element for time selection.	<i>controlBackgroundColor</i>	background-color
	<i>panelBorderColor</i>	border-color
	<i>subBorderColor</i>	border-right-color, border-bottom-color
.rf-cal-sp-btn This class defines the styles for a wrapper <code><td></code> element for spinner buttons in the pop-up element for time selection.	<i>headerBackgroundColor</i>	background-color, border-color
.rf-cal-sp-up This class defines styles for the Up spinner button.	No skin parameters.	
.rf-cal-sp-down This class defines styles for the Down spinner button.	No skin parameters.	
.rf-cal-sp-press This class defines styles for a spinner button when it is pressed.	No skin parameters.	
.rf-cal-edtr-shdw This class defines the styles for the calendar editor shadow.	<i>tableBackgroundColor</i>	background
.rf-cal-edtr-layout-shdw This class defines the styles for the layout shadow of a calendar editor.	<i>shadowBackgroundColor</i>	background-color
.rf-cal-edtr-btn This class defines styles for a button in the calendar editor.	No skin parameters.	
.rf-cal-edtr-btn-over This class defines the styles for the calendar editor button when it is	<i>panelBorderColor</i>	border-color
	<i>calendarSpecBackgroundColor</i>	background

Class (selector)	Skin Parameters	Mapped CSS properties
hovered over with the mouse cursor.		
.rf-cal-edtr-btn-sel This class defines the styles for the calendar editor button when it is selected.	<i>calendarCurrentBackgroundColor</i>	background-color
	<i>calendarCurrentTextColor</i>	color
.rf-cal-edtr-tl-over This class defines the styles for a toolbar item in the calendar editor when it is hovered over with the mouse cursor.	<i>additionalBackgroundColor</i>	background
	<i>tableBackgroundColor</i>	border-color
	<i>panelBorderColor</i>	border-right-color, border-bottom-color
.rf-cal-edtr-tl-press This class defines the styles for a toolbar item in the calendar editor when it is pressed.	<i>additionalBackgroundColor</i>	background
	<i>panelBorderColor</i>	border-color
	<i>tableBackgroundColor</i>	border-right-color, border-bottom-color
.rf-cal-time-inp This class defines styles for the time input field.	No skin parameters.	
.rf-cal-time-btn This class defines the styles for a button in the pop-up element for the calendar's time section.	<i>tableBackgroundColor</i>	border-color
	<i>panelBorderColor</i>	border-right-color, border-bottom-color
.rf-cal-time-btn-press This class defines the styles for a pressed button in the pop-up element for the calendar's time section.	<i>tableBackgroundColor</i>	border-right-color, border-bottom-color
	<i>panelBorderColor</i>	border-color
	<i>calendarWeekBackgroundColor</i>	background-color
.rf-cal-timepicker-cnt This class defines the styles for the content of the pop-up element during time selection.	<i>panelBorderColor</i>	border-color
	<i>additionalBackgroundColor</i>	background
	<i>generalSizeFont</i>	font-size
	<i>generalFamilyFont</i>	font-family
.rf-cal-timepicker-inp This class defines the styles for an input field in the time picker.	<i>generalSizeFont</i>	font-size
	<i>generalFamilyFont</i>	font-family

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-cal-timepicker-ok This class defines styles for the OK button in the time picker.	No skin parameters.	
.rf-cal-timepicker-cancel This class defines styles for the Cancel button in the time picker.	No skin parameters.	
.rf-cal-monthpicker-cnt This class defines the styles for the content of the pop-up element during month or year selection.	panelBorderColor	border-color
	tableBackgroundColor	background
	generalSizeFont	font-size
	generalFamilyFont	font-family
.rf-cal-monthpicker-ok This class defines the styles for the OK button for the month picker.	additionalBackgroundColor	background
	panelBorderColor	border-top-color
.rf-cal-monthpicker-cancel This class defines the styles for the Cancel button for the month picker.	additionalBackgroundColor	background
	panelBorderColor	border-top-color
.rf-cal-monthpicker-split This class defines the styles for the splitter in the month picker.	panelBorderColor	border-right-color

7.3. <rich:fileUpload>

The <rich:fileUpload> component allows the user to upload files to a server. It features multiple uploads, progress bars, restrictions on file types, and restrictions on sizes of the files to be uploaded.

7.3.1. Basic usage

Basic usage requires the `fileUploadListener` attribute. Use the attribute to reference a listener function on the server side after each file is uploaded. The listener should process files as required, such as storing them in the `session/db/filesystem/` directory. The component itself does not store uploaded files, so if the listener is not implemented they are not stored anywhere.

Example 7.5. Basic usage

```
<rich:fileUpload fileUploadListener="#{bean.listener}" />
```

7.3.2. Upload settings

Files are uploaded to either the temporary folder (different for each operating system) or to RAM (random-access memory), depending on the value of the *org.richfaces.fileUpload.createTempFiles* context parameter of the *web.xml* settings file for the project. If the parameter is set to *true*, the files are uploaded to the temporary folder.

To limit the maximum size of the uploaded files, define the byte size with the *org.richfaces.fileUpload.maxRequestSizes* context parameter of the *web.xml* settings file for the project.

7.3.3. Interactivity options

The text labels used in the component can be completely customized. Labels for the various controls of the component can be set using the following parameters:

addLabel

The *addLabel* parameter sets the label for the **Add** button.

clearAllLabel

The *clearAllLabel* parameter sets the label for the **Clear All** button.

clearLabel

The *clearLabel* parameter sets the label for the **Clear** button.

uploadLabel

The *uploadLabel* parameter sets the label for the **Upload** button.

The `<rich:fileUpload>` component provides a built-in progress bar to indicate the progress of each file that is uploaded. This progress bar can be replaced with a `<rich:progressBar>` component added to the *progress* facet. Refer to [Section 12.3, “<rich:progressBar>”](#) for details on the `<rich:progressBar>` component.

To disable the `<rich:fileUpload>` component, use the *disabled* attribute.

7.3.4. `<rich:fileUpload>` client-side events

There are a number of event handlers specific to the `<rich:fileUpload>` component:

- *filesubmit* is triggered before a file is uploaded.
- *uploadcomplete* is triggered after all files in the list have finished uploading.

7.3.5. Reference data

- *component-type*: org.richfaces.FileUpload
- *component-class*: org.richfaces.component.UIFileUpload
- *component-family*: org.richfaces.FileUpload
- *renderer-type*: org.richfaces.FileUploadRenderer

7.3.6. Style classes and skin parameters

Table 7.3. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-fu This class defines styles for the file upload control.	<i>generalBackgroundColor</i>	background-color
	<i>panelBorderColor</i>	border-color
.rf-fu-hdr This class defines styles for the header of the file upload control.	<i>headerBackgroundColor</i>	background-color, border-color
.rf-fu-lst This class defines styles for lists in the file upload control.	No skin parameters.	
.rf-fu-cntr-hdn This class defines styles for the file upload container when it is hidden.	No skin parameters.	
.rf-fu-btns-lft, .rf-fu-btns-rgh These classes define styles for buttons on the left and right of the file upload control.	No skin parameters.	
.rf-fu-btn-add This class defines styles for the Add button in the file upload control.	<i>trimColor</i>	background-color
	<i>panelBorderColor</i>	border-color
.rf-fu-btn-cnt-add This class defines styles for the content of the Add	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size

Class (selector)	Skin Parameters	Mapped CSS properties
button in the file upload control.		
.rf-fu-btn-add-dis This class defines styles for the Add button in the file upload control when it is disabled.	<i>tableFooterBackgroundColor</i>	background-color
	<i>tableFooterBackgroundColor</i>	border-color
.rf-fu-btn-cnt-add-dis This class defines styles for the content of the Add button in the file upload control when it is disabled.	<i>tabDisabledTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
.rf-fu-btn-upl This class defines styles for the Upload button in the file upload control.	<i>trimColor</i>	background-color
	<i>panelBorderColor</i>	border-color
.rf-fu-btn-cnt-upl This class defines styles for the content of the Upload button in the file upload control.	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
.rf-fu-btn-clr This class defines styles for the Clear button in the file upload control.	<i>trimColor</i>	background-color
	<i>panelBorderColor</i>	border-color
.rf-fu-btn-cnt-clr This class defines styles for the content of the Clear button in the file upload control.	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
.rf-fu-itm This class defines styles for an item in the file upload control.	<i>panelBorderColor</i>	border-bottom-color
.rf-fu-itm-lft, .rf-fu-itm-rgh These classes define styles for items on the left and right of the file upload control.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-fu-itm-lbl This class defines styles for the label of an item in the file upload control.	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
.rf-fu-itm-st This class defines styles for the status of an item in the file upload control.	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
.rf-fu-itm-lnk This class defines styles for a link item in the file upload control.	<i>generalLinkColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
.rf-fu-inp This class defines styles for the input field in the file upload control.	No skin parameters.	
.rf-fu-inp-cntr This class defines styles for the input field container in the file upload control.	No skin parameters.	

7.4. <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.

7.4.1. Basic usage

Basic usage requires the `value` attribute to point to the expression for the current value of the component. Validation and conversion rules for the JSF `UIInput` control apply as usual.

7.4.2. Interactivity options

When in the initial *view* state, the starting label can be set using the `defaultLabel` attribute. Alternatively, if the initial value is already set through the `value` attribute, this is displayed instead.

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 7.6, "Default label and value"](#).

Example 7.6. 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 in multiple ways:

- By default, pressing the **Enter** key will confirm and save the input.
- If `showControls="true"` is set, buttons for confirming or canceling are added to the component.
- If `saveOnBlur="true"` is set, the input is saved on the component's blur event.

Pressing the **Esc** key cancels editing in all cases.

7.4.3. JavaScript API

The `<rich:inplaceInput>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`getValue()`

Get the current value of the input control.

`setValue(newValue)`

Set the value of the input control to the *newValue* string passed as a parameter.

`isEditState()`

Returns `true` if the control is currently in the *edit* state, or `false` if the control is currently in the *view* state.

`isValueChanged()`

Returns `true` if the control's value has been changed from the default.

`save()`

Saves the current item as the control's value.

`cancel()`

Cancel editing the value.

`getInput()`

Return the DOM element for the input.

7.4.4. 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`

7.4.5. Style classes and skin parameters

Table 7.4. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-ii-d-s</code> This class defines styles for the in-place input when it is in the default state.	<i>editorBackgroundColor</i>	background-color
	<i>generalTextColor</i>	border-bottom-color
<code>.rf-ii-e-s</code> This class defines styles for the in-place input when it is in the editing state.	No skin parameters.	
<code>.rf-ii-c-s</code> This class defines styles for the in-place input when it is in the changed state.	No skin parameters.	
<code>.rf-ii-dis-s</code> This class defines styles for the in-place input when it is in the disabled state.	No skin parameters.	
<code>.rf-ii-fld</code> This class defines styles for the in-place input field.	<i>editBackgroundColor</i>	background-color, border-bottom-color
	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
<code>.rf-ii-dflt-lbl</code> This class defines styles for the default label of the in-place input.	No skin parameters.	
<code>.rf-ii-edit</code> This class defines styles for the in-place input when it is being edited.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-ii-btn</code> This class defines styles for the buttons for the in-place input.	<code>tabBackgroundColor</code>	background-color
	<code>panelBorderColor</code>	border-color
<code>.rf-ii-btn-p</code> This class defines styles for the buttons for the in-place input when they are pressed.	<code>tabBackgroundColor</code>	background-color
	<code>panelBorderColor</code>	border-color
<code>.rf-ii-btn-set</code> , <code>.rf-ii-btn-prepos</code> , <code>.rf-ii-btn-pos</code> These classes define the positioning of the buttons.	No skin parameters.	
<code>.rf-ii-btn-shdw</code> This class defines styles for the button shadows for the in-place input.	No skin parameters.	
<code>.rf-ii-btn-shdw-t</code> , <code>.rf-ii-btn-shdw-b</code> , <code>.rf-ii-btn-shdw-l</code> , <code>.rf-ii-btn-shdw-r</code> These classes define the top, bottom, left, and right edge of the button shadows.	No skin parameters.	
<code>.rf-ii-none</code> This class defines styles for the in-place input when it cannot be edited.	No skin parameters.	

7.5. <rich:inplaceSelect>

The `<rich:inplaceSelect>` component is similar to the `<rich:inplaceInput>` component, except that the `<rich:inplaceSelect>` component uses a drop-down selection box to enter text instead of a regular text field. Changes can be rendered either in-line or for the whole block, and inputs can be focused with keyboard navigation. The component is based on the JSF `UISelectOne` component, so all the standard rules for value definition, processing, conversion, and validation apply.

The component has three functional states:

- When in the *view* state, the component displays its initial setting, such as "click to edit".
- When in the *edit* state, the user can select a value from a drop-down list.
- When in the *changed* state, the new value for the component has been confirmed, but it can be edited again if required.

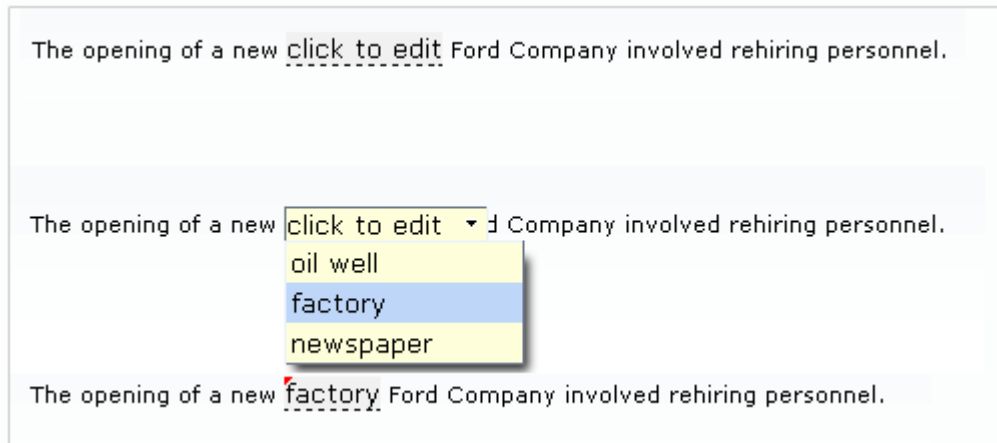


Figure 7.3. `<rich:inplaceSelect>`

7.5.1. Basic usage

Basic usage requires the `value` attribute to point to the expression for the current value of the component and a list of items. The list of items can be defined using the JSF components `<f:selectItem/>` and `<f:selectItems/>`.

Example 7.7. Defining list items for `<rich:inplaceSelect>`

```
<rich:inplaceSelect value="#{bean.inputValue}" defaultLabel="click to edit" >
  <f:selectItems value="#{bean.selectItems}" />
  <f:selectItem itemValue="1" itemLabel="Item 1" />
  <f:selectItem itemValue="2" itemLabel="Item 2" />
  <f:selectItem itemValue="3" itemLabel="Item 3" />
  <f:selectItem itemValue="4" itemLabel="Item 4" />
</rich:comboBox>
```

7.5.2. Interactivity options

When in the initial *view* state, the starting label can be set using the `defaultLabel` attribute, such as `defaultLabel="click to edit"`. Alternatively, if the initial value is already set through the `value` attribute, this is displayed instead.

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. When switching to *edit* mode,

the drop-down list of possible values will automatically be displayed; this can be deactivated by setting `openOnEdit="false"`.

Once a new value for the control is saved, the state switches to the "changed" state. Saving a new value for the control can be performed in a number of ways:

- Once the user selects an item from the drop-down list, the item is saved as the new control value. This is the default setting. If `saveOnSelect="false"` is set, the component applies the selected item but remains in the *edit* state so a different selection could be chosen. The value is then applied when the **Enter** key is pressed.
- If `saveOnBlur="true"` is set, the selected item is saved as the new control value when the control loses focus.
- If `showControls="true"` is set, buttons are added to the control to confirm or cancel the selection. The new control value is only saved once the user confirms the selection using the button.

Pressing the **Esc** key cancels editing in all cases.

7.5.3. JavaScript API

The `<rich:inplaceSelect>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`getValue()`

Get the current value of the select control.

`setValue(newValue)`

Set the value of the select control to the *newValue* string passed as a parameter.

`isEditState()`

Returns `true` if the control is currently in the *edit* state, or `false` if the control is currently in the *view* state.

`isValueChanged()`

Returns `true` if the control's value has been changed from the default.

`save()`

Saves the current item as the control's value.

`cancel()`

Cancel editing the value.

`getInput()`

Return the input entered into the control by the user.

`getLabel()`

Return the default label of the control.

`setLabel(newLabel)`

Set the default label of the control to the *newLabel* string passed as a parameter.

`showPopup()`

Show the pop-up list of possible values.

`hidePopup()`

Hide the pop-up list.

7.5.4. Reference data

- *component-type*: `org.richfaces.inplaceSelect`
- *component-class*: `org.richfaces.component.html.HtmlInplaceSelect`
- *component-family*: `org.richfaces.inplaceSelect`
- *renderer-type*: `org.richfaces.renderkit.inplaceSelectRenderer`
- *tag-class*: `org.richfaces.taglib.inplaceSelectTag`

7.5.5. Style classes and skin parameters

Table 7.5. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-is-d-s</code> This class defines styles for the in-place select when it is in the default state.	<i>editorBackgroundColor</i>	background-color
	<i>generalTextColor</i>	border-bottom-color
<code>.rf-is-e-s</code> This class defines styles for the in-place select when it is in the editing state.	No skin parameters.	
<code>.rf-is-c-s</code> This class defines styles for the in-place select when it is in the changed state.	No skin parameters.	
<code>.rf-is-dis-s</code> This class defines styles for the in-place select when it is in the disabled state.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-is-flt</code> This class defines styles for the in-place select field.	<code>editBackgroundColor</code>	background
	<code>generalTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
<code>.rf-is-opt</code> This class defines styles for an option for the in-place select.	<code>generalTextColor</code>	border-color
<code>.rf-is-sel</code> This class defines styles for the selected option of the in-place select.	<code>generalTextColor</code>	border-color
<code>.rf-is-lbl</code> This class defines styles for the label of the in-place select.	No skin parameters.	
<code>.rf-is-dflt-lbl</code> This class defines styles for the default label of the in-place select.	No skin parameters.	
<code>.rf-is-edit</code> This class defines styles for the in-place select when it is being edited.	No skin parameters.	
<code>.rf-is-btn</code> This class defines styles for the buttons for the in-place select.	<code>tabBackgroundColor</code>	background-color
	<code>panelBorderColor</code>	border-color
<code>.rf-is-btn-p</code> This class defines styles for the buttons for the in-place select when they are pressed.	<code>tabBackgroundColor</code>	background-color
	<code>panelBorderColor</code>	border-color
<code>.rf-is-btn-set</code> , <code>.rf-is-btn-prepos</code> , <code>.rf-is-btn-pos</code> These classes define the positioning of the buttons.	No skin parameters.	

Class (selector)	Skin Parameters		Mapped CSS properties
<code>.rf-is-lst-pos</code> This class defines the positioning of the list.	No skin parameters.		
<code>.rf-is-lst-dec</code> This class defines styles for a decreasing list for the in-place select.	<code>editBackgroundColor</code>		background-color
	<code>panelBorderColor</code>		border-color
<code>.rf-is-lst-scr1</code> This class defines styles for the list scrollbar.	No skin parameters.		
<code>.rf-is-shdw</code> This class defines styles for the in-place select shadow.	No skin parameters.		
<code>.rf-is-shdw-t, .rf-is-shdw-b, .rf-is-shdw-l, .rf-is-shdw-r</code> These classes define the top, bottom, left, and right edge of the in-place select shadows.	No skin parameters.		
<code>.rf-is-btn-shdw</code> This class defines styles for the button shadows for the in-place select.	No skin parameters.		
<code>.rf-is-none</code> This class defines styles for the in-place select when it cannot be edited.	No skin parameters.		

7.6. <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.

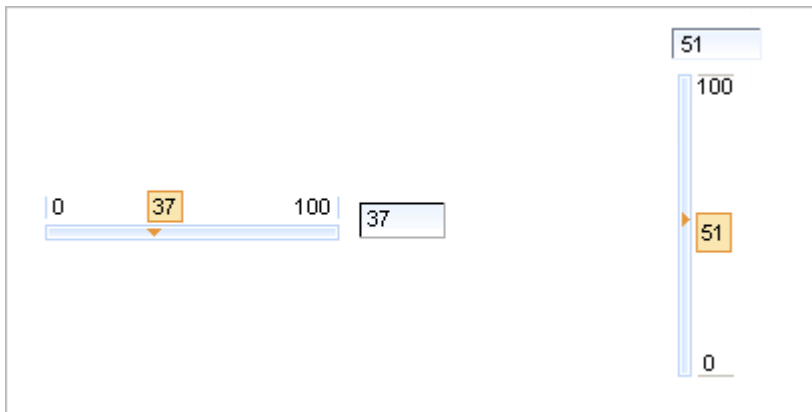


Figure 7.4. `<rich:inputNumberSlider>`

7.6.1. 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. Standard conversion and validation for the JSF `UIInput` component is applied.

7.6.2. 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.

7.6.3. JavaScript API

The `<rich:inputNumberSlider>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`getValue()`

Get the current value of the slider control.

`setValue(newValue)`

Set the value of the slider control to the `newValue` integer passed as a parameter.

`increase()`

Increase the value of the slider control by the gradient step amount.

`decrease()`

Decrease the value of the slider control by the gradient step amount.

