



# PRIMEFACES

## USER'S GUIDE

Author

Optimus Prime

Covers 2.2 and 2.2.1  
Last Update: 05.02.2011

*This guide is dedicated to my wife Nurcan,  
without her support PrimeFaces wouldn't exist.*

*Çağatay Çivici*

<b>1. Introduction</b>	<b>10</b>
<b>1.1 What is PrimeFaces?</b>	<b>10</b>
<b>1.2 Prime Technology</b>	<b>10</b>
<b>2. Setup</b>	<b>11</b>
<b>2.1 Download</b>	<b>11</b>
<b>2.2 Dependencies</b>	<b>12</b>
<b>2.3 Configuration</b>	<b>12</b>
<b>2.4 Hello World</b>	<b>12</b>
<b>3. Component Suite</b>	<b>13</b>
<b>3.1 AccordionPanel</b>	<b>13</b>
<b>3.2 AjaxBehavior</b>	<b>18</b>
<b>3.3 AjaxStatus</b>	<b>21</b>
<b>3.4 AutoComplete</b>	<b>24</b>
<b