7.6.4. 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`

7.6.5. Style classes and skin parameters

Table 7.6. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-insl</code> This class defines styles for the number slider itself.	No skin parameters.	
<code>.rf-insl-trc</code> This class defines styles for the number slider track.	<code>controlBackgroundColor</code>	background-color
	<code>panelBorderColor</code>	border-bottom-color
<code>.rf-insl-trc-cntr</code> This class defines styles for the container of the number slider track.	No skin parameters.	
<code>.rf-insl-mn</code> This class defines styles for the minimum label on the number slider.	<code>generalSizeFont</code>	font-size
	<code>generalFamilyFont</code>	font-family
	<code>generalTextColor</code>	color
	<code>panelBorderColor</code>	border-left-color
<code>.rf-insl-mx</code> This class defines styles for the maximum label on the number slider.	<code>generalSizeFont</code>	font-size
	<code>generalFamilyFont</code>	font-family
	<code>generalTextColor</code>	color
	<code>panelBorderColor</code>	border-right-color

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-insl-inp</code> This class defines styles for the input field on the number slider.	<code>generalSizeFont</code>	font-size
	<code>generalFamilyFont</code>	font-family
	<code>generalTextColor</code>	color
<code>.rf-insl-inp-cntr</code> This class defines styles for the container of the input field.	No skin parameters.	
<code>.rf-insl-hnd</code> This class defines styles for the handle on the number slider.	No skin parameters.	
<code>.rf-insl-hnd-cntr</code> This class defines styles for the container of the handle.	No skin parameters.	
<code>.rf-insl-hnd-sel</code> This class defines styles for the handle when it is selected.	No skin parameters.	
<code>.rf-insl-hnd-dis</code> This class defines styles for the handle when it is disabled.	No skin parameters.	
<code>.rf-insl-dec</code> , <code>.rf-insl-inc</code> These classes define styles for the step controls to decrease and increase the number.	No skin parameters.	
<code>.rf-insl-dec-sel</code> , <code>.rf-insl-inc-sel</code> These classes define styles for the step controls when they are selected.	No skin parameters.	
<code>.rf-insl-dec-dis</code> , <code>.rf-insl-inc-dis</code> These classes define styles for the step controls when they are disabled.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-insl-tt This class defines styles for the tool-tip on the number slider.	<i>generalSizeFont</i>	font-size
	<i>generalFamilyFont</i>	font-family
	<i>generalTextColor</i>	color
	<i>tipBorderColor</i>	border
	<i>tipBackgroundColor</i>	background-color

7.7. <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 7.5. <rich:inputNumberSpinner>

7.7.1. 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.

7.7.2. 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"`.

7.7.3. JavaScript API

The <rich:inputNumberSpinner> component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`getValue()`

Get the current value of the spinner control.

`setValue(newValue)`

Set the value of the spinner control to the *newValue* integer passed as a parameter.

`increase()`

Increase the value of the spinner control by the gradient step amount.

`decrease()`

Decrease the value of the spinner control by the gradient step amount.

7.7.4. 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`

7.7.5. Style classes and skin parameters

Table 7.7. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-insp</code> This class defines styles for the number spinner itself.	<code>panelBorderColor</code>	<code>border-color</code>
<code>.rf-insp-inp</code> This class defines styles for the input field on the number spinner.	<code>generalSizeFont</code>	<code>font-size</code>
	<code>generalFamilyFont</code>	<code>font-family</code>
	<code>generalTextColor</code>	<code>color</code>
	<code>controlBackgroundColor</code>	<code>background-color</code>
<code>.rf-insp-btns</code> This class defines styles for the buttons on the number spinner.	<code>headerBackgroundColor</code>	<code>background-color</code>
	<code>panelBorderColor</code>	<code>border-left-color</code>
<code>.rf-insp-dec</code> , <code>.rf-insp-inc</code> These classes define styles for the step controls to decrease and increase the number.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-insp-dec-dis,</code> <code>.rf-insp-inc-dis</code> These classes define styles for the step controls when they are disabled.	<code>.rf-</code> No skin parameters.	

7.8. <rich:select>

The <rich:select> component provides a drop-down list box for selecting a single value from multiple options. The <rich:select> component can be configured as a combo-box, where it will accept typed input. The component also supports keyboard navigation. The <rich:select> component functions similarly to the JSF `UISelectOne` component.



Figure 7.6. <rich:select>

7.8.1. Basic usage

Simple usage of the <rich:select> component requires the `value` attribute to store the selected value. Additionally, child tags to manage the list of selections are required. The child tags can either be a number of <f:selectItem> tags or a <f:selectItems> tag which points to a data model containing a list of selection items. The `value` attribute is used to store the current selection.

Example 7.8. Selection items

Using multiple <f:selectItem> tags

```
<rich:select>
  <f:selectItem itemValue="0" itemLabel="Option 1" />
  <f:selectItem itemValue="1" itemLabel="Option 2" />
  <f:selectItem itemValue="2" itemLabel="Option 3" />
  <f:selectItem itemValue="3" itemLabel="Option 4" />
  <f:selectItem itemValue="4" itemLabel="Option 5" />
</rich:select>
```

Using a single <f:selectItems> tag

```
<rich:select>
```

```
<f:selectItems value="#{bean.options}" />
</rich:select>
```

The arrow keys on a keyboard can be used to highlight different items in the list. If the control loses focus or the **Enter** key is pressed, the highlighted option is chosen as the value and the list is closed. Pressing the **Esc** key will close the list but not change the value.

7.8.2. Using manual input

The `<rich:select>` component allows the user to type into a text field to scroll through or filter the list. By default, the `<rich:select>` component functions as a drop-down list with no manual input. To add keyboard support for manual input, set `enableManualInput="true"`.

Once the user begins typing, the first available matching option is highlighted. If the typed text does not match any values in the list, no value is chosen and the drop-down list displays as empty. Other keyboard interaction remains the same as the basic drop-down list.

The standard JSF `<h:selectOne>` component does not offer this extended keyboard support. However, since the `<rich:select>` component is still based on the JSF `UISelectOne` component, it will not accept a value that does not match any items in the drop-down list. If an invalid value is entered, it is highlighted as erroneous and validation messages appear with the submission.

7.8.3. Advanced options

Use the `defaultLabel` attribute to set a place-holder label, such as `defaultLabel="select an option"`.

Server-side processing occurs in the same manner as for an `<h:selectOneMenu>` component. As such, custom objects used for selection items should use the same converters as for an `<h:selectOneMenu>` component.

7.8.4. JavaScript API

The `<rich:select>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`getValue()`

Get the current value of the text field.

`setValue(newValue)`

Set the value of the text field to the `newValue` string passed as a parameter.

`getLabel()`

Return the default label of the control.

`showPopup()`

Show the pop-up list of completion values.

`hidePopup()`

Hide the pop-up list.

7.8.5. Reference data

- *component-type*: `org.richfaces.select`
- *component-class*: `org.richfaces.component.html.HtmlSelect`
- *component-family*: `org.richfaces.select`
- *renderer-type*: `org.richfaces.renderkit.selectRenderer`
- *tag-class*: `org.richfaces.taglib.selectTag`

7.8.6. Style classes and skin parameters

Table 7.8. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-sel</code> This class defines styles for the select control itself.	No skin parameters.	
<code>.rf-sel-cntr</code> This class defines styles for the container of the select control.	<code>panelBorderColor</code>	<code>border-color</code>
<code>.rf-sel-inp</code> This class defines styles for the select control input field.	<code>controlBackgroundColor</code>	<code>background-color</code>
<code>.rf-sel-fld-err</code> This class defines styles for the input field when an error occurs.	No skin parameters.	
<code>.rf-sel-opt</code> This class defines styles for an option in the select control.	<code>generalTextColor</code>	<code>color</code>
	<code>generalSizeFont</code>	<code>font-size</code>
	<code>generalFamilyFont</code>	<code>font-family</code>
<code>.rf-sel-sel</code> This class defines styles for the selected option of the select control.	<code>generalTextColor</code>	<code>border-color</code>

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-sel-dflt-lbl This class defines styles for the default label of the select control.	No skin parameters.	
.rf-sel-btn This class defines styles for the button of the select control.	<i>headerBackgroundColor</i>	background-color
	<i>panelBorderColor</i>	border-left-color
.rf-sel-btn-arrow This class defines styles for the arrow on the button.	No skin parameters.	
.rf-sel-btn-dis This class defines styles for the button of the select control when it is disabled.	No skin parameters.	
.rf-sel-lst-scr1 This class defines styles for the list scrollbar.	No skin parameters.	
.rf-sel-shdw This class defines styles for the select control shadow.	No skin parameters.	
.rf-sel-shdw-t, .rf-sel-shdw-b, .rf-sel-shdw-l, .rf-sel-shdw-r These classes define the top, bottom, left, and right edge of the select control shadows.	No skin parameters.	

Panels

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

8.1. <rich:panel>

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

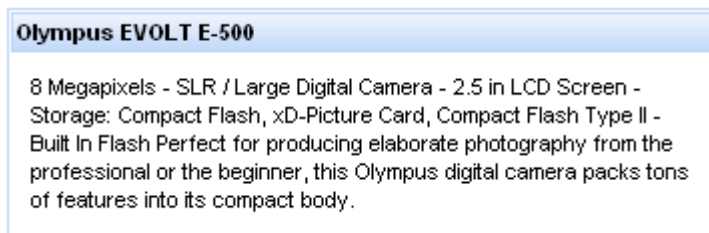


Figure 8.1. <rich:panel>

8.1.1. 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.

8.1.2. 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 8.1, “Adding a header”](#) demonstrates the two different approaches.

Example 8.1. 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.

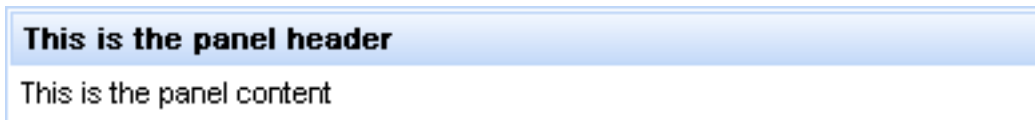


Figure 8.2. Adding a header

8.1.3. 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`

8.1.4. Style classes and skin parameters

Table 8.1. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-p</code> This class defines styles for the panel itself.	<i>generalBackgroundColor</i>	background-color
	<i>panelBorderColor</i>	color
<code>.rf-p-hdr</code> This class defines styles for the header of a panel.	<i>headerBackgroundColor</i>	background-color, border-color
	<i>headerTextColor</i>	color
	<i>headerSizeFont</i>	font-size
	<i>headerWeightFont</i>	font-weight
	<i>headerFamilyFont</i>	font-family
<code>.rf-p-b</code> This class defines styles for the body of a panel.	<i>generalTextColor</i>	color
	<i>generalSizeFont</i>	font-size
	<i>generalFamilyFont</i>	font-family

8.2. `<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 expand the selected one. Each panel contained in a `<rich:accordion>` component is a `<rich:accordionItem>` component.



Figure 8.3. A `<rich:accordion>` component containing three `<rich:accordionItem>` components

8.2.1. 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 8.2.8, “`<rich:accordionItem>`”](#) for details on the `<rich:accordionItem>` component.



Form elements required

All `<rich:tabPanel>` components should be wrapped in a form element when using either `ajax` or `server` mode, as usual for submitting components.

8.2.2. Switching panels

The `activeItem` attribute holds the active panel name. This name is a reference to the `name` identifier of the active child `<rich:accordionItem>` 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:accordionItem>` component causes the parent `<rich:accordion>` component to perform a common submission, completely refreshing the page. Only one panel at a time is rendered 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 rendered to the client side.

client

Activation of a `<rich:accordionItem>` component causes the parent `<rich:accordion>` component to perform updates on the client side. All the panels are rendered on the client side during the initial page render. JavaScript changes the styles such that one panel component becomes hidden while the other is shown.

8.2.3. `<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 `itemchange` event points to the function to perform when the switchable item is changed.
- The `beforeitemchange` event points to the function to perform when before the switchable item is changed.

8.2.4. `<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.

8.2.5. JavaScript API

The `<rich:accordion>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions, which are common to all switchable panels:

`getItems()`

Return an array of the items contained in the accordion control.

`getItemsNames()`

Return an array of the names of the items contained in the accordion control.

`switchToItem(itemName)`

Switch to and display the item identified by the `itemName` string passed as a parameter.

`firstItem(), prevItem(), nextItem(), lastItem()`

Switch to and display the first item, the previous item, the next item, or the last item.

8.2.6. 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`

8.2.7. Style classes and skin parameters

Table 8.2. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-ac</code> This class defines styles for the accordion control itself.	<code>panelBorderColor</code>	border-color
	<code>generalBackgroundColor</code>	background
<code>.rf-ac-itm-hdr</code> This class defines styles for the header of an accordion item.	<code>panelBorderColor</code>	border-bottom-color
	<code>headerBackgroundColor</code>	background-color
	<code>headerTextColor</code>	color
	<code>headerWeightFont</code>	font-weight
	<code>headerFamilyFont</code>	font-family
	<code>headerSizeFont</code>	font-size
<code>.rf-ac-itm-hdr-act</code> , <code>.rf-ac-itm-hdr-inact</code> These classes define styles for the header when the item is either active (expanded) or inactive (collapsed).	No skin parameters.	
<code>.rf-ac-itm-hdr-dis</code> This class defines styles for the header when it is disabled.	<code>tabDisabledTextColor</code>	color
<code>.rf-ac-itm-gr</code> This class defines styles for an item group.	No skin parameters.	
<code>.rf-ac-itm-cnt</code> This class defines styles for the content of an accordion item.	<code>panelBorderColor</code>	border-bottom-color
	<code>generalTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
<code>.rf-ac-itm-ico</code> This class defines styles for the item icon.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-ac-itm-exp-ico</code> This class defines styles for the expanded icon for an item.	No skin parameters.	
<code>.rf-ac-itm-ico-act, .rf-ac-itm-ico-inact</code> These classes define styles for the icon when the item is either active (expanded) or inactive (collapsed).	No skin parameters.	
<code>.rf-ac-itm-lbl</code> This class defines styles for the item label.	No skin parameters.	
<code>.rf-ac-itm-lbl-act, .rf-ac-itm-lbl-inact</code> These classes define styles for the label when the item is either active (expanded) or inactive (collapsed).	No skin parameters.	

8.2.8. <rich:accordionItem>

The `<rich:accordionItem>` component is a panel for use with the `<rich:accordion>` component. `<rich:accordionItem>` components can be added dynamically using iteration models with the `<c:forEach>` tag.

8.2.8.1. Basic usage

Basic usage of the `<rich:accordionItem>` component requires the `header` 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 `header` attribute. This would allow for additional styles and custom content to be applied to the tab.

8.2.8.2. <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 `enter` event points to the function to perform when the mouse enters the panel.

- The `leave` event points to the function to perform when the mouse leaves the panel.

8.2.8.3. 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`

8.3. <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.

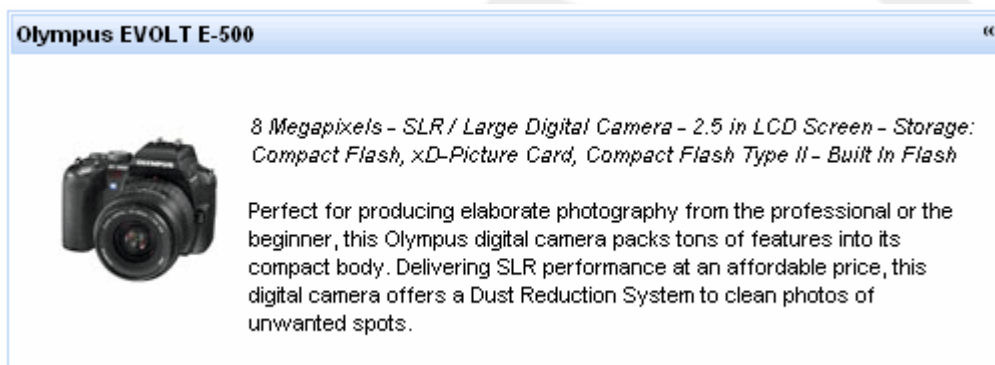


Figure 8.4. `<rich:collapsiblePanel>`

8.3.1. 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.

8.3.2. 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 refreshing the page. Only one panel at a time is rendered to the client side.

ajax

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

client

The `<rich:collapsiblePanel>` component changes the state on the client side without any additional requests being sent.

8.3.3. Appearance

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

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

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

- The `PanelToggleEvent` event occurs on the server side when the `<rich:collapsiblePanel>` component is expanded or collapsed in either the `ajax` or `server` modes. It can be processed using the `panelTogglerListener` attribute.

8.3.5. JavaScript API

The `<rich:collapsiblePanel>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`switchPanel()`

Switch the state of the collapsible panel (expanded or collapsed).

8.3.6. 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`

8.3.7. Style classes and skin parameters

Table 8.3. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-cp</code> This class defines styles for the collapsible panel itself.	<code>panelBorderColor</code>	color
	<code>generalBackgroundColor</code>	background
<code>.rf-cp-hdr</code> This class defines styles for the header of a collapsible panel.	<code>headerBackgroundColor</code>	background-color, border-color
	<code>headerTextColor</code>	color
	<code>headerWeightFont</code>	font-weight
	<code>headerFamilyFont</code>	font-family
	<code>headerSizeFont</code>	font-size
<code>.rf-cp-hdr-exp, .rf-cp-hdr-colps</code> These classes define styles for the header when the item is either expanded or collapsed.	No skin parameters.	
<code>.rf-cp-gr</code> This class defines styles for a collapsible panel group.	No skin parameters.	
<code>.rf-cp-b</code> This class defines styles for the body of a collapsible panel.	<code>generalTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
<code>.rf-cp-ico</code> This class defines styles for the panel icon.	No skin parameters.	
<code>.rf-cp-exp-ico</code> This class defines styles for the expanded icon for a panel.	No skin parameters.	
<code>.rf-cp-ico-exp, .rf-cp-ico-colps</code> These classes define styles for the icon when	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
the panel is either expanded or collapsed.		
.rf-cp-lbl This class defines styles for the panel label.	No skin parameters.	
.rf-cp-lbl-exp, .rf-cp-lbl-collapsed These classes define styles for the label when the panel is either expanded or collapsed.	No skin parameters.	

8.4. <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.

8.4.1. Basic usage

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

8.4.2. 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 ???.
- Using the rich:component function. For details on the function, refer to [Section 15.3, “rich:component”](#).

For explicit referencing when using the functions, the component can be given an id identifier.

Example 8.2, “<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 8.2. <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 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.

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`.

8.4.3. 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.

8.4.4. 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 `minWidth` 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`.



Embedded objects in the panel

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

However, 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>` or `<object>` tags are being used in the parent view. Do not set it to `true` for applications that do not require it.

8.4.5. 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 8.3](#), “Header and controls” demonstrates the use of the facets.

Example 8.3. 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>
```

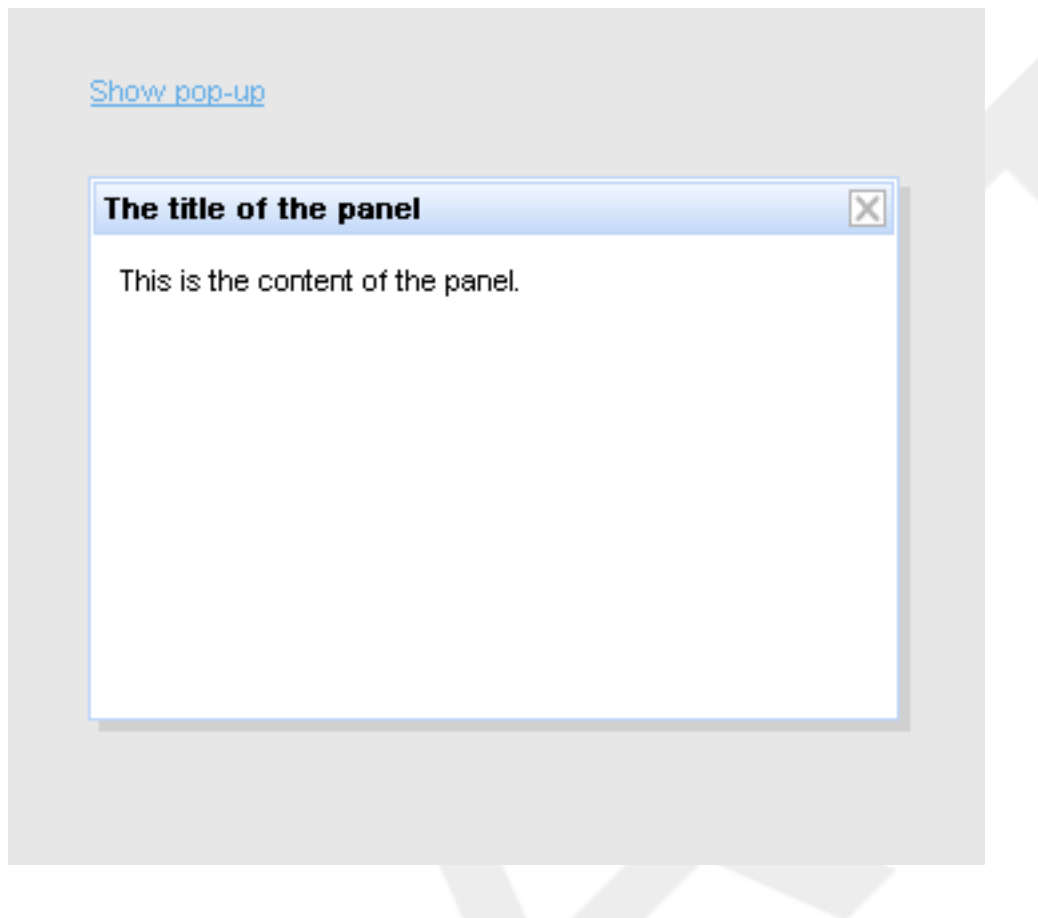


Figure 8.5. Header and controls

8.4.6. 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 `trimOverlaidElements` attribute is set to `false` then child components will not be trimmed if they extend beyond the borders of the pop-up panel. For example, if using a calendar, select, or other pop-up component, set `trimOverlaidElements="false"`.

8.4.7. JavaScript API

The `<rich:popupPanel>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`getTop()`

Return the top co-ordinate for the position of the pop-up panel.

`getLeft()`

Return the left co-ordinate for the position of the pop-up panel.

`moveTo(top, left)`

Move the pop-up panel to the co-ordinates specified with the *top* and *left* parameters.

`resize(width, height)`

Resize the pop-up panel to the size specified with the *width* and *height* parameters.

`show()`

Show the pop-up panel.

`hide()`

Hide the pop-up panel.

8.4.8. 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`

8.4.9. Style classes and skin parameters

Table 8.4. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters		Mapped CSS properties
<code>.rf-pp-btn</code> This class defines styles for the pop-up panel button.	No skin parameters.		
<code>.rf-pp-shade</code> This class defines styles for the shading that covers the page when presenting a modal pop-up panel.	No skin parameters.		
<code>.rf-pp-cntr</code> This class defines styles for the container for the pop-up panel.	<code>panelBorderColor</code>		border
	<code>generalBackgroundColor</code>		background

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-pp-hdr This class defines styles for the header of the pop-up panel.	<i>headerBackgroundColor</i>	background
.rf-pp-hdr-cnt This class defines styles for the content of the header.	<i>headerTextColor</i>	color
	<i>headerWeightFont</i>	font-weight
	<i>headerFamilyFont</i>	font-family
	<i>headerSizeFont</i>	font-size
.rf-pp-cnt This class defines styles for the content of the pop-up panel.	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
.rf-pp-cnt-sclrl This class defines styles for the scroll bars of the pop-up panel.	<i>generalBackgroundColor</i>	background
.rf-pp-hndlr This class defines styles for borders of the pop-up panel. The border handler is used to re-size the panel.	No skin parameters.	
.rf-pp-hndlr-t, .rf-pp-hndlr-b, .rf-pp-hndlr-l, .rf-pp-hndlr-r, .rf-pp-hndlr-tl, .rf-pp-hndlr-tr, .rf-pp-hndlr-bl, .rf-pp-hndlr-br These classes define styles for the top, bottom, left, right, top-left, top-right, bottom-left, and bottom-right edges of the border handler.	No skin parameters.	

8.5. <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 8.5.7, “<rich:tab>”](#) for further details on the <rich:tab> component.

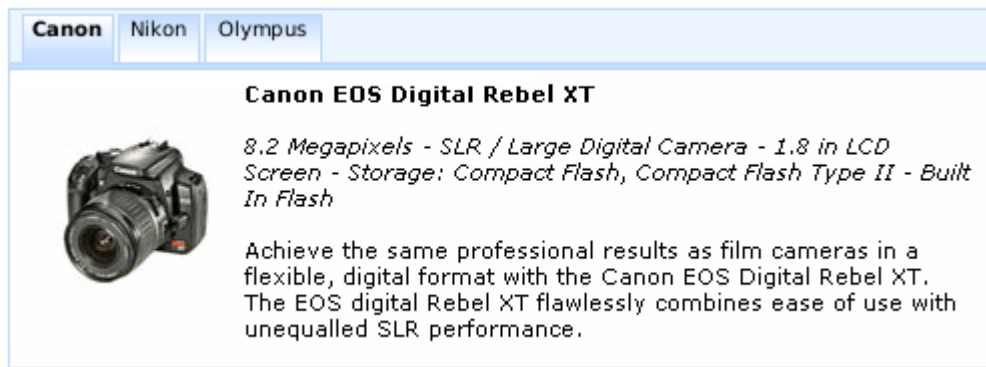


Figure 8.6. A `<rich:tabPanel>` component containing three `<rich:tab>` components



Form elements required

All `<rich:tabPanel>` components should be wrapped in a form element when using either `ajax` or `server` mode, as usual for submitting components.

8.5.1. 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 refreshing the page. Only one tab at a time is rendered 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 panel is refreshed. Only one tab at a time is rendered to the client side.

`client`

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

8.5.2. `<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 `itemchange` event points to the function to perform when the switchable item is changed.
- The `beforeitemchange` event points to the function to perform when before the switchable item is changed.

8.5.3. <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.

8.5.4. JavaScript API

The <rich:tabPanel> component can be controlled through the JavaScript API. The JavaScript API provides the following functions, which are common to all switchable panels:

`getItems()`

Return an array of the tabs contained in the tab panel.

`getItemsNames()`

Return an array of the names of the tabs contained in the tab panel.

`switchToItem(itemName)`

Switch to and display the item identified by the `itemName` string passed as a parameter.

`firstItem(), prevItem(), nextItem(), lastItem()`

Switch to and display the first item, the previous item, the next item, or the last item.

8.5.5. 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`

8.5.6. Style classes and skin parameters

Table 8.5. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-tab-hdr</code> This class defines styles for a tab header.	<code>panelBorderColor</code>	border
	<code>tabBackgroundColor</code>	background-color
	<code>generalTextColor</code>	color

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-tab-hdr-act</code> This class defines styles for a tab header when it is active.	<code>additionalBackgroundColor</code>	background-color
<code>.rf-tab-hdr-inact</code> This class defines styles for a tab header when it is inactive.	No skin parameters.	
<code>.rf-tab-hdr-dis</code> This class defines styles for a tab header when it is disabled.	<code>tabDisabledTextColor</code>	color
<code>.rf-tab-hdr-tabline-vis</code> This class defines styles for the header tab line when it is visible.	<code>additionalBackgroundColor</code>	background-color
	<code>panelBorderColor</code>	border-color
<code>.rf-tab-hdr-tabs</code> This class defines styles for the tabs in the header.	No skin parameters.	
<code>.rf-tab-hdr-spcr</code> This class defines styles for the tab header spacer.	<code>panelBorderColor</code>	border-bottom
<code>.rf-tab-lbl</code> This class defines styles for the tab label.	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
<code>.rf-tab-hdn</code> This class defines styles for the tab when it is hidden.	No skin parameters.	
<code>.rf-tab-hdr-scr1-lft,</code> <code>.rf-tab-hdr-scr1-rgh</code> These classes define styles for the left and right controls for the tab header scroller.	<code>additionalBackgroundColor</code>	background
	<code>panelBorderColor</code>	border
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
<code>.rf-tab-hdr-tablst</code> This class define styles for the tab header list.	<code>additionalBackgroundColor</code>	background
	<code>panelBorderColor</code>	border
	<code>generalFamilyFont</code>	font-family

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-tab-hdr-brd This class define styles for the tab header border.	<i>tabBackgroundColor</i>	background
	<i>panelBorderColor</i>	border
.rf-tab-cnt This class define styles for the content of the tab panel.	<i>generalBackgroundColor</i>	background
	<i>panelBorderColor</i>	border
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size

8.5.7. <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.

8.5.7.1. Basic usage

Basic usage of the <rich:tab> component requires only the tab header and tab content. No additional attributes are required.

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 custom components to be applied to the tab header. 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.

8.5.7.2. 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 8.5, “<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.

8.5.7.3. `<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 `enter` event points to the function to perform when the mouse enters the tab.
- The `leave` attribute points to the function to perform when the mouse leaves the tab.

8.5.7.4. 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`

8.5.7.5. Style classes and skin parameters

The `<rich:tab>` component uses the same styles as those applied to the parent `<rich:tabPanel>` component. Refer to [Section 8.5.6, “Style classes and skin parameters”](#) for details.

8.6. `<rich:togglePanel>`

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.

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

Refer to [Section 8.6.5, “`<rich:toggleControl>`”](#) and [Section 8.6, “`<rich:togglePanel>`”](#) for details on how to use the components together.

8.6.1. 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 panel component with a connected `<rich:toggleControl>` behavior.

The child components are shown in the order in which they are defined in the view, as shown in [Example 8.4, “Basic usage”](#).



Form elements required

All `<rich:tabPanel>` components should be wrapped in a form element when using either ajax or server mode, as usual for submitting components.

Example 8.4. Basic usage

```
<rich:togglePanel#id="layout"#activeItem="item1">
  <rich:togglePanelItem#id="item1">
    <!--content-->
  </rich:togglePanelItem>
  <rich:togglePanelItem#id="item2">
    <!--content-->
  <rich:togglePanelItem>
</rich:togglePanel>
<h:commandButton>
  <rich:toggleControl#activePanel="layout"/><!--cycles#through#the#states-->
</h:commandButton>
```

8.6.2. 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 refreshing the page. Only one child at a time is rendered 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 panel is refreshed. Only one child at a time is rendered to the client side.

client

Activation of a child component causes the parent `<rich:togglePanel>` component to update on the client side. All the items are rendered to the client side during the initial page render.

JavaScript changes the styles such that one child component becomes hidden while the other is shown.

8.6.3. JavaScript API

The `<rich:togglePanel>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions, which are common to all switchable panels:

`getItems()`

Return an array of the items contained in the toggle panel.

`getItemsNames()`

Return an array of the names of the items contained in the toggle panel.

`switchToItem(itemName)`

Switch to and display the item identified by the *itemName* string passed as a parameter.

`firstItem(), prevItem(), nextItem(), lastItem()`

Switch to and display the first item, the previous item, the next item, or the last item.

8.6.4. 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`

8.6.5. `<rich:toggleControl>`

The `<rich:toggleControl>` behavior can be attached to any interface component, whether inside or outside the controlled panel itself. It works with a `<rich:togglePanel>` component to switch between different `<rich:togglePanelItem>` components. Refer to [Section 8.6, “<rich:togglePanel>”](#) and [Section 8.6.6, “<rich:togglePanelItem>”](#) for details on how to use the components together.

The `<rich:toggleControl>` implements the JSF `BehaviorHolder` component, which provides events to attached components and behaviors.

8.6.5.1. Basic usage

If the `<rich:toggleControl>` component is positioned inside a `<rich:togglePanel>` component, no panel attachment attributes need to be defined, as the control is assumed to

switch through the <rich:togglePanelItem> components of its parent <rich:togglePanel> component.

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.

8.6.5.2. Specifying the next state

The <rich:toggleControl> component can switch the attached <rich:togglePanel> component in multiple ways:

- By default, the <rich:toggleControl> component will cycle through <rich:togglePanelItem> components in the order they are defined within the view.

Example 8.5. Default switching

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

- The next item to switch to can be explicitly defined by including a <rich:toggleControl> component within a <rich:togglePanelItem> component. Point the `targetItem` to the <rich:togglePanelItem> to switch to when the state is next changed.

Example 8.6. Explicit switching

```
<rich:togglePanel activeItem="item1">
  <rich:togglePanelItem id="item1">
    <!--content-->
    <h:commandButton>
      <rich:toggleControl targetItem="item2"> <!--switches to item2 -->
    </h:commandButton>
  </rich:togglePanelItem>
  <rich:togglePanelItem id="item2">
    <!--content-->
  </rich:togglePanelItem>
</rich:togglePanel>
```

```

        <rich:toggleControl targetItem="item1"> <!--switches to item1 -->
    </h:commandButton>
    <rich:togglePanelItem>
</rich:togglePanel>

```

- Alternatively, use the `targetItem` attribute with keywords to switch items. The `@first`, `@previous`, `@next`, and `@last` keywords switch to the first item, the previous item, the next item, and the last item respectively.

Example 8.7. Keyword-based switching

```

<rich:togglePanel activeItem="item1">
    <rich:togglePanelItem id="item1">
        <!--content-->
        <h:commandButton>
            <rich:toggleControl targetItem="@next"> <!--switches to next item
(item2)-->
        </h:commandButton>
    </rich:togglePanelItem>
    <rich:togglePanelItem id="item2">
        <!--content-->
        <h:commandButton>
            <rich:toggleControl targetItem="@previous"> <!--switches to previous
item (item1)-->
        </h:commandButton>
    </rich:togglePanelItem>
</rich:togglePanel>

```

8.6.5.3. 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`

8.6.6. <rich:togglePanelItem>

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

8.6.6.1. 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

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

9.1. <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 9, Tables and grids](#).

9.1.1. 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 9.1, “<a4j:repeat> example”](#) shows how to use <a4j:repeat> to maintain a simple table.

Example 9.1. <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.

9.1.2. 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 an 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.

9.1.3. 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`

9.2. <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 9.6, "<rich:extendedDataTable>"](#) for further details.

9.2.1. 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.

9.2.2. 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. `demoTable` demonstrates a simple table implementation.

Example 9.2. <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:dataTable>
```

```

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

```

United States Capitals

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

Figure 9.1. `<rich:dataTable>` example

For details on filtering and sorting data tables, refer to [Section 9.10, “Table filtering”](#) and [Section 9.11, “Table sorting”](#).

9.2.3. Partial updates

As `<rich:dataTable>` the component is based on the `<a4j:repeat>` component, it can be partially updated with Ajax. Refer to [Section 9.1.2, “Limited views and partial updates”](#) for details on partially updating the `<rich:dataTable>` component.

9.2.4. JavaScript API

The `<rich:dataTable>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`expandAllSubTables()`

Expand any sub-tables contained in the data table.

`collapseAllSubTables()`

Collapse any sub-tables contained in the data table.

`switchSubTables()`

Switch the expanded or collapsed state of any sub-tables contained in the data table.

`filter(columnId, newFilterValue, [isClearPreviousFilters])`

Filter the table based on the column specified with the `columnId` parameter. Use the `newFilterValue` parameter as the filter value. The optional `isClearPreviousFilters` parameter is a boolean value which, if set to `true`, will clear any previous filters applied to the table.

`sort(columnId, [direction], [isClearPreviousSorting])`

Sort the table based on the column specified with the `columnId` parameter. The option `direction` parameter specifies whether to sort in ascending or descending order. The optional `isClearPreviousSorting` parameter is a boolean value which, if set to `true`, will clear any previous sorting applied to the table.

9.2.5. 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`

9.2.6. Style classes and skin parameters

Table 9.1. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-dt</code> This class defines styles for the table.	<code>tableBackgroundColor</code>	background-color
	<code>tableBorderWidth</code>	border-left-width, border-top-width
	<code>tableBorderColor</code>	border-left-color, border-top-color
<code>.rf-dt-cap</code> This class defines styles for the table caption.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-dt-r</code> This class defines styles for a table row.	No skin parameters.	
<code>.rf-dt-fst-r</code> This class defines styles for the first row in a table.	No skin parameters.	
<code>.rf-dt-c</code> This class defines styles for a table cell.	<code>tableBackgroundColor</code>	background-color
	<code>tableBorderWidth</code>	border-bottom-width, border-right-width
	<code>tableBorderColor</code>	border-bottom-color, border-right-color
	<code>generalTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
<code>.rf-dt-nd</code> This class defines styles for a node.	<code>tableBorderWidth</code>	border-bottom-width, border-right-width
	<code>tableBorderColor</code>	border-bottom-color, border-right-color
	<code>generalTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
<code>.rf-dt-hdr</code> This class defines styles for a table header.	No skin parameters.	
<code>.rf-dt-hdr-fst</code> This class defines styles for the first header.	No skin parameters.	
<code>.rf-dt-hdr-c</code> This class defines styles for a header cell.	<code>tableHeaderBackgroundColor</code>	background-color
	<code>tableBorderWidth</code>	border-bottom-width, border-right-width
	<code>tableBorderColor</code>	border-bottom-color, border-right-color
	<code>tableHeaderTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-dt-shdr</code> This class defines styles for a table sub-header.	No skin parameters.	
<code>.rf-dt-shdr-fst</code> This class defines styles for the first sub-header.	No skin parameters.	
<code>.rf-dt-shdr-c</code> This class defines styles for a sub-header cell.	<code>tableHeaderBackgroundColor</code>	background-color
	<code>tableBorderWidth</code>	border-bottom-width, border-right-width
	<code>tableBorderColor</code>	border-bottom-color, border-right-color
	<code>tableHeaderTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
<code>.rf-dt-ftr</code> This class defines styles for a table footer.	No skin parameters.	
<code>.rf-dt-ftr-fst</code> This class defines styles for the first footer.	No skin parameters.	
<code>.rf-dt-ftr-c</code> This class defines styles for a footer cell.	<code>tableFooterBackgroundColor</code>	background-color
	<code>tableBorderWidth</code>	border-bottom-width, border-right-width
	<code>tableBorderColor</code>	border-bottom-color, border-right-color
	<code>generalTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
<code>.rf-dt-sftr</code> This class defines styles for a table sub-footer.	No skin parameters.	
<code>.rf-dt-sftr-fst</code> This class defines styles for the first sub-footer.	No skin parameters.	
<code>.rf-dt-sftr-c</code> This class defines styles for a sub-footer cell.	<code>tableFooterBackgroundColor</code>	background-color
	<code>tableBorderWidth</code>	border-bottom-width, border-right-width

Class (selector)	Skin Parameters	Mapped CSS properties
	<i>tableBorderColor</i>	border-bottom-color, border-right-color
	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size

9.3. <rich:column>

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

9.3.1. Basic usage

In general usage, the <rich:column> component is used in the same way as the JavaServer Faces (JSF) <h:column> component. It requires no extra attributes for basic usage, as shown in [Example 9.3, “Basic column example”](#).

Example 9.3. Basic column example

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


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

Figure 9.2. Basic column example

9.3.2. 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 9.4, “Column spanning example”](#).

Example 9.4. Column spanning example

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

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

Figure 9.3. Column spanning example

9.3.3. 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 9.5, “Row spanning example”](#) and the resulting [Figure 9.5, “Complex headers using column groups”](#) show the first column of the table spanning three rows.

Example 9.5. Row spanning example

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

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

Figure 9.4. Row spanning example

For details on filtering and sorting columns, refer to [Section 9.10, “Table filtering”](#) and [Section 9.11, “Table sorting”](#).

9.3.4. 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`

9.4. <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.

9.4.1. Complex headers

The `<rich:columnGroup>` can also be used to create complex headers in a table. [Example 9.6](#), “Complex headers using column groups” and the resulting [Figure 9.5](#), “Complex headers using column groups” demonstrate how complex headers can be achieved.

Example 9.6. Complex headers using column groups

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

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

Figure 9.5. Complex headers using column groups

9.4.2. 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`

9.5. <rich:collapsibleSubTable>

The `<rich:collapsibleSubTable>` component acts as a child element to a `<rich:dataTable>` component. It allows sections of a table to be grouped into collapsible sections. The `<rich:collapsibleSubTable>` component works with the `<rich:collapsibleSubTableToggler>` component, which allows the user to expand and collapse the sub tables.

9.5.1. Basic usage

The `<rich:collapsibleSubTable>` component requires the same basic attributes as the `<rich:dataTable>` component. The `value` attribute points to the collection, and the `var` attribute specifies a variable to use when iterating through the collection.

In addition, the `<rich:collapsibleSubTable>` component typically needs a corresponding `<rich:collapsibleSubTableToggler>` component to allow expanding and collapsing. Declare the `id` identifier on the `<rich:collapsibleSubTable>` element so that the `<rich:collapsibleSubTableToggler>` component can reference it. Refer to [Section 9.5.5](#),

“[<rich:collapsibleSubTableToggler>](#)” for details on the `<rich:collapsibleSubTableToggler>` component.

Example 9.7. Basic usage

```
<rich:dataTable value="#{carsBean.inventoryVendorLists}" var="list">
  <f:facet name="header">
    <rich:columnGroup>
      <rich:column colspan="6">
        <h:outputText value="Cars marketplace" />
      </rich:column>
      <rich:column breakRowBefore="true">
        <h:outputText value="Model" />
      </rich:column>
      <rich:column>
        <h:outputText value="Price" />
      </rich:column>
      <rich:column>
        <h:outputText value="Mileage" />
      </rich:column>
      <rich:column>
        <h:outputText value="VIN Code" />
      </rich:column>
      <rich:column>
        <h:outputText value="Items stock" />
      </rich:column>
      <rich:column>
        <h:outputText value="Days Live" />
      </rich:column>
    </rich:columnGroup>
  </f:facet>
  <rich:column colspan="6">
    <rich:collapsibleSubTableToggler for="sbtbl" />
    <h:outputText value="#{list.vendor}" />
  </rich:column>
  <rich:collapsibleSubTable value="#{list.vendorItems}" var="item" id="sbtbl"
    expandMode="client">
    <rich:column>
      <h:outputText value="#{item.model}" />
    </rich:column>
    <rich:column>
      <h:outputText value="#{item.price}" />
    </rich:column>
    <rich:column>
      <h:outputText value="#{item.mileage}" />
    </rich:column>
    <rich:column>
      <h:outputText value="#{item.vin}" />
    </rich:column>
  </rich:collapsibleSubTable>
</rich:dataTable>
```

```

</rich:column>
<rich:column>
    <h:outputText value="#{item.stock}" />
</rich:column>
<rich:column>
    <h:outputText value="#{item.daysLive}" />
</rich:column>
<f:facet name="footer">
    <h:outputText value="Total of #{list.vendor} Cars: #{list.count}" />
</f:facet>
</rich:collapsibleSubTable>
</rich:dataTable>

```

The resulting tables contains multiple sub-tables, grouping the list of cars by vendor. Each sub-table can be expanded or collapsed using the toggle with the vendor's name. The screenshot shows all sub-tables collapsed except for the sub-table for Ford cars.

Cars marketplace					
Model	Price	Mileage	VIN Code	Items stock	Days Live
⤴ Chevrolet					
⤵ Ford					
Taurus	23810	36710.0	CCEJYNFDFIZKQZWEJ	ORCOZLP	90
Taurus	26685	63661.0	KIPQZFJLGBOMITWZ	YNKCJZ	87
Taurus	47151	41638.0	MQJGCFPZTTMVMVORT	GNQWTW	82
Taurus	47017	25851.0	KCCJMHUVSDSOVIXOB	ZZEGUSN	66
Taurus	40805	47046.0	DZECUNNGCJYVHSVQZ	FNASXG	34
Explorer	24774	10659.0	REGUTBPGVCMTJWKNG	ZNJBQOJ	23
Explorer	34632	62375.0	ZYFFCGQHAOZXGQDEL	VHOEGYE	19
Explorer	15348	27136.0	ADAYRYSTSQJWJOKED	YZYJWF	65
Explorer	47482	47222.0	ZJORWUJUIBFYBWYPY	AOGBDT	63
Explorer	30610	14070.0	BSZWVUQFJWVCARGIW	QRXBYAI	26
Total of Ford Cars: 23					
⤴ GMC					
⤴ Infiniti					
⤴ Nissan					
⤴ Toyota					

9.5.2. Expanding and collapsing the sub-table

Use the boolean `expanded` attribute to control the current state of the sub-table.

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

server

The default setting. Expansion of the `<rich:collapsibleSubTable>` component performs a common submission, completely re-rendering the page.

ajax

Expansion of the `<rich:collapsibleSubTable>` component performs an Ajax form submission, and the content of the data table is rendered.

client

Expansion of the `<rich:collapsibleSubTable>` component updates the data table on the client side.

9.5.3. Reference data

- *component-type*: `org.richfaces.CollapsibleSubTable`
- *component-class*: `org.richfaces.component.html.HtmlCollapsibleSubTable`
- *component-family*: `org.richfaces.CollapsibleSubTable`
- *renderer-type*: `org.richfaces.CollapsibleSubTable`
- *tag-class*: `org.richfaces.taglib.CollapsibleSubTableTag`

9.5.4. Style classes

Table 9.2. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-cst</code> This class defines styles for the table.	No skin parameters.	
<code>.rf-cst-r</code> This class defines styles for a table row.	No skin parameters.	
<code>.rf-cst-fst-r</code> This class defines styles for the first row in a table.	No skin parameters.	
<code>.rf-cst-c</code> This class defines styles for a table cell.	<code>tableBackgroundColor</code>	background-color
	<code>tableBorderWidth</code>	border-bottom-width, border-right-width
	<code>tableBorderColor</code>	border-bottom-color, border-right-color
	<code>generalTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-cst-hdr</code> This class defines styles for a table header.	No skin parameters.	
<code>.rf-cst-hdr-fst</code> This class defines styles for the first header.	No skin parameters.	
<code>.rf-cst-hdr-fst-r</code> This class defines styles for the first row in the header.	No skin parameters.	
<code>.rf-cst-hdr-c</code> This class defines styles for a header cell.	<code>tableSubHeaderBackgroundColor</code>	background-color
	<code>tableBorderWidth</code>	border-bottom-width, border-right-width
	<code>tableBorderColor</code>	border-bottom-color, border-right-color
	<code>generalTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
<code>.rf-cst-shdr</code> This class defines styles for a table sub-header.	No skin parameters.	
<code>.rf-cst-shdr-fst</code> This class defines styles for the first sub-header.	No skin parameters.	
<code>.rf-cst-shdr-c</code> This class defines styles for a sub-header cell.	<code>tableSubHeaderBackgroundColor</code>	background-color
	<code>tableBorderWidth</code>	border-bottom-width, border-right-width
	<code>tableBorderColor</code>	border-bottom-color, border-right-color
	<code>generalTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
<code>.rf-cst-ftr</code> This class defines styles for a table footer.	No skin parameters.	
<code>.rf-cst-ftr-fst</code> This class defines styles for the first footer.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-cst-ftr-c This class defines styles for a footer cell.	<i>tableSubFooterBackgroundColor</i>	background-color
	<i>tableBorderWidth</i>	border-bottom-width, border-right-width
	<i>tableBorderColor</i>	border-bottom-color, border-right-color
	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
.rf-cst-sftr This class defines styles for a table sub-footer.	No skin parameters.	
.rf-cst-sftr-fst This class defines styles for the first sub-footer.	No skin parameters.	
.rf-cst-sftr-c This class defines styles for a sub-footer cell.	<i>tableSubFooterBackgroundColor</i>	background-color
	<i>tableBorderWidth</i>	border-bottom-width, border-right-width
	<i>tableBorderColor</i>	border-bottom-color, border-right-color
	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size

9.5.5. <rich:collapsibleSubTableToggler>

The <rich:collapsibleSubTableToggler> component provides a toggle control for the user to expand and collapse sub-tables.

9.5.5.1. Basic usage

The <rich:collapsibleSubTableToggler> component requires the `for` attribute. The `for` attribute references the `id` identifier of the <rich:collapsibleSubTable> component to control.

Refer to [Example 9.7, “Basic usage”](#) for an example using the <rich:collapsibleSubTable> component. In the example, the toggle control is placed in a column that spans the width of the table. Output text next to the toggle control displays the car vendor's name for that sub-table.

9.5.5.2. Reference data

- `component-type`: `org.richfaces.CollapsibleSubTableToggler`
- `component-class`: `org.richfaces.component.html.HtmlCollapsibleSubTableToggler`

- *component-family*: org.richfaces.CollapsibleSubTableToggler
- *renderer-type*: org.richfaces.CollapsibleSubTableToggler
- *tag-class*: org.richfaces.taglib.CollapsibleSubTableTogglerTag

9.5.5.3. Style classes and skin parameters

Style classes (selectors)

.rf-csttg

This class defines styles for a toggle control.

.rf-csttg-exp

This class defines styles for a toggle control which expands the sub-table.

.rf-csttg-colps

This class defines styles for a toggle control which collapses the sub-table.

9.6. <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 main attributes not included in the <rich:dataTable> component:

- clientRows
- frozenColumns
- height
- onselectionchange
- selectedClass
- selection
- selectionMode

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

- breakBefore
- columnGroup
- colSpan
- rowSpan



Complex sub-tables

Due to the complex mark-up involved in the `<rich:extendedDataTable>` component, it does not support the use of the `<rich:collapsibleSubTable>` component. The `<rich:collapsibleSubTable>` component is only available with the `<rich:dataTable>` component.

Similarly, complex row and column spanning using the `breakBefore`, `columnGroup`, `colSpan`, and `rowSpan` attributes is also not available with the `<rich:extendedDataTable>` component.

9.6.1. 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 9.2, “<rich:dataTable>”](#) for details.

9.6.2. 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 and footer facets.

9.6.3. Extended features

Example 9.8. `<rich:extendedDataTable>` example

This example `<rich:extendedDataTable>` component demonstrates horizontal and vertical scrolling and frozen columns. Each feature is detailed in this section.

```
<rich:extendedDataTable value="#{carsBean.allInventoryItems}"
    var="car" id="table" frozenColumns="2"
    style="height:300px; width:500px;" selectionMode="none">
  <f:facet name="header">
    <h:outputText value="Cars marketplace" />
  </f:facet>
  <rich:column>
    <f:facet name="header">
      <h:outputText value="vendor" />
    </f:facet>
    <h:outputText value="#{car.vendor}" />
  </rich:column>
  <rich:column>
    <f:facet name="header">
      <h:outputText value="Model" />
    </f:facet>
    <h:outputText value="#{car.model}" />
  </rich:column>
</rich:extendedDataTable>
```

```
</f:facet>
  <h:outputText value="#{car.model}" />
</rich:column>
<rich:column>
  <f:facet name="header">
    <h:outputText value="Price" />
  </f:facet>
  <h:outputText value="#{car.price}" />
</rich:column>
<rich:column>
  <f:facet name="header">
    <h:outputText value="Mileage" />
  </f:facet>
  <h:outputText value="#{car.mileage}" />
</rich:column>
<rich:column>
  <f:facet name="header">
    <h:outputText value="VIN Code" />
  </f:facet>
  <h:outputText value="#{car.vin}" />
</rich:column>
<rich:column>
  <f:facet name="header">
    <h:outputText value="Items stock" />
  </f:facet>
  <h:outputText value="#{car.stock}" />
</rich:column>
<rich:column>
  <f:facet name="header">
    <h:outputText value="Days Live" />
  </f:facet>
  <h:outputText value="#{car.daysLive}" />
</rich:column>
</rich:extendedDataTable>
```

Cars marketplace				
vendor	Model	Price	Mileage	VIN Code
Chevrolet	Corvette	17226	25965.0	ILLAKAWAZDZ
Chevrolet	Corvette	34229	46429.0	RCPNSRYGXOM
Chevrolet	Corvette	27982	50209.0	NWLGCEVEHGI
Chevrolet	Corvette	51825	72998.0	NGVZSCIZGSM
Chevrolet	Corvette	52845	34364.0	PSDRUYYOIJG
Chevrolet	Malibu	37874	37273.0	VLFPQPWNEFD
Chevrolet	Malibu	15600	71441.0	EXLJGDWOZSA
Chevrolet	Malibu	52447	46700.0	NLMGJZAKBRD
Chevrolet	Malibu	27129	36254.0	OIPFUIENLEHS
Chevrolet	Malibu	28846	77162.0	WRCOOFREZLL
Chevrolet	Malibu	46165	60590.0	HUFTTHQHSFJF
Chevrolet	Malibu	18263	37790.0	JLMHNAFESHVD

9.6.3.1. Scrolling

The example table shown in [Example 9.8, “<rich:extendedDataTable> example”](#) features both horizontal and vertical scrolling. Scrolling occurs automatically when the contents of the table exceed the dimensions specified with the `height` and `width` attributes. Headers and footers remain in place and visible when the table is scrolled.

Large tables can use Ajax "lazy" loading to cache data on the client during scrolling. Use the `clientRows` attribute to specify the number of rows to load. The specified number of rows are loaded on the initial rendering and with every vertical scroll. If the `clientRows` attribute is not specified, all the rows are loaded on the client without the use of Ajax.

In addition to Ajax scrolling, the `<rich:extendedDataTable>` component can also be used with the `<rich:dataScroller>` component in the same way as a regular `<rich:dataTable>` component. Refer to [Section 9.9, “<rich:dataScroller>”](#) for full details on using the `<rich:dataScroller>` component.

9.6.3.2. Frozen columns

The example table shown in [Example 9.8, “<rich:extendedDataTable> example”](#) has the first two columns frozen so that they remain visible if the user scrolls horizontally through the table. Note that the horizontal scrollbar does not encompass these frozen columns. To freeze columns, use the `frozenColumns` attribute to specify the number of columns on the left-hand side of the table to freeze.

9.6.3.3. Row selection

Row selection is determined by the `selectionMode` attribute. Setting the attribute to `none` allows for no row selection capability. The example table shown in [Example 9.8, “<rich:extendedDataTable> example”](#) does not allow row selection.

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 `multiple`, the user can select multiple rows. Holding down the **Ctrl** key while clicking selects additional rows with each click. Holding down the **Shift** key while clicking selects all the rows in a range.

The `selection` attribute points to a collection of objects. It holds the `rowKey` identifiers to track which rows are selected. [Example 9.9, “Selecting multiple rows”](#) shows how to implement multiple row selection in the same table from [Example 9.8, “<rich:extendedDataTable> example”](#).

Example 9.9. Selecting multiple rows

```
<rich:extendedDataTable value="#{extTableSelectionBean.inventoryItems}"
    var="car" selection="#{extTableSelectionBean.selection}"
    id="table" frozenColumns="2"
    style="height:300px; width:500px;" selectionMode="multiple">
    ...
```

The accompanying `ExtSelectionBean` bean handles which rows are selected. The rows are identified by their `rowKey` identifiers.

```
package org.richfaces.demo.tables;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import javax.faces.bean.ManagedBean;
import javax.faces.bean.ManagedProperty;
import javax.faces.bean.SessionScoped;
import javax.faces.event.AjaxBehaviorEvent;

import org.richfaces.component.UIExtendedDataTable;
import org.richfaces.demo.tables.model.cars.InventoryItem;

@ManagedBean
@SessionScoped

public class ExtTableSelectionBean implements Serializable{

    private Collection<Object> selection;
    @ManagedProperty(value = "#{carsBean.allInventoryItems}")
    private List<InventoryItem> inventoryItems;
    private List<InventoryItem> selectionItems = new ArrayList<InventoryItem>();

    public void selectionListener(AjaxBehaviorEvent event){
```

```
UIExtendedDataTable dataTable = (UIExtendedDataTable)event.getComponent();
Object originalKey = dataTable.getRowKey();
selectionItems.clear();
for (Object selectionKey: selection) {
    dataTable.setRowKey(selectionKey);
    if (dataTable.isRowAvailable()){
        selectionItems.add((InventoryItem)dataTable.getRowData());
    }
}
dataTable.setRowKey(originalKey);
}

public Collection<Object> getSelection() {
    return selection;
}

public void setSelection(Collection<Object> selection) {
    this.selection = selection;
}

public List<InventoryItem> getInventoryItems() {
    return inventoryItems;
}

public void setInventoryItems(List<InventoryItem> inventoryItems) {
    this.inventoryItems = inventoryItems;
}

public List<InventoryItem> getSelectionItems() {
    return selectionItems;
}

public void setSelectionItems(List<InventoryItem> selectionItems) {
    this.selectionItems = selectionItems;
}
}
```


Cars marketplace				
vendor	Model	Price	Mileage	VIN Code
Chevrolet	Corvette	17226	25965.0	ILLAKAWAZDZ
Chevrolet	Corvette	34229	46429.0	RCPNSRYGXOM
Chevrolet	Corvette	27982	50209.0	NWLGCEVEHGI
Chevrolet	Corvette	51825	72998.0	NGVZSCIZGSM
Chevrolet	Corvette	52845	34364.0	PSDRUYYOIJG
Chevrolet	Malibu	37874	37273.0	VLFPQPWNEFD
Chevrolet	Malibu	15600	71441.0	EXLJGDWOZSA
Chevrolet	Malibu	52447	46700.0	NLMGJZAKBRD
Chevrolet	Malibu	27129	36254.0	OIPFUIENLEHSX
Chevrolet	Malibu	28846	77162.0	WRCOOFREZLL
Chevrolet	Malibu	46165	60590.0	HUFTTHQHSFJF
Chevrolet	Malibu	18263	37790.0	JL MHNAFESHVD

9.6.3.4. Rearranging columns

Columns in a `<rich:extendedDataTable>` component can be rearranged by the user by dragging each column to a different position. A graphical representation of the column is displayed during dragging. [Figure 9.6, “Dragging columns”](#) illustrates the **Price** column being dragged to a new location. The small blue arrow indicates where the column will be moved to if it is dropped in the current position. [Figure 9.7, “Rearranged columns”](#) shows the result of dragging the **Price** column.

Cars marketplace				
vendor	Model	Price	Mileage	VIN Code
Chevrolet	Corvette	17226	25965.0	ILLAKAWAZDZ
Chevrolet	Corvette	34229	46429.0	RCPNSRYGXOM
Chevrolet	Corvette	27982	50209.0	NWLGCEVEHGI
Chevrolet	Corvette	51825	72998.0	NGVZSCIZGSM
Chevrolet	Corvette	52845	34364.0	PSDRUYYOIJG
Chevrolet	Malibu	37874	37273.0	VLFPQPWNEFD
Chevrolet	Malibu	15600	71441.0	EXLJGDWOZSA
Chevrolet	Malibu	52447	46700.0	NLMGJZAKBRD
Chevrolet	Malibu	27129	36254.0	OIPFUIENLEHSX
Chevrolet	Malibu	28846	77162.0	WRCOOFREZLL
Chevrolet	Malibu	46165	60590.0	HUFTTHQHSFJF
Chevrolet	Malibu	18263	37790.0	JL MHNAFESHVD

Figure 9.6. Dragging columns

Cars marketplace				
vendor	Model	Mileage	Price	VIN Code
Chevrolet	Corvette	25965.0	17226	ILLAKAWAZDZ
Chevrolet	Corvette	46429.0	34229	RCPNSTRYGXOM
Chevrolet	Corvette	50209.0	27982	NWLGCEVEHGM
Chevrolet	Corvette	72998.0	51825	NGVZSCIZGSM
Chevrolet	Corvette	34364.0	52845	PSDRUYYOIJG
Chevrolet	Malibu	37273.0	37874	VLFPQPWNEFD
Chevrolet	Malibu	71441.0	15600	EXLJGDWOZSA
Chevrolet	Malibu	46700.0	52447	NLMGJZAKBRD
Chevrolet	Malibu	36254.0	27129	OIPFUIENLEHSX
Chevrolet	Malibu	77162.0	28846	WRCOOFREZLL
Chevrolet	Malibu	60590.0	46165	HUFTTHQHSFJF
Chevrolet	Malibu	37790.0	18263	JL MHNAFESHVD

Figure 9.7. Rearranged columns

9.6.3.5. Filtering and sorting

The `<rich:extendedDataTable>` component can include filtering and sorting in the same way as a regular `<rich:dataTable>` component. For full details on filtering tables, refer to [Section 9.10, “Table filtering”](#). For full details on sorting tables, refer to [Section 9.11, “Table sorting”](#).

9.6.4. JavaScript API

The `<rich:extendedDataTable>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

```
sort()
    Sort the data table.

filter()
    Filter the data table.

clearSorting()
    Clear any sorting that is currently applied to the table.

clearFiltering()
    Clear any filtering that is currently applied to the table.

selectRow(index)
    Select the row specified by the index parameter.
```

`selectRows([startIndex, stopIndex])`

Select all the rows in the table. Optionally, select only those rows between the indexes specified with the *startIndex* and *stopIndex* parameters.

`deselectRow`

Deselect the row that is currently selected.

`setActiveRow(index)`

Set the active row to that specified by the *index* parameter.

9.6.5. 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`

9.6.6. Style classes and skin parameters

Table 9.3. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-edt</code> This class defines styles for the table.	<code>tableBorderWidth</code> , <code>tableBorderColor</code>	border
	<code>tableBackgroundColor</code>	background-color
<code>.rich-edt-cnt</code> This class defines styles for the table content.	No skin parameters.	
<code>.rf-edt-c</code> This class defines styles for a table cell.	<code>tableBorderWidth</code> , <code>tableBorderColor</code>	border-bottom
	<code>tableBorderWidth</code> , <code>tableBorderColor</code>	border-right
<code>.rf-edt-c-cnt</code> This class defines styles for the contents of a cell.	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
<code>.rf-edt-tbl-hdr</code> This class defines styles for the table header.	<code>tableBorderWidth</code> , <code>tableBorderColor</code>	border-bottom
	<code>tableHeaderTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size

Class (selector)	Skin Parameters	Mapped CSS properties
	<i>tableHeaderTextColor</i>	color
.rich-edt-hdr This class defines styles for a header.	No skin parameters.	
.rf-edt-hdr-c This class defines styles for a table header cell.	<i>tableBorderWidth</i> , <i>tableBorderColor</i>	border-bottom
	<i>tableBorderWidth</i> , <i>tableBorderColor</i>	border-right
.rf-edt-hdr-c-cnt This class defines styles for the contents of a header cell.	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
	<i>tableHeaderTextColor</i>	color
.rf-edt-tbl-ftr This class defines styles for the table footer.	<i>tableBorderWidth</i> , <i>tableBorderColor</i>	border-top
	<i>tableFooterBackgroundColor</i>	background-color
.rich-edt-ftr This class defines styles for a footer.	<i>tableBorderWidth</i> , <i>tableBorderColor</i>	border-top
	<i>tableFooterBackgroundColor</i>	background-color
.rich-edt-ftr-cnt This class defines styles for the content of a footer.	No skin parameters.	
.rf-edt-ftr-c This class defines styles for a table footer cell.	<i>tableBorderWidth</i> , <i>tableBorderColor</i>	border-bottom
	<i>tableBorderWidth</i> , <i>tableBorderColor</i>	border-right
.rf-edt-ftr-c-cnt This class defines styles for the contents of a footer cell.	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
	<i>generalTextColor</i>	color
.rf-edt-ftr-emp This class defines styles for an empty footer cell.	<i>tableBorderWidth</i> , <i>tableBorderColor</i>	border-right
.rich-edt-ftr-fzn This class defines styles for a frozen footer.	No skin parameters.	
.rich-edt-b This class defines styles for the body of the table.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-edt-r-sel</code> This class defines styles for the selected row.	<code>tableBorderWidth</code> , <code>tableBorderColor</code>	<code>border-right</code>
<code>.rich-edt-r-act</code> This class defines styles for the active row.	No skin parameters.	
<code>.rich-edt-rsz</code> This class defines styles for the table resizer.	No skin parameters.	
<code>.rich-edt-rsz-cntr</code> This class defines styles for the resize container.	No skin parameters.	
<code>.rich-edt-rsz-mkr</code> This class defines styles for the resize marker.	<code>generalTextColor</code>	<code>border-left</code>
<code>.rf-edt-rord</code> This class defines styles for the re-order functionality.	<code>tableBorderWidth</code> , <code>tableBorderColor</code>	<code>border</code>
	<code>tableHeaderBackgroundColor</code> / <code>tableBackgroundColor</code>	<code>background-color</code>
<code>.rich-edt-rord-mkr</code> This class defines styles for the re-order marker.	No skin parameters.	
<code>.rich-edt-spcr</code> This class defines a spacer for Internet Explorer 7 compatibility.	No skin parameters.	

9.7. <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.

Car Store	
Chevrolet Corvette Price: 46071 Mileage: 40446.0	Chevrolet Corvette Price: 46416 Mileage: 48531.0
Chevrolet Corvette Price: 47822 Mileage: 15438.0	Chevrolet Corvette Price: 16629 Mileage: 69237.0

Navigation: <<< << 1 2 3 4 5 6 7 8 9 10 >> >>>

Figure 9.8. The `<rich:dataGrid>` component

9.7.1. 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.

9.7.2. Customizing the grid

The number of columns for the grid is specified with the `columns` attribute, and the number of elements to layout among the columns is determined with the `elements` attribute. The `first` attribute references the zero-based element in the data model from which the grid starts.

Example 9.10. `<rich:dataGrid>` example

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

  <rich:dataGrid value="#{dataTableScrollerBean.allCars}" var="car" columns="2" elements="4" first="0">
    <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>
  </rich:dataGrid>
</h:form>
</rich:panel>
```

```

</rich:panel>
<f:facet name="footer">
    <rich:datascroller></rich:datascroller>
</f:facet>
</rich:dataGrid>
</h:form>
</rich:panel>

```

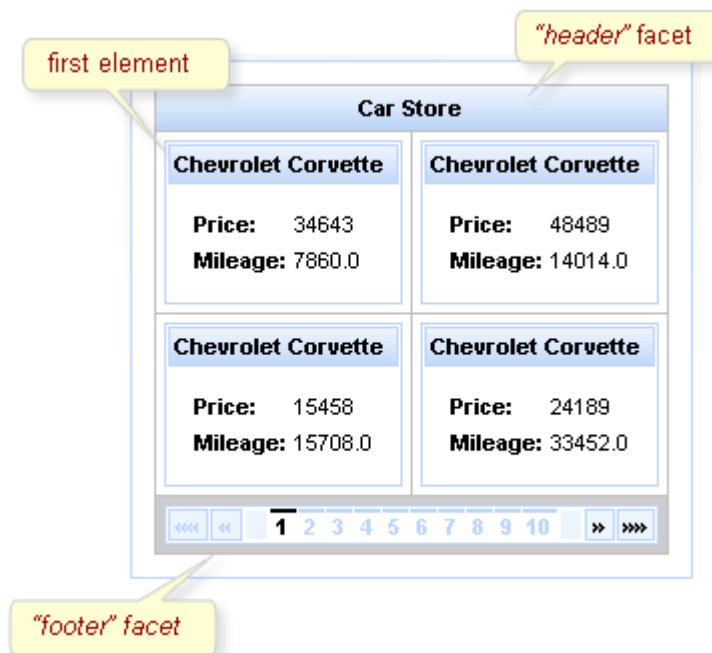


Figure 9.9. `<rich:dataGrid>` example

9.7.3. Partial updates

As `<rich:dataGrid>` the component is based on the `<a4j:repeat>` component, it can be partially updated with Ajax. Refer to [Section 9.1.2, “Limited views and partial updates”](#) for details on partially updating the `<rich:dataGrid>` component.

9.7.4. 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`

9.7.5. Style classes and skin parameters

Table 9.4. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-dg This class defines styles for the grid.	<i>tableBackgroundColor</i>	background-color
	<i>tableBorderWidth</i>	border-left-width, border-top-width
	<i>tableBorderColor</i>	border-left-color, border-top-color
.rf-dg-cap This class defines styles for the grid caption.	No skin parameters.	
.rf-dg-r This class defines styles for a grid row.	No skin parameters.	
.rf-dg-c This class defines styles for a grid cell.	<i>tableBorderWidth</i>	border-bottom-width, border-right-width
	<i>tableBorderColor</i>	border-bottom-color, border-right-color
	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
.rf-dg-nd-c This class defines styles for a node cell.	<i>tableBorderWidth</i>	border-bottom-width, border-right-width
	<i>tableBorderColor</i>	border-bottom-color, border-right-color
	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
.rf-dg-th This class defines styles for the grid header section.	<i>tableBorderWidth</i>	border-bottom-width
	<i>tableBorderColor</i>	border-bottom-color
.rf-dg-h This class defines styles for a grid header.	No skin parameters.	
.rf-dg-h-f This class defines styles for the first header.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-dg-h-r</code> This class defines styles for a header row.	No skin parameters.	
<code>.rf-dg-h-c</code> This class defines styles for a header cell.	<code>tableHeaderBackgroundColor</code>	background-color
	<code>tableBorderWidth</code>	border-bottom-width, border-right-width
	<code>tableBorderColor</code>	border-bottom-color, border-right-color
	<code>tableHeaderTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
<code>.rf-dg-f</code> This class defines styles for a grid footer.	No skin parameters.	
<code>.rf-dg-f-f</code> This class defines styles for the first footer.	No skin parameters.	
<code>.rf-dg-f-c</code> This class defines styles for a footer cell.	<code>tableFooterBackgroundColor</code>	background-color
	<code>tableBorderWidth</code>	border-bottom-width, border-right-width
	<code>tableBorderColor</code>	border-bottom-color, border-right-color
	<code>generalTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size

9.8. <rich:list>

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

9.8.1. 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 Language).

9.8.2. Type of list

By default, the list is displayed as an un-ordered 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 `` HTML element.

- 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 9.10. Unordered list

`ordered`

The list is presented as a numbered series of items, similar to the `` HTML element.

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

Figure 9.11. 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.

```

Chevrolet Corvette
  Price:18098
  Mileage:16296.0
Chevrolet Malibu
  Price:36523
  Mileage:46112.0
Chevrolet Malibu
  Price:33307
  Mileage:57709.0
Chevrolet Malibu
  Price:34248
  Mileage:62821.0
Chevrolet Malibu
  Price:51555
  Mileage:51549.0

```

Figure 9.12. 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 9.11, “Data definition list”](#).

Example 9.11. 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>

```

9.8.3. 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.

9.8.4. 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 9.12, “<rich:list> example”](#) shows a simple example using the `<rich:list>` component.

Example 9.12. <rich:list> example

```

<h:form>
<rich:list var="cars" 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 9.13. <rich:list> example**9.8.5. 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

9.8.6. Style classes and skin parameters

Table 9.5. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-ulst-itm This class defines styles for an item in an unordered list.	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
.rf-olst-itm This class defines styles for an item in an unordered list.	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
.rf-dlst-trm This class defines styles for the term of an item in a definition list.	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
.rf-dlst-dfn This class defines styles for the definition of an item in a definition list.	<i>generalTextColor</i>	color
	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size

9.9. <rich:dataScroller>

The <rich:dataScroller> component is used for navigating through multiple pages of tables or grids.



Figure 9.14. The <rich:dataScroller> component

9.9.1. Basic usage

The `<rich:dataScroller>` must be placed in the `footer` facet of the table or grid it needs to control. Alternatively, use the `for` attribute to bind the parent table or grid to the scroller.

The bound table or grid should also have the `rows` attribute defined to limit the number of rows per page.

The `<rich:dataScroller>` component must be re-rendered whenever a filter changes on the bound table, so that the scroller matches the current model for the table.

Example 9.13. Basic usage

```
<rich:dataTable id="table" value="#{capitalsBean.capitals}" var="cap" rows="5">
  <!-- table content -->
  ...
</rich:dataTable>
<rich:datascroller for="table" maxPages="5">
  <f:facet name="first">
    <h:outputText value="First" />
  </f:facet>
  <f:facet name="last">
    <h:outputText value="Last" />
  </f:facet>
</rich:datascroller>
```

9.9.2. Appearance and interactivity

The `page` attribute is a value-binding attribute used to define and save the current page number.

The `<rich:dataScroller>` component provides a range of controllers for scrolling through tables and grids:

Controls for scrolling by a specific amount

The component includes controls for switching to the first page, the last page, the next page, and the previous page, as well as controls for fast-forwarding or rewinding by a set amount. Use the `fastStep` attribute to set the number of pages to skip when fast-forwarding or rewinding.

The appearance of these controls can be customized using the following facets: `first`, `last`, `next`, `previous`, `fastForward`, and `fastRewind`. Additionally, there are facets for the controls' disabled states: `first_disabled`, `last_disabled`, `next_disabled`, `previous_disabled`, `fastforward_disabled`, and `rewind_disabled`.

Page controls

The component also features a series of numbered controls to jump to a specific page. Use the `maxPages` attribute to limit the number of page controls that appear. The current page control is highlighted.

The `pageIndexVar` and `pagesVar` attributes are request-scope variables for the current page and the total number of pages. Use these attributes with the `pages` facet to provide information about the pages of the table, as shown in [Example 9.14](#), “*pages facet*”.

Example 9.14. `pages` facet

```
<rich:dataTable value="#{capitalsBean.capitals}" var="cap" rows="5">
  <!-- table content -->
  ...
  <f:facet name="footer">
    <rich:datascroller pageIndexVar="pageIndex" pagesVar="pages">
      <f:facet name="pages">
        <h:outputText value="#{pageIndex} / #{pages}" />
      </f:facet>
    </rich:datascroller>
  </f:facet>
</rich:dataTable>
```

To add optional separators between controls, define the separators with the `controlsSeparator` facet.

9.9.3. JavaScript API

The `<rich:dataScroller>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`switchToPage(pageIndex)`

Switch to the page specified with the `pageIndex` parameter.

`next()`

Switch to the next page.

`previous()`

Switch to the previous page.

`first()`

Switch to the first page.

`last()`

Switch to the last page.

`fastForward()`

Step forward through the pages by the `fastStep` amount.

`fastRewind()`

Step backward through the pages by the `fastStep` amount.

9.9.4. Reference data

- `component-type`: `org.richfaces.DataScroller`
- `component-class`: `org.richfaces.component.html.HtmlDataScroller`
- `component-family`: `org.richfaces.DataScroller`
- `renderer-type`: `org.richfaces.DataScrollerRenderer`
- `tag-class`: `org.richfaces.taglib.DataScrollerTag`

9.9.5. Style classes and skin parameters

Table 9.6. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-ds</code> This class defines styles for the data scroller.	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
	<code>tableBackgroundColor</code>	background
<code>.rf-ds-btn</code> This class defines styles for buttons in the data scroller.	<code>generalTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
	<code>tableBorderColor</code>	border-color
	<code>headerBackgroundColor</code>	background-color
<code>.rf-ds-btn-first</code> This class defines styles for the first button.	No skin parameters.	
<code>.rf-ds-btn-fastrwd</code> This class defines styles for the fast rewind button.	No skin parameters.	
<code>.rf-ds-btn-prev</code> This class defines styles for the previous button.	No skin parameters.	
<code>.rf-ds-btn-next</code> This class defines styles for the next button.	No skin parameters.	
<code>.rf-ds-btn-fastfwd</code> This class defines styles for the fast forward button.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-ds-btn-last</code> This class defines styles for the last button.	No skin parameters.	
<code>.rf-ds-nmb-btn</code> This class defines styles for page number buttons in the data scroller.	<code>generalTextColor</code>	color
	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
	<code>tableBorderColor</code>	border-color
	<code>tableBackgroundColor</code>	background-color
<code>.rf-ds-press</code> This class defines styles for a data scroller when a control is pressed.	<code>tableBorderColor</code>	border-color
<code>tableBackgroundColor</code>	background	
<code>.rf-ds-act</code> This class defines styles for an active data scroller.	<code>tableBorderColor</code>	color
<code>.rf-ds-dis</code> This class defines styles for a disabled data scroller.	<code>tableBorderColor</code>	color

9.10. Table filtering



Documentation in development

This section is currently under development. Any features it describes may not be available in the current release of RichFaces.

Tables entries can be filtered by the user through either the basic method built in to the `<rich:column>` component, or by defining external filters. Refer to [Section 9.3, “<rich:column>”](#) for details on using the `<rich:column>` component in tables.

9.10.1. Basic filtering

The built-in filtering abilities of the `<rich:column>` component allow the user to enter text as a filtering value. The table displays only those entries that begin with the filter value.

Set the `filterValue` attribute to point to the value used to filter the column. This can be either an initial filtering value on the page, or a value binding on the server. The `filterValue` value is then used with the JavaScript `startsWith()` method to filter the column entries based on the

data specified with the `filterBy` attribute. Expressions in the `filterBy` attribute must refer to the variable declared in the table's `var` attribute, which is used to fill the contents of the table.

The filter is processed and the table is rendered when the `onblur` event occurs for the column. This can be changed by defining a different event with the `filterEvent` attribute. For example, to implement live updating such that the filter refreshes after every keystroke, set `filterEvent="keyup"`.

Example 9.15. Basic filtering

```
<rich:dataTable value="#{capitalsBean.capitals}" var="cap">
  <f:facet name="header">
    <rich:column>
      <h:outputText value="State Name">
    </rich:column>
    <rich:column>
      <h:outputText value="State Capital">
    </rich:column>
  </f:facet>
  <rich:column filterValue="#{capitalsBean.currentStateFilterValue}"
    filterBy="#{cap.state}" filterEvent="keyup">
    <h:outputText value="#{cap.state}" />
  </rich:column>
  <rich:column filterValue="#{capitalsBean.currentNameFilterValue}"
    filterBy="#{cap.name}" filterEvent="keyup">
    <h:outputText value="#{cap.name}" />
  </rich:column>
</rich:dataTable>
```

The example uses the basic filtering method on both columns in the table.

State Name	State Capital
<input type="text" value="n"/>	<input type="text"/>
Nebraska	Lincoln
Nevada	Carson City
New Hampshire	Concord
New Jersey	Trenton
New Mexico	Santa Fe
New York	Albany
North Carolina	Raleigh
North Dakota	Bismarck

9.10.2. External filtering

If you require more advanced filtering using custom functions or expressions, use the external filtering properties of the `<rich:column>` component.

Use the `filterExpression` attribute to define an expression that can be evaluated as a boolean value. The expression checks if each table entry satisfies the filtering condition when the table is rendered.

Use the `filterMethod` attribute to define a method binding. The method needs to accept an object as a parameter and return a boolean value. Similar to the `filterExpression` attribute, the table is rendered only with those entries that satisfy the filtering condition. By defining a custom filtering method, you can implement complex business logic to filter a table.

Example 9.16. External filtering

```
<rich:dataTable value="#{capitalsBean.capitals}" var="cap" id="table">
  <f:facet name="header">
    <rich:column>
      <h:outputText value="State Name">
    </rich:column>
    <rich:column>
      <h:outputText value="State Time Zone">
    </rich:column>
  </f:facet>
  <rich:column filterMethod="#{filteringBean.filterStates}">
    <f:facet name="header">
      <h:inputText value="#{filteringBean.filterValue}" id="input">
        <a4j:ajax event="keyup" render="table"
          ignoreDupResponses="true" requestDelay="700"/>
      </h:inputText>
    </f:facet>
    <h:outputText value="#{cap.state}"/>
  </rich:column>
  <rich:column filterExpression=
    "#{fn:containsIgnoreCase(cap.timeZone, filteringBean.filterZone)}">
    <f:facet name="header">
      <h:selectOneMenu value="#{filteringBean.filterZone}">
        <f:selectItems value="#{filteringBean.filterZones}" />
        <a4j:ajax event="change" render="table" />
      </h:selectOneMenu>
    </f:facet>
    <h:outputText value="#{cap.timeZone}"/>
  </rich:column>
</rich:dataTable>
```

The example uses a filter expression on the first column and a filter method on the second column.

State Name	State Time Zone
<input type="text" value="n"/>	<input type="text" value="-5"/>
New Hampshire	GMT-5
New Jersey	GMT-5
New York	GMT-5
North Carolina	GMT-5

9.11. Table sorting



Documentation in development

This section is currently under development. Any features it describes may not be available in the current release of RichFaces.

Tables entries can be sorted by defining external sorting algorithms. Refer to [Section 9.3](#), “<rich:column>” for details on using the <rich:column> component in tables.



Sorting non-English tables

To sort a table whose contents are not in English, add the `org.richfaces.datatableUsesViewLocale` context parameter to the project's `web.xml` settings file. Set the value of the context parameter to `true`.

9.11.1. External sorting

Bind the `sortOrder` attribute to bean properties to manage the sorting order externally. The bean must handle all the sorting algorithms. [Example 9.17](#), “*External sorting*” demonstrates table sorting using an external control.

Example 9.17. External sorting

```
<rich:dataTable value="#{dataTableScrollerBean.allCars}"
    var="category" rows="20" id="table" reRender="ds2"
    sortPriority="#{sortingBean.prioritList}">
  <rich:column id="make" sortBy="#{category.make}"
    sortOrder="#{sortingBean.makeDirection}">
    <f:facet name="header">
      <h:outputText styleClass="headerText" value="Make" />
    </f:facet>
    <h:outputText value="#{category.make}" />
  </rich:column>
  <rich:column id="model" sortBy="#{category.model}">
```

```

        sortOrder="#{sortingBean.modelDirection}">
    <f:facet name="header">
        <h:outputText styleClass="headerText" value="Model" />
    </f:facet>
    <h:outputText value="#{category.model}" />
</rich:column>
<rich:column id="price" sortBy="#{category.price}"
    sortOrder="#{sortingBean.priceDirection}">
    <f:facet name="header">
        <h:outputText styleClass="headerText" value="Price" />
    </f:facet>
    <h:outputText value="#{category.price}" />
</rich:column>
<rich:column id="mileage" sortBy="#{category.mileage}"
    sortOrder="#{sortingBean.mileageDirection}">
    <f:facet name="header">
        <h:outputText styleClass="headerText" value="Mileage" />
    </f:facet>
    <h:outputText value="#{category.mileage}" />
</rich:column>
</rich:dataTable>

```

The example uses an external control to manage the table's sorting.

Make	Model	Price	Mileage
Chevrolet	Corvette	21227	32792.0
Chevrolet	Corvette	26659	14208.0
Chevrolet	Corvette	45779	24349.0
Chevrolet	Malibu	15708	74154.0
Chevrolet	Malibu	54037	71257.0
Chevrolet	Malibu	39646	31674.0
Chevrolet	S-10	21599	39670.0
Chevrolet	S-10	18042	65112.0
Chevrolet	S-10	42683	22633.0

When using the `sortMode="multiple"` configuration, set the priority by which columns are sorted with the `sortPriorities` attribute.

Use the `sortExpression` attribute to define a bean property to use for sorting the column. The expression checks each table entry against the sorting expression during rendering.

Trees

Read this chapter for details on components that use tree structures.

10.1. <rich:tree>

The <rich:tree> component provides a hierarchical tree control. Each <rich:tree> component typically consists of <rich:treeNode> child components. The appearance and behavior of the tree and its nodes can be fully customized.

10.1.1. Basic usage

The <rich:tree> component requires the `value` attribute to point to the data model for populating the tree. The data model must be either an `org.richfaces.model.TreeNode` interface, an `org.richfaces.model.TreeDataModel` interface, or a `javax.swing.tree.TreeNode` interface. The `var` attribute declares the variable used for iterating through the data model, so that child <rich:treeNode> components can reference each iteration.

Ideally, the <rich:tree> component needs one or more <rich:treeNode> components to work with the data model. However if no <rich:treeNode> components are provided the tree creates default nodes instead.

Example 10.1. Basic usage

This example demonstrates basic usage of the <rich:tree> component using an `org.richfaces.model.TreeNode` data model.

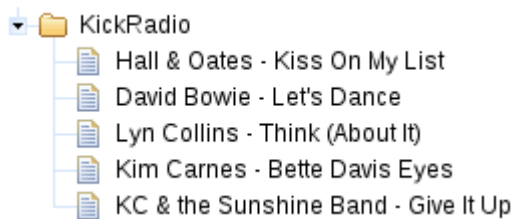
The data model is constructed as follows:

```
private TreeNodeImpl<String> stationRoot = new TreeNodeImpl<String>();
private TreeNodeImpl<String> stationNodes = new TreeNodeImpl<String>();
private String[] kickRadioFeed = { "Hall & Oates - Kiss On My List",
                                   "David Bowie - Let's Dance",
                                   "Lyn Collins - Think (About It)",
                                   "Kim Carnes - Bette Davis Eyes",
                                   "KC & the Sunshine Band - Give It Up" };

stationRoot.setData("KickRadio");
stationNodes.addChild(0, stationRoot);
for (int i = 0; i < kickRadioFeed.length; i++){
    TreeNodeImpl<String> child = new TreeNodeImpl<String>();
    child.setData(kickRadioFeed[i]);
    stationRoot.addChild(i, child);
}
```

The tree then accesses the nodes of the model using the `station` variable:

```
<rich:tree value="#{stations.stationNodes}" var="station">
  <rich:treeNode>
    <h:outputText value="#{station}" />
  </rich:treeNode>
</rich:tree>
```



10.1.2. Appearance

Different nodes in the tree can have different appearances, such as node icons, depending on the type of data the node contains. Use the `nodeType` attribute to differentiate the types of nodes; the node is then rendered according to the `<rich:treeNode>` component with the corresponding `type` attribute. [Example 10.2, “nodeType attribute”](#) shows a `<rich:tree>` component with three different child `<rich:treeNode>` components defined to represent three different node appearances. Refer to [Section 10.1.9.2, “Appearance”](#) for details on customizing the appearance of `<rich:treeNode>` components.

Example 10.2. `nodeType` attribute

```
<rich:tree style="width:300px" value="#{library.data}" var="item" nodeType="#{item.type}">
  <rich:treeNode type="artist" iconExpanded="/images/tree/singer.png" iconCollapsed="/images/tree/singer.png">
    <h:outputText value="#{item.name}" />
  </rich:treeNode>
  <rich:treeNode type="album" iconExpanded="/images/tree/disc.png" iconCollapsed="/images/tree/disc.png">
    <h:outputText value="#{item.album}" />
  </rich:treeNode>
  <rich:treeNode type="song" iconLeaf="/images/tree/song.png">
    <h:outputText value="#{item.title}" />
  </rich:treeNode>
</rich:tree>
```




If the `nodeType` attribute returns null, the node is rendered as a "*typeless*" (or default) node. The *typeless* node is the first child `<rich:treeNode>` component with a valid `rendered` attribute, but without a defined `type` attribute.

If the `nodeType` attribute is not included and there are no child `<rich:treeNode>` components, the tree constructs a default node itself.

Icons for different nodes and node states can be defined for the whole tree using the following attributes:

`iconLeaf`

The `iconLeaf` attribute points to the icon to use for any node that does not contain any child nodes.

`iconExpanded` and `iconCollapsed`

The `iconExpanded` and `iconCollapsed` attributes point to the icons to use for expanded and collapsed nodes respectively. If these attributes are defined, the `icon` attribute is not used.

10.1.3. Expanding and collapsing tree nodes

The mode for performing submissions when nodes are expanded or collapsed is determined by the `toggleType` attribute, which can have one of the following three values:

`ajax`

This is the default setting. The `<rich:tree>` component performs an Ajax form submission, and only the content of the tree is rendered.

`server`

The `<rich:tree>` component performs a common submission, completely re-rendering the page.

`client`

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

By default, the event to expand or collapse a tree node is a mouse click. To specify a different event, use the `toggleNodeEvent` attribute.

10.1.4. Selecting tree nodes

The mode for performing submissions when nodes are selected is determined by the `selectionType` attribute, which can have one of the following three values:

`ajax`

This is the default setting. The `<rich:tree>` component performs an Ajax form submission, and only the content of the tree is rendered.

`server`

The `<rich:tree>` component performs a common submission, completely re-rendering the page.

`client`

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

10.1.5. Identifying nodes with the `rowKeyConverter` attribute

If the `<rich:tree>` component uses a custom data model, the data model provides unique keys for tree nodes so they can be identified during a client request. The `<rich:tree>` component can use strings as key values. These strings may contain special characters that are not allowed by browsers, such as the left angle bracket (`<`) and ampersand (`&`). To allow these characters in the keys, a row key converter must be provided.

To apply a converter to the `<rich:tree>` component, define it with the `rowKeyConverter` attribute.

10.1.6. Event handling

In addition to the standard Ajax events and HTML events, the `<rich:tree>` component uses the following client-side events:

- The `nodetoggle` event is triggered when a node is expanded or collapsed.
- The `beforenodetoggle` event is triggered before a node is expanded or collapsed.
- The `selectionchange` event is triggered when a node is selected.
- The `beforeselectionchange` event is triggered before a node is selected.

The `<rich:tree>` component uses the following server-side listeners:

- The `toggleListener` listener processes expand and collapse events.
- The `selectionChangeListener` listener processes the request when a node is selected.

10.1.7. Reference data

- `component-type: org.richfaces.tree`

- `component-class`: `org.richfaces.component.html.Htmltree`
- `component-family`: `org.richfaces.tree`
- `renderer-type`: `org.richfaces.treeRenderer`
- `tag-class`: `org.richfaces.taglib.treeTag`

10.1.8. Style classes

Styling for the `<rich:tree>` component is mostly applied to the tree nodes. Refer to [Section 10.1.9.5, “Style classes and skin parameters”](#) for details on styling tree nodes. In addition, the `<rich:tree>` component can make use of the style classes outlined in [Style classes \(selectors\)](#).

Style classes (selectors)

`.rf-tr-nd`

This class defines styles for the nodes in a tree.

`.rf-tr-nd-last`

This class defines styles for last node in a tree.

`.rf-tr-nd-colps`

This class defines styles for a collapsed tree node.

`.rf-tr-nd-exp`

This class defines styles for an expanded tree node.

10.1.9. `<rich:treeNode>`

The `<rich:treeNode>` component is a child component of the `<rich:tree>` component. It represents nodes in the parent tree. The appearance and functionality of each tree node can be customized.

10.1.9.1. Basic usage

The `<rich:treeNode>` component must be a child of a `<rich:tree>` component. It does not need any attributes declared for basic usage, but can provide markup templates for tree nodes of particular types. Refer to [Example 10.1, “Basic usage”](#) for an example of basic `<rich:treeNode>` component usage.

10.1.9.2. Appearance

Refer to [Section 10.1.2, “Appearance”](#) for the `<rich:tree>` component for details and examples on styling nodes and icons. Icon styling for individual `<rich:treeNode>` components uses the same attributes as the parent `<rich:tree>` component: `iconLeaf`, `iconExpanded`, and `iconCollapsed`. Icon-related attributes specified for child `<rich:treeNode>` components overwrite any global icon attributes of the parent `<rich:tree>` component.

Use the `rendered` attribute to determine whether the node should actually be rendered in the tree or not. Using the `rendered` attribute in combination with the `<rich:treeNode>` type attribute can allow further style differentiation between node content, as shown in [Example 10.3, “rendered attribute”](#).

Example 10.3. `rendered` attribute

The `rendered` attribute is used to differentiate between music albums that are in stock and those that are not. The item type attributes return values that are otherwise identical; only the `item.exist` property differs, so it is used for the `rendered` attribute.

```
<rich:tree style="width:300px" value="#{library.data}"
  var="item" nodeFace="#{item.type}">
  ...
  <rich:treeNode type="album" iconLeaf="/images/tree/album.gif"
    rendered="#{item.exist}">
    <h:outputText value="#{item.name}" />
  </rich:treeNode>
  <rich:treeNode type="album" iconLeaf="/images/tree/album_absent.gif"
    rendered="#{not item.exist}">
    <h:outputText value="#{item.name}" />
  </rich:treeNode>
  ...
</rich:tree>
```



10.1.9.3. Interactivity

Interactivity with individual nodes, such as expanding, collapsing, and other event handling, can be managed by the parent `<rich:tree>` component. Refer to [Section 10.1.3, “Expanding and collapsing tree nodes”](#) and [Section 10.1.6, “Event handling”](#) for further details.

Use the `expanded` attribute to determine whether the node is expanded or collapsed.

10.1.9.4. Reference data

- *component-type*: `org.richfaces.treeNode`
- *component-class*: `org.richfaces.component.html.HtmlTreeNode`
- *component-family*: `org.richfaces.treeNode`
- *renderer-type*: `org.richfaces.treeNodeRenderer`
- *tag-class*: `org.richfaces.taglib.treeNodeTag`

10.1.9.5. Style classes and skin parameters

Table 10.1. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-trn</code> This class defines styles for a tree node.	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size
<code>.rf-trn-lbl</code> This class defines styles for a tree node label.	No skin parameters.	
<code>.rf-trn-cnt</code> This class defines styles for tree node content.	No skin parameters.	
<code>.rf-trn-sel</code> This class defines styles for a selected tree node.	<i>additionalBackgroundColor</i>	background
<code>.rf-trn-ldn</code> This class defines styles for a tree node when it is loading.	<i>additionalBackgroundColor</i>	background
<code>.rf-trn-hnd</code> This class defines styles for a tree node handle.	No skin parameters.	
<code>.rf-trn-hnd-lf</code> This class defines styles for the handle of a leaf node.	No skin parameters.	
<code>.rf-trn-hnd-colps</code> This class defines styles for the handle of a collapsed node.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-trn-hnd-exp</code> This class defines styles for the handle of an expanded node.	No skin parameters.	
<code>.rf-trn-hnd-ldn-fct</code> This class defines styles for the loading facet of a tree node handle.	No skin parameters.	
<code>.rf-trn-ico</code> This class defines styles for tree node icon.	No skin parameters.	
<code>.rf-trn-ico-lf</code> This class defines styles for the icon of a leaf node.	No skin parameters.	
<code>.rf-trn-ico-colps</code> This class defines styles for the icon of a collapsed node.	No skin parameters.	
<code>.rf-trn-ico-exp</code> This class defines styles for the icon of an expanded node.	No skin parameters.	
<code>.rf-trn-ico-cst</code> This class defines styles for a custom node icon.	No skin parameters.	

10.2. Tree adaptors

Use a tree adaptor to populate a tree model declaratively from a non-hierarchical model, such as a list or a map.

10.2.1. `<rich:treeModelAdaptor>`

The `<rich:treeModelAdaptor>` component takes an object which implements the `Map` or `Iterable` interfaces. It adds all the object entries to the parent node as child nodes.

10.2.1.1. Basic usage

The `<rich:treeModelAdaptor>` component is added as a nested child component to a `<rich:tree>` component, or to another tree adaptor component.

The `<rich:treeModelAdaptor>` component requires the `nodes` attribute for basic usage. The `nodes` attribute defines a collection of elements to iterate through for populating the nodes.

Define the appearance of each node added by the adaptor with a child `<rich:treeNode>` component. Refer to [Section 10.1.9, “<rich:treeNode>”](#) for details on the `<rich:treeNode>` component.

10.2.1.2. Multiple levels

`<rich:treeModelAdaptor>` components can further be nested in other `<rich:treeModelAdaptor>` components to subsequently populate lower levels of the tree.

To access the current element at each iteration, use the `var` attribute of either the parent `<rich:tree>` component or the `<rich:treeModelAdaptor>` component itself. [Example 10.4, “Nested <rich:treeModelAdaptor> components”](#) demonstrates a series of nested `<rich:treeModelAdaptor>` components, each using the parent's `var` attribute to reference the current element.

Example 10.4. Nested `<rich:treeModelAdaptor>` components

```
<rich:tree>

  <rich:treeNodesAdaptor id="project" nodes="#{loaderBean.projects}" var="project">
    <rich:treeNode>
      <h:commandLink action="#{project.click}" value="Project:
#{project.name}" />
    </rich:treeNode>
    <rich:treeNodesAdaptor id="srcDir" var="srcDir" nodes="#{project.srcDirs}">
      <rich:treeNode>
        <h:commandLink action="#{srcDir.click}" value="Source directory:
#{srcDir.name}" />
      </rich:treeNode>
      <rich:treeNodesAdaptor id="pkg" var="pkg" nodes="#{srcDir.packages}">
        <rich:treeNode>
          <h:commandLink action="#{pkg.click}" value="Package: #{pkg.name}" />
        </rich:treeNode>
        <rich:treeNodesAdaptor id="class" var="class" nodes="#{pkg.classes}">
          <rich:treeNode>
            <h:commandLink action="#{class.click}" value="Class:
#{class.name}" />
          </rich:treeNode>
        </rich:treeNodesAdaptor>
      </rich:treeNodesAdaptor>
    </rich:treeNodesAdaptor>
  </rich:treeNodesAdaptor>
</rich:tree>
```

Each `<rich:treeModelAdaptor>` component is mapped to a list of objects. During the iteration, the corresponding object properties are used to define the node labels and actions, and are in turn used for iterating through nested lists.

10.2.1.3. Identifying nodes

Adaptors that use `Map` interfaces or models with non-string keys require a row key converter in order to correctly identify nodes. Refer to [Section 10.1.5, “Identifying nodes with the `rowKeyConverter` attribute](#)” for details on the use of the `rowKeyConverter` attribute.

Adaptors that use `Iterable` interfaces have simple integer row keys. A default converter is provided and does not need to be referenced explicitly.

10.2.1.4. Reference data

- `component-type`: `org.richfaces.treeModelAdaptor`
- `component-class`: `org.richfaces.component.html.HtmltreeModelAdaptor`
- `component-family`: `org.richfaces.treeModelAdaptor`
- `tag-class`: `org.richfaces.taglib.treeModelAdaptorTag`

10.2.2. <rich:treeModelRecursiveAdaptor>

The `<rich:treeModelRecursiveAdaptor>` component iterates through recursive collections in order to populate a tree with hierarchical nodes, such as for a file system with multiple levels of directories and files.

10.2.2.1. Basic usage

The `<rich:treeModelRecursiveAdaptor>` component is an extension of the `<rich:treeModelAdaptor>` component. As such, the `<rich:treeModelRecursiveAdaptor>` component uses all of the same attributes. Refer to [Section 10.2.1, “<rich:treeModelAdaptor>”](#) for details on the `<rich:treeModelAdaptor>` component.

In addition, the `<rich:treeModelRecursiveAdaptor>` component requires the `root` attribute. The `root` attribute defines the collection to use at the top of the recursion. For subsequent levels, the `node` attribute is used for the collection.

Example 10.5, “Basic usage” demonstrates how the `<rich:treeModelRecursiveAdaptor>` component can be used in conjunction with the `<rich:treeModelAdaptor>` component to recursively iterate through a file system and create a tree of directories and files.

Example 10.5. Basic usage

```
<rich:tree var="item">

  <rich:treeModelRecursiveAdaptor roots="#{fileSystemBean.sourceRoots}" nodes="#{item.directories}">
    <rich:treeNode>
      #{item.shortPath}
    </rich:treeNode>
  </rich:treeModelRecursiveAdaptor>
  <rich:treeModelAdaptor nodes="#{item.files}">
```



```

        <rich:treeNode>#{item}</rich:treeNode>
    </rich:treeModelAdaptor>
</rich:treeModelRecursiveAdaptor>
</rich:tree>

```

The <rich:treeModelRecursiveAdaptor> component references the `FileSystemBean` class as the source for the data.

```

@ManagedBean
@RequestScoped
public class FileSystemBean {
    private static final String SRC_PATH = "/WEB-INF";

    private List<FileSystemNode> srcRoots;

    public synchronized List<FileSystemNode> getSourceRoots() {
        if (srcRoots == null) {
            srcRoots = new FileSystemNode(SRC_PATH).getDirectories();
        }
        return srcRoots;
    }
}

```

The `FileSystemBean` class in turn uses the `FileSystemNode` class to recursively iterate through the collection.

```

public class FileSystemNode {
    ...
    public synchronized List<FileSystemNode> getDirectories() {
        if (directories == null) {
            directories = Lists.newArrayList();
            Iterables.addAll(directories,transform(filter(getResourcePaths(),containsPattern("/
            $")), FACTORY));
        }
        return directories;
    }

    public synchronized List<String> getFiles() {
        if (files == null) {
            files = new ArrayList<String>();
            Iterables.addAll(files,transform(filter(getResourcePaths(),not(containsPattern("/
            $"))), TO_SHORT_PATH));
        }
        return files;
    }
}

```

```

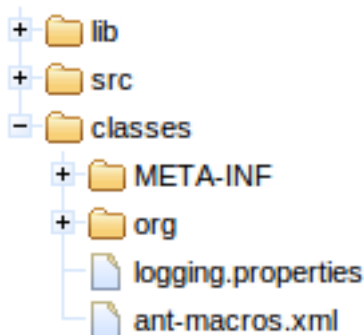
private Iterable<String> getResourcePaths() {
    FacesContext facesContext = FacesContext.getCurrentInstance();
    ExternalContext externalContext = facesContext.getExternalContext();
    Set<String> resourcePaths = externalContext.getResourcePaths(this.path);

    if (resourcePaths == null) {
        resourcePaths = Collections.emptySet();
    }
    return resourcePaths;
}
...
}

```

The `getDirectories()` function is used recursively until the object has the collection of children. The model adaptor calls the `getFiles()` function at each level in order to add the file nodes.

The resulting tree hierarchically lists the directories and files in the collection.



10.2.2.2. Identifying nodes

Adaptors that use `Map` interfaces or models with non-string keys require a row key converter in order to correctly identify nodes. Refer to [Section 10.1.5, “Identifying nodes with the `rowKeyConverter` attribute”](#) for details on the use of the `rowKeyConverter` attribute.

Adaptors that use `Iterable` interfaces have simple integer row keys. A default converter is provided and does not need to be referenced explicitly.

10.2.2.3. Reference data

- *component-type*: `org.richfaces.treeModelRecursiveAdaptor`
- *component-class*: `org.richfaces.component.html.HtmltreeModelRecursiveAdaptor`
- *component-family*: `org.richfaces.treeModelRecursiveAdaptor`
- *tag-class*: `org.richfaces.taglib.treeModelRecursiveAdaptorTag`

Menus and toolbars

Read this chapter for details on menu and toolbar components.

11.1. <rich:dropDownMenu>

The <rich:dropDownMenu> component is used for creating a drop-down, hierarchical menu. It can be used with the <rich:toolbar> component to create menus in an application's toolbar.

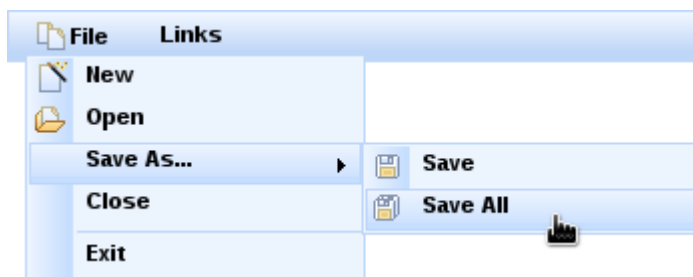


Figure 11.1. <rich:dropDownMenu>

11.1.1. Basic usage

The <rich:dropDownMenu> component only requires the `label` attribute for basic usage. Use the `label` attribute to define the text label that appears as the title of the menu. Clicking on the title drops the menu down.

Alternatively, use the `label` facet to define the menu title. If the `label` facet is used, the `label` attribute is not necessary.

11.1.2. Menu content

To set the content of the drop-down menu and any sub-menus, use the <rich:menuItem>, <rich:menuGroup>, and <rich:menuSeparator> components. These components are detailed in [Section 11.2, “Menu sub-components”](#).

11.1.3. Appearance

Use the `jointPoint` and `direction` attributes to determine the direction and location of the menu when it appears. The `jointPoint` and `direction` attributes both use the following settings:

`topLeft`, `topRight`, `bottomLeft`, `bottomRight`

When used with the `jointPoint` attribute, the menu is attached to the top-left, top-right, bottom-left, or bottom-right of the control as appropriate.

When used with the `direction` attribute, the menu appears to the top-left, top-right, bottom-left, or bottom-right of the joint location as appropriate.

`auto`

The direction or joint location is determined automatically.

`autoLeft`, `autoRight`, `autoTop`, `autoBottom`

When used with the `jointPoint` attribute, the joint location is determined automatically, but defaults to either the left, right, top, or bottom of the control as appropriate.

When used with the `direction` attribute, the menu direction is determined automatically, but defaults to either the left, right, top, or bottom of the joint location as appropriate.

11.1.4. Expanding and collapsing the menu

By default, the menu drops down when the title is clicked. To drop down with a different event, use the `eventShow` attribute to define the event instead.

Menus can be navigated using the keyboard. Additionally, menus can be navigated programmatically using the JavaScript API. The JavaScript API allows the following methods:

`show()`

The `show()` method shows the menu.

`hide()`

The `hide()` method hides the menu.

`activateItem(menuItemId)`

The `activateItem(menuItemId)` activates the menu item with the `menuItemId` identifier.

Use the `mode` attribute to determine how the menu requests are submitted:

- `server`, the default setting, submits the form normally and completely refreshes the page.
- `ajax` performs an Ajax form submission, and re-renders elements specified with the `render` attribute.
- `client` causes the `action` and `actionListener` items to be ignored, and the behavior is fully defined by the nested components or custom JavaScript instead of responses from submissions.

11.1.5. Reference data

- `component-type`: `org.richfaces.DropDownMenu`
- `component-class`: `org.richfaces.component.html.HtmlDropDownMenu`
- `component-family`: `org.richfaces.DropDownMenu`
- `renderer-type`: `org.richfaces.DropDownMenuRenderer`
- `tag-class`: `org.richfaces.taglib.DropDownMenuTag`

11.1.6. Style classes and skin parameters

Table 11.1. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-ddm-lbl This class defines styles for the label of the drop-down menu.	<i>headerFamilyFont</i>	font-family
.rf-ddm-dis This class defines styles for the drop-down menu when it is disabled.	<i>tabDisabledTextColor</i>	color
.rf-ddm-lbl-dis This class defines styles for the label of the drop-down menu when it is disabled.	<i>headerFamilyFont</i>	font-family
.rf-ddm-pos This class defines the positioning of the drop-down menu.	No skin parameters.	
.rf-ddm-lbl-unsel This class defines styles for the label of the drop-down menu when it is unselected.	No skin parameters.	
.rf-ddm-lst This class defines styles for the drop-down list.	<i>panelBorderColor</i>	border-color
	<i>additionalBackgroundColor</i>	background-color
.rf-ddm-lst-bg This class defines styles for the background of the drop-down list.	<i>additionalBackgroundColor</i>	border-color
.rf-ddm-sublst This class defines the positioning of the menu when used as a sub-menu.	No skin parameters.	
.rf-ddm-itm This class defines styles for a menu item.	<i>generalFamilyFont</i>	font-family
	<i>generalSizeFont</i>	font-size

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-ddm-itm-sel</code> This class defines styles for a menu item when it is selected.	<code>headerBackgroundColor</code>	<code>border-color</code>
	<code>tabBackgroundColor</code>	<code>background-color</code>
<code>.rf-ddm-itm-unsel</code> This class defines styles for a menu item when it is unselected.	No skin parameters.	
<code>.rf-ddm-itm-dis</code> This class defines styles for a menu item when it is disabled.	<code>tabDisabledTextColor</code>	<code>color</code>
<code>.rf-ddm-itm-lbl</code> This class defines styles for the label in a menu item.	<code>generalTextColor</code>	<code>color</code>
<code>.rf-ddm-itm-ic</code> This class defines styles for the icon in a menu item.	No skin parameters.	
<code>.rf-ddm-emptyIcon</code> This class defines styles for an empty icon in a menu item.	No skin parameters.	
<code>.rf-ddm-sep</code> This class defines styles for a menu separator.	<code>panelBorderColor</code>	<code>border-top-color</code>
<code>.rf-ddm-nd</code> This class defines styles for a menu node.	No skin parameters.	

11.2. Menu sub-components

The `<rich:menuItem>`, `<rich:menuGroup>`, and `<rich:menuSeparator>` components are used to construct menus for the `<rich:dropDownMenu>` component. Refer to [Section 11.1, “<rich:dropDownMenu>”](#) for more details on the `<rich:dropDownMenu>` component.

11.2.1. `<rich:menuItem>`

The `<rich:menuItem>` component represents a single item in a menu control. The `<rich:menuItem>` component can also be used as a separate component without a parent menu component, such as on a toolbar.

11.2.1.1. Basic usage

The `<rich:menuItem>` component requires the `label` attribute for basic usage. The `label` attribute is the text label for the menu item.

11.2.1.2. Appearance

Icons can be added to menu items through the use of two icon attributes. The `icon` attribute specifies the normal icon, while the `iconDisabled` attribute specifies the icon for a disabled item.

Alternatively, define facets with the names `icon` and `iconDisabled` to set the icons. If facets are defined, the `icon` and `iconDisabled` attributes are ignored. Using facets for icons allows more complex usage; example shows a checkbox being used in place of an icon.

Example 11.1. Icon facets

```
<rich:menuItem value="Show comments">
  <f:facet name="icon">
    <h:selectBooleanCheckbox value="#{bean.property}" />
  </f:facet>
</rich:menuItem>
```

11.2.1.3. Submission modes

Use the `submitMode` attribute to determine how the menu item requests are submitted:

- `server`, the default setting, submits the form normally and completely refreshes the page.
- `ajax` performs an Ajax form submission, and re-renders elements specified with the `render` attribute.
- `client` causes the `action` and `actionListener` items to be ignored, and the behavior is fully defined by the nested components instead of responses from submissions.

11.2.1.4. JavaScript API

The `<rich:menuItem>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`activate()`

Activate the menu item as though it were selected.

11.2.1.5. Reference data

- `component-type`: `org.richfaces.MenuItem`
- `component-class`: `org.richfaces.component.html.HtmlMenuItem`

- *component-family*: `org.richfaces.DropDownMenu`
- *renderer-type*: `org.richfaces.MenuItemRenderer`
- *tag-class*: `org.richfaces.taglib.MenuItemTag`

11.2.2. <rich:menuGroup>

The `<rich:menuGroup>` component represents an expandable sub-menu in a menu control. The `<rich:menuGroup>` component can contain a number of `<rich:menuItem>` components, or further nested `<rich:menuGroup>` components.

11.2.2.1. Basic usage

The `<rich:menuGroup>` component requires the `value` attribute for basic usage. The `value` attribute is the text label for the menu item.

Additionally, the `<rich:menuGroup>` component must contain child `<rich:menuItem>` components or `<rich:menuGroup>` components.

11.2.2.2. Appearance

Icons can be added to menu groups through the use of two icon attributes. The `icon` attribute specifies the normal icon, while the `iconDisabled` attribute specifies the icon for a disabled group.

The `<rich:menuGroup>` component can be positioned using the `jointPoint` and `direction` attributes, the same as the parent menu control. For details on the `jointPoint` and `direction` attributes, refer to [Section 11.1.3, “Appearance”](#).

11.2.2.3. JavaScript API

The `<rich:menuGroup>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`show()`
Show the menu group.

`hide()`
Hide the menu group.

11.2.2.4. Reference data

- *component-type*: `org.richfaces.MenuGroup`
- *component-class*: `org.richfaces.component.html.HtmlMenuGroup`
- *component-family*: `org.richfaces.DropDownMenu`
- *renderer-type*: `org.richfaces.MenuGroupRenderer`
- *tag-class*: `org.richfaces.taglib.MenuGroupTag`

11.2.3. <rich:menuSeparator>

The <rich:menuSeparator> component represents a separating divider in a menu control.

11.2.3.1. Basic usage

The <rich:menuSeparator> component does not require any attributes for basic usage. Add it as a child to a menu component to separator menu items and menu groups.

11.2.3.2. Reference data

- *component-type*: org.richfaces.MenuSeparator
- *component-class*: org.richfaces.component.html.HtmlMenuSeparator
- *component-family*: org.richfaces.DropDownMenu
- *renderer-type*: org.richfaces.MenuSeparatorRenderer
- *tag-class*: org.richfaces.taglib.MenuSeparatorTag

11.3. <rich:panelMenu>

The <rich:panelMenu> component is used in conjunction with <rich:panelMenuItem> and <rich:panelMenuGroup> to create an expanding, hierarchical menu. The <rich:panelMenu> component's appearance can be highly customized, and the hierarchy can stretch to any number of sub-levels.

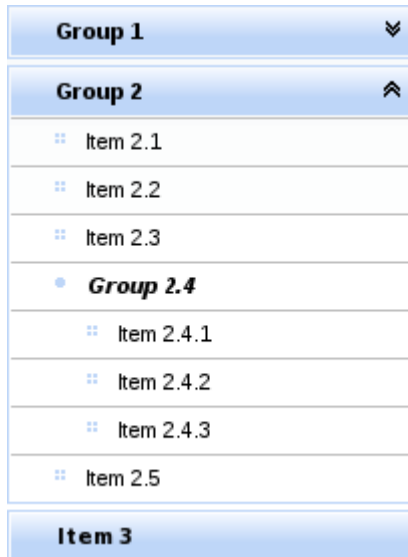
Example 11.2. richpanelMenu

```
<rich:panelMenu mode="ajax"
    topGroupExpandedRightIcon="chevronUp"
    topGroupCollapsedRightIcon="chevronDown"
    groupExpandedLeftIcon="disc"
    groupCollapsedLeftIcon="disc">
    <rich:panelMenuGroup label="Group 1">
        <rich:panelMenuItem label="Item 1.1"/>
        <rich:panelMenuItem label="Item 1.2"/>
        <rich:panelMenuItem label="Item 1.3"/>
    </rich:panelMenuGroup>
    <rich:panelMenuGroup label="Group 2">
        <rich:panelMenuItem label="Item 2.1"/>
        <rich:panelMenuItem label="Item 2.2"/>
        <rich:panelMenuItem label="Item 2.3"/>
        <rich:panelMenuGroup label="Group 2.4">
            <rich:panelMenuItem label="Item 2.4.1"/>
            <rich:panelMenuItem label="Item 2.4.2"/>
            <rich:panelMenuItem label="Item 2.4.3"/>
        </rich:panelMenuGroup>
    </rich:panelMenuGroup>
```

```

    <rich:panelMenuItem label="Item 2.5"/>
  </rich:panelMenuGroup>
  <rich:panelMenuItem label="Item 3"/>
</rich:panelMenu>

```



11.3.1. Basic usage

The `<rich:panelMenu>` component does not need any extra attributes declared for basic usage. However, it does require child `<rich:panelMenuGroup>` and `<rich:panelMenuItem>` components. Refer to [Section 11.3.9, “<rich:panelMenuGroup>”](#) and [Section 11.3.10, “<rich:panelMenuItem>”](#) for details on these child components.

11.3.2. Interactivity options

The `activeItem` attribute is used to point to the name of the currently selected menu item.

By default, the event to expand the menu is a mouse click. Set the `expandEvent` attribute to specify a different event to expand menus. Multiple levels of sub-menus can be expanded in one action. Set `expandSingle="true"` to only expand one sub-menu at a time.

Similarly, the default event to collapse the menu is a mouse click. Set the `collapseEvent` attribute to specify a different event to collapse menus.

As with other control components, set `disabled="true"` to disable the `<rich:panelMenu>` component. Child menu components can be disabled in the same way.

11.3.3. Appearance

Icons for the panel menu can be chosen from a set of standard icons. Icons can be set for the top panel menu, child panel menus, and child item. There are three different menu states that the icon represents, as well as icons for both the left and right side of the item title.

`topGroupExpandedLeftIcon`, `topGroupExpandedRightIcon`

These attributes determine the icons for the top level menu when it is expanded.

`topGroupCollapsedLeftIcon`, `topGroupCollapsedRightIcon`

These attributes determine the icons for the top level menu when it is collapsed.

`topGroupDisabledLeftIcon`, `topGroupDisabledRightIcon`

These attributes determine the icons for the top level menu when it is disabled.

`topItemLeftIcon`, `topItemRightIcon`

These attributes determine the icons for a top level menu item.

`topItemDisabledLeftIcon`, `topItemDisabledRightIcon`

These attributes determine the icons for a top level menu item when it is disabled.

`groupExpandedLeftIcon`, `groupExpandedRightIcon`

These attributes determine the icons for sub-menus that are not the top-level menu when they are expanded.

`groupCollapsedLeftIcon`, `groupCollapsedRightIcon`

These attributes determine the icons for sub-menus that are not the top-level menu when they are collapsed.

`groupDisabledLeftIcon`, `groupDisabledRightIcon`

These attributes determine the icons for sub-menus that are not the top-level menu when they are disabled.

`itemLeftIcon`, `itemRightIcon`

These attributes determine the icons for items in the menus.

`itemDisabledLeftIcon`, `itemDisabledRightIcon`

These attributes determine the icons for items in the menus when they are disabled.

[Example 11.2](#), “*richpanelMenu*” demonstrates the use of icon declaration at the panel menu level. The standard icons are shown in [Figure 11.2](#), “<Standard icons>”.



Figure 11.2. <Standard icons>

Alternatively, point the icon attributes to the paths of image files. The image files are then used as icons.

Any icons specified by child `<rich:panelMenuGroup>` and `<rich:panelMenuItem>` components overwrite the relevant icons declared with the parent `<rich:panelMenu>` component.

11.3.4. Submission modes

The `itemMode` attribute defines the submission mode for normal menu items that link to content, and the `groupMode` attribute defines the submission mode for menu items that expand and collapse. The settings for these attributes apply to the entire menu unless a menu item defines its own individual `itemMode` or `groupMode`. The possible values for `itemMode` and `groupMode` are as follows:

- `server`, the default setting, which submits the form normally and completely refreshes the page.
- `ajax`, which performs an Ajax form submission, and re-renders elements specified with the `render` attribute.
- `client`, which causes the `action` and `actionListener` items to be ignored, and the behavior is fully defined by the nested components instead of responses from submissions.

11.3.5. `<rich:panelMenu>` server-side events

The `<rich:panelMenu>` component fires the `ItemChangeEvent` event on the server side when the menu is changed. The event only fires in the `server` and `ajax` submission modes. The event provides the `itemChangeListener` attribute to reference the event listener.

11.3.6. JavaScript API

The `<rich:panelMenu>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`expandAll()`

Expand all the panel menu groups in the component.

`collapseAll()`

Collapse all the panel menu groups in the component.

`selectItem(id)`

Select the menu item with the `id` identifier.

11.3.7. Reference data

- `component-type`: `org.richfaces.PanelMenu`
- `component-class`: `org.richfaces.component.html.HtmlPanelMenu`
- `component-family`: `org.richfaces.PanelMenu`
- `renderer-type`: `org.richfaces.PanelMenuRenderer`
- `tag-class`: `org.richfaces.taglib.PanelMenuTag`

11.3.8. Style classes and skin parameters

Table 11.2. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters		Mapped CSS properties
.rf-pm This class defines styles for the panel menu itself.	No skin parameters.		
.rf-pm-gr This class defines styles for a panel menu group.	<i>panelBorderColor</i>		border-top-color
.rf-pm-exp, .rf-pm-colps These classes define styles for the panel menu when it is expanded or collapsed.	No skin parameters.		
.rf-pm-ico This class defines styles for the panel menu icons.	No skin parameters.		
.rf-pm-ico-exp, .rf-pm-ico-colps These classes define styles for the panel menu icons when they are expanded or collapsed.	No skin parameters.		
.rf-pm-hdr-exp, .rf-pm-hdr-colps These classes define styles for the panel menu headers when they are expanded or collapsed.	No skin parameters.		
.rf-pm-itm This class defines styles for a panel menu item.	<i>panelBorderColor</i>		border-top-color
	<i>generalTextColor</i>		color
.rf-pm-itm-gr This class defines styles for a panel menu item as part of a panel menu group.	No skin parameters.		
.rf-pm-itm:hover This class defines styles for a panel menu item	<i>additionalBackgroundColor</i>		background-color

Class (selector)	Skin Parameters	Mapped CSS properties
when the mouse hovers over it.		
.rf-pm-itm-sel This class defines styles for a panel menu item when it is selected.	No skin parameters.	
.rf-pm-itm-dis This class defines styles for a panel menu item when it is disabled.	<i>tabDisabledTextColor</i>	color
.rf-pm-itm-ico This class defines styles for the icon in a panel menu item.	No skin parameters.	
.rf-pm-itm-exp-ico This class defines styles for the icon in a panel menu item when it is expanded.	No skin parameters.	
.rf-pm-itm-lbl This class defines styles for the label in a panel menu item.	<i>generalSizeFont</i>	font-size
	<i>generalFamilyFont</i>	font-family
.rf-pm-gr This class defines styles for a panel menu group.	<i>panelBorderColor</i>	border-top-color
.rf-pm-gr-gr This class defines styles for a panel menu group as part of another panel menu group.	No skin parameters.	
.rf-pm-gr-sel This class defines styles for a panel menu group when it is selected.	No skin parameters.	
.rf-pm-gr-hdr This class defines styles for the header of a panel menu group.	<i>generalTextColor</i>	color

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-pm-gr-hdr:hover This class defines styles for the header of a panel menu group when the mouse hovers over it.	<i>additionalBackgroundColor</i>	background
.rf-pm-gr-hdr-dis This class defines styles for the header of a panel menu group when it is disabled.	<i>tabDisabledTextColor</i>	color
.rf-pm-gr-ico This class defines styles for the icon in a panel menu group.	No skin parameters.	
.rf-pm-gr-exp-ico This class defines styles for the icon in a panel menu group when it is expanded.	No skin parameters.	
.rf-pm-gr-lbl This class defines styles for the label in a panel menu group.	<i>generalSizeFont</i>	font-size
	<i>generalFamilyFont</i>	font-family
.rf-pm-gr-cnt This class defines styles for the content of a panel menu group.	No skin parameters.	
.rf-pm-top-itm This class defines styles for the top panel menu item.	<i>panelBorderColor</i>	border-color
	<i>generalTextColor</i>	color
.rf-pm-top-itm-gr This class defines styles for the top panel menu item as part of a panel menu group.	No skin parameters.	
.rf-pm-top-itm:hover This class defines styles for the top panel menu item when the mouse hovers over it.	<i>headerTextColor</i>	color

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-pm-top-itm-sel This class defines styles for the top panel menu item when it is selected.	No skin parameters.	
.rf-pm-top-itm-dis This class defines styles for the top panel menu item when it is disabled.	<i>tabDisabledTextColor</i>	color
.rf-pm-top-itm-ico This class defines styles for the icon in the top panel menu item.	No skin parameters.	
.rf-pm-top-itm-exp-ico This class defines styles for the icon in the top panel menu item when it is expanded.	No skin parameters.	
.rf-pm-top-itm-lbl This class defines styles for the label in the top panel menu item.	<i>generalSizeFont</i>	font-size
	<i>generalFamilyFont</i>	font-family
.rf-pm-top-gr This class defines styles for the top panel menu group.	<i>panelBorderColor</i>	border-color
.rf-pm-top-gr-gr This class defines styles for the top panel menu group as part of another panel menu group.	No skin parameters.	
.rf-pm-top-gr-sel This class defines styles for the top panel menu group when it is selected.	No skin parameters.	
.rf-pm-top-gr-hdr This class defines styles for the header of the top panel menu group.	<i>headerTextColor</i>	color
	<i>headerBackgroundColor</i>	background-color

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-pm-top-gr-hdr-dis</code> This class defines styles for the header of the top panel menu group when it is disabled.	<code>tabDisabledTextColor</code>	color
	<code>additionalBackgroundColor</code>	background-color
<code>.rf-pm-top-gr-ico</code> This class defines styles for the icon in the top panel menu group.	No skin parameters.	
<code>.rf-pm-top-gr-exp-ico</code> This class defines styles for the icon in the top panel menu group when it is expanded.	No skin parameters.	
<code>.rf-pm-top-gr-lbl</code> This class defines styles for the label in the top panel menu group.	<code>generalSizeFont</code>	font-size
	<code>generalFamilyFont</code>	font-family
<code>.rf-pm-top-gr-cnt</code> This class defines styles for the content of the top panel menu group.	No skin parameters.	

11.3.9. <rich:panelMenuGroup>

The `<rich:panelMenuGroup>` component defines a group of `<rich:panelMenuItem>` components inside a `<rich:panelMenu>`.

11.3.9.1. Basic usage

The `<rich:panelMenuGroup>` component needs the `label` attribute declared, which specifies the text to show for the menu entry. Alternatively, the `label` facet can be used to specify the menu text.

In addition, the `<rich:panelMenuGroup>` component at least one `<rich:panelMenuGroup>` or `<rich:panelMenuItem>` components as child elements.

11.3.9.2. Appearance

Icons for the menu group are inherited from the parent `<rich:panelMenu>` component. Refer to [Section 11.3.3, “Appearance”](#) for details on icon attributes and facets. Alternatively, the menu group's icons can be re-defined at the `<rich:panelMenuGroup>` component level, and these settings will be used instead of the parent component's settings.

11.3.9.3. Submission modes

If the `mode` attribute is unspecified, the submission behavior for the group is inherited from the parent `<rich:panelMenu>`. Otherwise, the `mode` setting is used instead of the parent's behavior. Refer to [Section 11.3.4, “Submission modes”](#) for submission mode settings.

11.3.9.4. `<rich:panelMenuGroup>` server-side events

The `<rich:panelMenuGroup>` component fires the `ActionEvent` event on the server side when the menu group receives a user action. The event only fires in the `server` and `ajax` submission modes. The event provides the `action` attribute to specify the user action performed, and the `actionListener` attribute to reference the event listener.

11.3.9.5. JavaScript API

The `<rich:panelMenuGroup>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`expand()`

Expand this panel menu group.

`collapse()`

Collapse this panel menu group.

`select(id)`

Select the menu item with the `id` identifier.

11.3.9.6. Reference data

- `component-type`: `org.richfaces.PanelMenuGroup`
- `component-class`: `org.richfaces.component.html.HtmlPanelMenuGroup`
- `component-family`: `org.richfaces.PanelMenuGroup`
- `renderer-type`: `org.richfaces.PanelMenuGroupRenderer`
- `tag-class`: `org.richfaces.taglib.PanelMenuGroupTag`

11.3.10. `<rich:panelMenuItem>`

The `<rich:panelMenuItem>` component represents a single item inside a `<rich:panelMenuGroup>` component, which is in turn part of a `<rich:panelMenu>` component.

11.3.10.1. Basic usage

The `<rich:panelMenuItem>` component needs the `label` attribute declared, which specifies the text to show for the menu entry. Alternatively, the `label` facet can be used to specify the menu text.

11.3.10.2. Appearance

Icons for the menu item are inherited from the parent `<rich:panelMenu>` or `<rich:panelMenuGroup>` component. Refer to [Section 11.3.3, “Appearance”](#) for details on icon attributes and facets. Alternatively, the menu item's icons can be re-defined at the `<rich:panelMenuItem>` component level, and these settings will be used instead of the parent component's settings.

11.3.10.3. Submission modes

If the `mode` is unspecified, the submission behavior for the item is inherited from the parent `<rich:panelMenu>`. Otherwise, the `mode` setting is used instead of the parent's behavior.

11.3.10.4. <rich:panelMenuItem> server-side events

The `<rich:panelMenuItem>` component fires the `ActionEvent` event on the server side when the menu item receives a user action. The event only fires in the `server` and `ajax` submission modes. The event provides the `action` attribute to specify the user action performed, and the `actionListener` attribute to reference the event listener.

11.3.10.5. JavaScript API

The `<rich:panelMenuItem>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

`select()`
Select this menu item.

11.3.10.6. Reference data

- `component-type`: `org.richfaces.PanelMenuItem`
- `component-class`: `org.richfaces.component.html.HtmlPanelMenuItem`
- `component-family`: `org.richfaces.PanelMenuItem`
- `renderer-type`: `org.richfaces.PanelMenuItemRenderer`
- `tag-class`: `org.richfaces.taglib.PanelMenuItemTag`

11.4. <rich:toolbar>

The `<rich:toolbar>` component is a horizontal toolbar. Any JavaServer Faces (JSF) component can be added to the toolbar.



Figure 11.3. `<rich:toolbar>`

11.4.1. Basic usage

The `<rich:toolbar>` component does not require any attributes to be defined for basic usage. Add child components to the `<rich:toolbar>` component to have them appear on the toolbar when rendered.





Example 11.3. Basic usage

```
<rich:toolbar>
  <h:commandLink value="News" />
  <h:commandLink value="Reviews" />
  <h:commandLink value="Galleries" />
</rich:toolbar>
```

11.4.2. Appearance

Set the width and height of the toolbar using the common `width` and `height` attributes.

Items on the toolbar can be separated by a graphical item separator. Use the `itemSeparator` attribute to specify one of the standard separator styles:

- `none`, the default appearance, does not show any item separators.
- `disc` shows a small circular disc to separate items:

- `grid` shows a grid pattern to separate items:

- `line` shows a vertical line to separate items:

- `square` shows a small square to separate items:


Alternatively, use the `itemSeparator` attribute to specify a URL to an image. The image is then used as an item separator. The appearance of the item separator can be additionally customized by using the `itemSeparator facet`.

11.4.3. Grouping items

Group together multiple items on the toolbar by using the `<rich:toolbarGroup>` child component. Refer to [Section 11.4.6, “<rich:toolbarGroup>”](#) for full details on the `<rich:toolbarGroup>` component.

11.4.4. Reference data

- `component-type`: `org.richfaces.Toolbar`
- `component-class`: `org.richfaces.component.html.HtmlToolbar`
- `component-family`: `org.richfaces.Toolbar`
- `renderer-type`: `org.richfaces.ToolbarRenderer`
- `tag-class`: `org.richfaces.taglib.ToolbarTag`

11.4.5. Style classes and skin parameters

Table 11.3. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-tb</code> This class defines styles for the toolbar itself.	<code>panelBorderColor</code>	<code>border-color</code>
	<code>headerTextColor</code>	<code>color</code>
	<code>headerBackgroundColor</code>	<code>background-color</code>
	<code>headerFamilyFont</code>	<code>font-family</code>
	<code>headerSizeFont</code>	<code>font-size</code>
	<code>headerWeightFont</code>	<code>font-weight</code>
<code>.rf-tb-itm</code> This class defines styles for an item in the toolbar.	No skin parameters.	
<code>.rf-tb-sep</code> This class defines styles for a separator in the toolbar.	No skin parameters.	
<code>.rf-tb-sep-grid</code> , <code>.rf-tb-sep-line</code> , <code>.rf-tb-sep-disc</code> , <code>.rf-tb-sep-square</code> These classes define styles for grid, line, disc, and square separators.	No skin parameters.	

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-tb-cntr This class defines styles for the container of the toolbar.	No skin parameters.	

11.4.6. <rich:toolbarGroup>

The <rich:toolbarGroup> component is a child component of the <rich:toolbar> component. The <rich:toolbarGroup> component is used to group a number of items together on a toolbar.

11.4.6.1. Basic usage

Like the <rich:toolbar> parent component, the <rich:toolbarGroup> component does not require any extra attributes for basic functionality. Add child components to the <rich:toolbarGroup> component to have them appear grouped on the parent toolbar when rendered.

11.4.6.2. Appearance

Similar to the <rich:toolbar> component, items within a <rich:toolbarGroup> can be separated by specifying the `itemSeparator` attribute. Refer to [Section 11.4.2, “Appearance”](#) for details on using the `itemSeparator` attribute.

Groups of toolbar items can be located on either the left-hand side or the right-hand side of the parent toolbar. By default, they appear to the left. To locate the toolbar group to the right of the parent toolbar, set `location="right"`.

Example 11.4. <rich:toolbarGroup>

```
<rich:toolBar height="26" itemSeparator="grid">
  <rich:toolBarGroup>
    <h:graphicImage value="/images/icons/create_doc.gif"/>
    <h:graphicImage value="/images/icons/create_folder.gif"/>
    <h:graphicImage value="/images/icons/copy.gif"/>
  </rich:toolBarGroup>
  <rich:toolBarGroup>
    <h:graphicImage value="/images/icons/save.gif"/>
    <h:graphicImage value="/images/icons/save_as.gif"/>
    <h:graphicImage value="/images/icons/save_all.gif"/>
  </rich:toolBarGroup>
  <rich:toolBarGroup location="right">
    <h:graphicImage value="/images/icons/find.gif"/>
    <h:graphicImage value="/images/icons/filter.gif"/>
  </rich:toolBarGroup>
</rich:toolBar>
```

The example shows how to locate a toolbar group to the right-hand side of the parent toolbar. It also demonstrates how item separators on the parent toolbar work with toolbar groups.



11.4.6.3. Reference data

- *component-type*: `org.richfaces.ToolbarGroup`
- *component-class*: `org.richfaces.component.html.HtmlToolbarGroup`
- *component-family*: `org.richfaces.ToolbarGroup`
- *renderer-type*: `org.richfaces.ToolbarGroupRenderer`
- *tag-class*: `org.richfaces.taglib.ToolbarGroupTag`

Output and messages

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

12.1. `<rich:message>`

The `<rich:message>` component renders a single message relating to another specific component. The message consists of two parts, both of which are optional: the marker icon and the textual label. The appearance of the message can be customized, and tool-tips can be used for further information about the message.

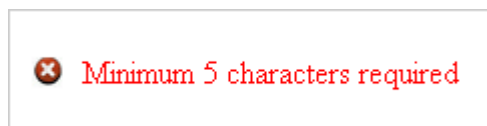


Figure 12.1. `rich:message` component

12.1.1. Basic usage

The `<rich:message>` component needs the `for` attribute to point to the `id` identifier of the related component. The message is displayed depending on the state of the linked component's requests.

The `<rich:message>` component is automatically rendered after an Ajax request, even without the use of an `<a4j:outputPanel>` component.

12.1.2. Appearance

The `passedLabel` attribute contains a message to display when there are no errors; that is, when validation passes successfully. The `showSummary` attribute specifies whether to display only a summary of the full message. The full message can be displayed in a tool-tip when hovering the mouse over the summary.

Facets are used to define the marker icons for different message states:

`errorMarker`

Defines the icon for messages with the `error` severity class.

`fatalMarker`

Defines the icon for messages with the `fatal` severity class.

`infoMarker`

Defines the icon for messages with the `info` severity class.

`passedMarker`

Defines the icon for messages that are not of the `error`, `fatal`, `info`, or `warn` severity classes.

`warnMarker`

Defines the icon for messages with the `warn` severity class.

Example 12.1. rich:message example

```
<rich:message for="id" passedLabel="No errors" showSummary="true">
  <f:facet name="errorMarker">
    <h:graphicImage url="/image/error.png"/>
  </f:facet>
  <f:facet name="passedMarker">
    <h:graphicImage url="/image/passed.png"/>
  </f:facet>
</rich:message>
```

12.1.3. Reference data

- `component-type`: `org.richfaces.component.RichMessage`
- `component-class`: `org.richfaces.component.html.HtmlRichMessage`
- `component-family`: `org.richfaces.component.RichMessage`
- `renderer-type`: `org.richfaces.RichMessageRenderer`
- `tag-class`: `org.richfaces.taglib.RichMessageTag`

12.1.4. Style classes and skin parameters

Table 12.1. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-msg</code> This class defines styles for the message itself.	<code>generalFamilyFont</code>	font-family
	<code>generalSizeFont</code>	font-size
<code>.rf-msg-err</code> This class defines styles for an error message.	<code>errorColor</code>	color
<code>.rf-msg-ftl</code> This class defines styles for a fatal message.	<code>errorColor</code>	color

Class (selector)	Skin Parameters	Mapped CSS properties
.rf-msg-inf This class defines styles for an information message.	<i>generalTextColor</i>	color
.rf-msg-wrn This class defines styles for a warning message.	<i>warningTextColor</i>	color
.rf-msg-ok This class defines styles for a basic OK message.	<i>generalTextColor</i>	color
.rf-msg-sum, .rf-msg-det These classes define styles for the summary or details of a message.	No skin parameters.	

12.2. <rich:messages>

The <rich:messages> component works similarly to the <rich:message> component, but can display validation messages for a group of components instead of just a single component. Refer to [Section 12.1, “<rich:message>”](#) for details on the <rich:message> component.

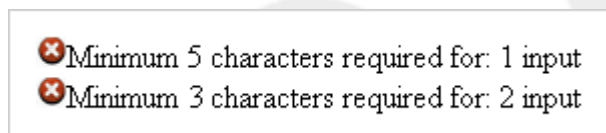


Figure 12.2. rich:messages component

12.2.1. Basic usage

The <rich:messages> component doesn't require any extra attributes for basic usage. It displays all messages relating to requests from components in the same container element.

The <rich:message> component is automatically rendered after an Ajax request, even without the use of an <a4j:outputPanel> component.

12.2.2. Appearance

The <rich:messages> component displays error messages for each validating component in the same container. The `passedLabel` attribute contains a message to display when there are no errors; that is, when validation for all components passes successfully. The `layout` attribute defines how the messages are presented: either using `list` for a list layout, or `table` for a tabular layout.

All messages use the same state icons, specified by using facets in the same way as for the `<rich:message>` component:

`errorMarker`

Defines the icon for messages with the `error` severity class.

`fatalMarker`

Defines the icon for messages with the `fatal` severity class.

`infoMarker`

Defines the icon for messages with the `info` severity class.

`passedMarker`

Defines the icon for messages that are not of the `error`, `fatal`, `info`, or `warn` severity classes.

`warnMarker`

Defines the icon for messages with the `warn` severity class.

Example 12.2. rich:messages example

```
<h:form>
  <rich:messages passedLabel="Data validated." layout="list">
    <f:facet name="header">
      <h:outputText value="Entered Data Status:"></h:outputText>
    </f:facet>
    <f:facet name="passedMarker">
      <h:graphicImage value="/images/ajax/passed.gif" />
    </f:facet>
    <f:facet name="errorMarker">
      <h:graphicImage value="/images/ajax/error.gif" />
    </f:facet>
  </rich:messages>

  <h:panelGrid columns="2">
    <h:outputText value="Name:" />

    <h:inputText label="Name" id="name" required="true" value="#{userBean.name}">
      <f:validateLength minimum="3" />
    </h:inputText>
    <h:outputText value="Job:" />
    <h:inputText label="Job" id="job" required="true" value="#{userBean.job}">
      <f:validateLength minimum="3" maximum="50" />
    </h:inputText>
    <h:outputText value="Address:" />

    <h:inputText label="Address" id="address" required="true" value="#{userBean.address}">
      <f:validateLength minimum="10" />
    </h:inputText>
    <h:outputText value="Zip:" />
```

```

<h:inputText label="Zip" id="zip" required="true" value="#{userBean.zip}">
    <f:validateLength minimum="4" maximum="9" />
</h:inputText>
<f:facet name="footer">
    <a4j:commandButton value="Validate" />
</f:facet>
</h:panelGrid>
</h:form>

```

12.2.3. Reference data

- *component-type*: org.richfaces.component.RichMessages
- *component-class*: org.richfaces.component.html.HtmlRichMessages
- *component-family*: org.richfaces.component.RichMessages
- *renderer-type*: org.richfaces.RichMessagesRenderer
- *tag-class*: org.richfaces.taglib.RichMessagesTag

12.2.4. Style classes and skin parameters

The `<rich:messages>` component uses the same styling as the `<rich:message>` component. Refer to [Section 12.1.4, “Style classes and skin parameters”](#) for details.

12.3. <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.

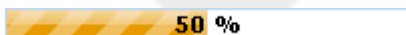


Figure 12.3. `<rich:progressBar>`

12.3.1. 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 12.3. Basic usage

```

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

```

12.3.2. 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 12.4. Using the `label` attribute

```
<rich:progressBar value="#{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 12.5. 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 parameters 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 12.6. 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 12.7, “Using the `param` parameter”](#).

Example 12.7. 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 12.8. 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>
```

12.3.3. 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 12.9. Using set intervals

```
<rich:progressBar value="#{bean.incValue}" progressInterval="200" enabled="#{bean.enabled1}"/>
```

12.3.4. 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.

12.3.5. JavaScript API

The `<rich:progressBar>` component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

getValue()

Return the current value displayed on the progress bar.

setValue()

Set the current value to display on the progress bar.

getMinValue()

Return the minimum value for the progress bar.

getMaxValue()

Return the maximum value for the progress bar.

disable()

Disables the progress bar.

enable()

Enables the progress bar.

isEnabled()

Returns a boolean value indicating whether the progress bar is enabled.

12.3.6. Reference data

- *component-type*: `org.richfaces.ProgressBar`

- *component-class*: `org.richfaces.component.html.HtmlProgressBar`
- *component-family*: `org.richfaces.component.ProgressBar`
- *renderer-type*: `org.richfaces.renderkit.ProgressBarRenderer`
- *tag-class*: `org.richfaces.taglib.ProgressBarTag`

12.3.7. Style classes and skin parameters

Table 12.2. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters	Mapped CSS properties
<code>.rf-pb-lbl</code> This class defines styles for labels on the progress bar.	No skin parameters.	
<code>.rf-pb-prgs</code> This class defines styles for the progressed portion of the progress bar.	<code>panelBorderColor</code>	<code>border-color</code>
	<code>selectControlColor</code>	<code>background-color</code>
<code>.rf-pb-init</code> , <code>.rf-pb-fin</code> These classes define styles for the initial state and finished state.	<code>generalTextColor</code>	<code>color</code>
	<code>generalFamilyFont</code>	<code>font-family</code>
	<code>generalSizeFont</code>	<code>font-size</code>

12.4. <rich:tooltip>

The `<rich:tooltip>` component provides an informational tool-tip. The tool-tip can be attached to any control and is displayed when hovering the mouse cursor over the control.



Figure 12.4. `<rich:tooltip>`

12.4.1. Basic usage

For basic usage, define the tool-tip text using the `value` attribute. The `<rich:tooltip>` component is then automatically attached to the parent element, and is usually shown when the mouse cursor hovers.

Alternatively, the content of the tool-tip can be defined inside the `<rich:tooltip>` tags, and the `value` attribute is not used. This allows HTML tags to be used to define the content, and provides for rich content such as images, links, buttons, and other RichFaces components.

Example 12.10. Defining tool-tip content

Basic content

```
<rich:panel>
  <rich:toolTip value="This is a tool-tip."/>
</rich:panel>
```

Rich content

```
<rich:panel>
  <rich:toolTip>
    This is a <b>tool-tip</b>.
  </rich:toolTip>
</rich:panel>
```

12.4.2. Attaching the tool-tip to another component

If not otherwise specified, the tool-tip is attached to the parent element in which it is defined. The `for` attribute is used to attach the tool-tip to another component, pointing to the target component's `id` identifier. This allows the `<rich:toolTip>` component to be specified outside the target element. This approach is demonstrated in [Example 12.11, “Attaching the tool-tip”](#).

Example 12.11. Attaching the tool-tip

```
<rich:panel id="panelId">
  ...
</rich:panel>

<rich:toolTip value="This is a tool-tip." for="panelId"/>
```

The `<rich:toolTip>` component can alternatively be left unattached, and is instead invoked through an event handler on the target component. To leave the `<rich:toolTip>` component unattached, set `attached="false"`, and define the event handler to trigger the tool-tip on the target component. This approach is demonstrated in [Example 12.12, “Unattached tool-tips”](#). When leaving the `<rich:toolTip>` component unattached, ensure it has an `id` identifier defined. If the `<rich:toolTip>` component is nested inside the target element, it must be the last child. If it is defined outside the target element, it must be nested in an `<a4j:form>` component.

Example 12.12. Unattached tool-tips

```
<rich:panel id="panelId" onclick="#{rich:component('tooltipId')}.show(event);" />

<a4j:form>
    <rich:toolTip id="toolTipId" attached="false" value="This is a tool-tip."/>
</a4j:form>
```

12.4.3. Appearance

By default, the `<rich:toolTip>` component is positioned intelligently based on the position of the mouse cursor. Use the `direction` attribute to specify a corner of the target component at which to display the tool-tip instead. Possible values include `top-left`, `top-right`, `bottom-left`, and `bottom-right`. Use the `horizontalOffset` and `verticalOffset` attributes to specify the horizontal offset and vertical offset at which to display the tool-tip.

Use the `hideEvent` attribute to specify when the tool-tip is hidden. The default value is `none`, so the tool-tip remains shown. However, it can be linked to an event on the target component, such as the `mouseout` event.

Set `followMouse="true"` to cause the tool-tip to follow the user's mouse movements.

Advanced appearance features are demonstrated in [Example 12.13, "Advanced tool-tip usage"](#).

12.4.4. Update mode

The mode for updating the tool-tip is determined by the `mode` attribute, which can have one of the following values:

ajax

The tool-tip content is requested from the server with every activation.

client

The tool-tip content is rendered once on the server. An external submit causes the content to re-render.

When using `mode="ajax"`, define the `defaultContent` facet. The tool-tip displays the content of the `defaultContent` facet while loading the actual content from the server.

Example 12.13. Advanced tool-tip usage

```
<h:commandLink value="Simple Link" id="link">
    <rich:toolTip followMouse="true" direction="top-
right" mode="ajax" value="#{bean.toolTipContent}"
horizontalOffset="5" verticalOffset="5" layout="block">
```

```
<f:facet name="defaultContent">
    <f:verbatim>Loading...</f:verbatim>
</f:facet>
</rich:tooltip>
</h:commandLink>
```

12.4.5. <rich:tooltip> client-side events

The <rich:tooltip> component supports the following client-side events:

click

This event is activated when the tool-tip is clicked with the mouse.

dblclick

This event is activated when the tool-tip is double-clicked with the mouse.

mouseout

This event is activated when the mouse cursor leaves the tool-tip.

mousemove

This event is activated when the mouse cursor moves over the tool-tip.

mouseover

This event is activated when the mouse cursor hovers over the tool-tip.

show

This event is activated when the tool-tip is shown.

complete

This event is activated when the tool-tip is completed.

hide

This event is activated when the tool-tip is hidden.

12.4.6. JavaScript API

The <rich:tooltip> component can be controlled through the JavaScript API. The JavaScript API provides the following functions:

show(event)

Show the tool-tip.

hide(event)

Hide the tool-tip.

12.4.7. Reference data

- *component-type*: org.richfaces.component.tooltip

- *component-class*: `org.richfaces.component.html.HtmlTooltip`
- *component-family*: `org.richfaces.component.tooltip`
- *renderer-type*: `org.richfaces.renderkit.html.tooltipRenderer`
- *tag-class*: `org.richfaces.taglib.HtmlTooltipTag`

12.4.8. Style classes and skin parameters

Table 12.3. Style classes (selectors) and corresponding skin parameters

Class (selector)	Skin Parameters		Mapped CSS properties
<code>.rf-tt</code> This class defines styles for the tool-tip itself.	No skin parameters.		
<code>.rf-tt-loading</code> This class defines styles for the tool-tip when it is loading.	No skin parameters.		
<code>.rf-tt-cnt</code> This class defines styles for the tool-tip content.	No skin parameters.		
<code>.rf-tt-cntr</code> This class defines styles for the progressed portion of the progress bar.	<code>tipBorderColor</code>		<code>border-color</code>
	<code>generalFamilyFont</code>		<code>font-family</code>
	<code>generalSizeFont</code>		<code>font-size</code>

Drag and drop

Read this chapter for details on adding drag and drop support to controls.

13.1. `<rich:dragSource>`

The `<rich:dragSource>` component can be added to a component to indicate it is capable of being dragged by the user. The dragged item can then be dropped into a compatible drop area, designated using the `<rich:dropTarget>` component.

13.1.1. Basic usage

To add drag support to a component, attach the `<rich:dragSource>` component as a child element.

The `type` attribute must be specified, and can be any identifying string. Dragged items can only be dropped in drop zones where the `type` attribute of the `<rich:dragSource>` component is listed in the `acceptedTypes` attribute of the `<rich:dropTarget>` component.

13.1.2. Dragging an object

Use the `dragIndicator` parameter to customize the appearance of a dragged object while it is being dragged. The `dragIndicator` parameter must point to the `id` identifier of a `<rich:dragIndicator>` component. If the `dragIndicator` attribute is not defined, the drag indicator appears as a clone of the `<rich:dragSource>` component's parent control.

To bind data to the dragged object, use the `dragValue` attribute. The `dragValue` attribute specifies an item in a data model, which is then bound to the parent component when it is dragged. This facilitates handling event data during a drop event.

13.1.3. Reference data

- `component-type`: `org.richfaces.DragSource`
- `component-class`: `org.richfaces.component.html.HtmlDragSource`
- `component-family`: `org.richfaces.DragSource`
- `renderer-type`: `org.richfaces.DragSourceRenderer`
- `tag-class`: `org.richfaces.taglib.DragSourceTag`

13.2. `<rich:dropTarget>`

The `<rich:dropTarget>` component can be added to a component so that the component can accept dragged items. The dragged items must support the `<rich:dragSource>` component, and be of a compatible drop type.

13.2.1. Basic usage

To allow dragged items to be dropped on a component, attach the `<rich:dropTarget>` component as a child element to the component.

The `acceptedTypes` attribute must be specified. The `acceptedTypes` attribute is a comma-separated list of strings that match the types of dragged items. Dragged items can only be dropped in drop zones where the `type` attribute of the `<rich:dragSource>` component is listed in the `acceptedTypes` attribute of the `<rich:dropTarget>` component.

The `acceptedTypes` attribute can optionally be set to either `@none` or `@all`. If set to `@none`, the component will not accept any type of dropped object. If set to `@all`, the component accepts all dropped objects. If the `acceptedTypes` attribute is not specified, the default value is `null`, which is the same as a `@none` setting.

13.2.2. Handling dropped data

To provide additional parameters for a drop event, use the `dropValue` attribute.

The `<rich:dropTarget>` component raises the `DropEvent` server-side event when an object is dropped. The event uses the following parameters:

- The `dragSource` identifies the component being dragged (the parent of the `<rich:dragSource>` component).
- The `dragValue` parameter is the content of the `<rich:dragSource>` component's `dragValue` attribute.
- The `dropValue` parameter is the content of the `<rich:dropTarget>` component's `dropValue` attribute.

13.2.3. Reference data

- `component-type`: `org.richfaces.DropTarget`
- `component-class`: `org.richfaces.component.html.HtmlDropTarget`
- `component-family`: `org.richfaces.DropTarget`
- `renderer-type`: `org.richfaces.DropTargetRenderer`
- `tag-class`: `org.richfaces.taglib.DropTargetTag`

13.2.4. Style classes

Style classes (selectors)

`.rf-drp-hvr`

This class defines styles for the drop target when a dragged item is hovering over it.

.rf-drp-hlight

This class defines styles for a highlighted drop target.

13.3. <rich:dragIndicator>

The <rich:dragIndicator> component defines a graphical element to display under the mouse cursor during a drag-and-drop operation.

13.3.1. Basic usage

If included without any attributes specified, the drag indicator will appear as an empty dotted outline, as shown in



Figure 13.1. An empty drag indicator

13.3.2. Styling the indicator

The drag indicator can be styled depending on the current state of the dragged element. There are three attributes for different states. The attributes reference the CSS class to use for styling the drag indicator when the dragged element is in the relevant state.

`acceptClass`

The `acceptClass` attribute specifies the style when the dragged element is over an acceptable drop target. It indicates that the `type` attribute of the element's <rich:dragSource> component matches `acceptedTypes` attribute of the drop target's <rich:dropTarget> component.

`rejectClass`

The `rejectClass` attribute specifies the style when the dragged element is over a drop target that is not acceptable. It indicates that the `type` attribute of the element's <rich:dragSource> component is not found in the `acceptedTypes` attribute of the drop target's <rich:dropTarget> component.

`draggingClass`

The `draggingClass` attribute specifies the style when the dragged element is being dragged. It indicates that the dragged element is not over a drop target.

13.3.3. Reference data

- `component-type: org.richfaces.DragIndicator`

- *component-class*: `org.richfaces.component.html.HtmlDragIndicator`
- *component-family*: `org.richfaces.DragIndicator`
- *renderer-type*: `org.richfaces.DragIndicatorRenderer`
- *tag-class*: `org.richfaces.taglib.DragIndicatorTag`

13.3.4. Style classes

Style classes (selectors)

`.rf-ind`

This class defines styles for the drag indicator.

`.rf-ind-drag`

This class defines styles for the indicator when it is being dragged.

`.rf-ind-acpt`

This class defines styles for the indicator when it is over an acceptable drop target.

`.rf-ind-rejt`

This class defines styles for the indicator when it is over an unacceptable drop target.

Layout and appearance

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

14.1. <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.

14.1.1. 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.

14.1.2. 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 14.1, “Avoiding syntax confusion”* demonstrates how to use double backslashes to escape colon characters in `id` identifier values.

Example 14.1. 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="..." />
```

14.1.3. 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.

14.1.4. 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:

`domready`

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 14.2. `<rich:jQuery>` example

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

```
<rich:jQuery selector="#customList"
  tr:odd timing="domready" 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 (`load`) 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

14.1.5. 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 14.3, "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 14.3. Calling a `<rich:jQuery>` component as a function

```
<h:graphicImage width="50" value="/images/price.png"
  onmouseover="enlargePic(this,{pwidth:'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.

14.1.6. 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 `domready` 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 `complete` event attribute of the component that triggered the Ajax interaction.

14.1.7. 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 2.4.4.1](#), “*data*” for details on the `data` attribute.

15.1. <rich:componentControl>

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

15.1.1. Basic usage

The `operation` attribute is required to attach JavaScript functions to the parent component, along with either the `target` or `selector` attributes. Use the `operation` attribute to specify the JavaScript API function to perform. Use the `target` attribute to define the `id` identifier of the target component, or use the `selector` attribute to define a number of target components through the use of valid jQuery selectors.

Use the `event` attribute to specify the event that triggers the JavaScript API function call if it is different from the default triggering event for the parent component.

Example 15.1. <rich:componentControl> basic usage

```
<h:commandButton value="Show Modal Panel">
  <!--componentControl is attached to the commandButton-->

  <rich:componentControl target="ccModalPanelID" event="click" operation="show" /
>
</h:commandButton>
```

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

15.1.2. Passing parameters to API methods

The `operation` can receive parameters through nested `<f:param>` elements.

Example 15.2. Using parameters

```
<rich:componentControl event="click" target="modalPanel" operation="show">
```

```
<f:param value="width" name="500"/>
</rich:componentControl>
```

15.1.3. 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

15.2. 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.

15.3. 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.

15.4. 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.

15.5. 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 15.3. rich:findComponent example

```
<h:inputText id="myInput">
  <a4j:support event="keyup" render="outtext"/>
</h:inputText>
```



```
<h:outputText id="outtext" value="#{rich:findComponent('myInput').value}" />
```

15.6. 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 15.4. 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')}" />
```

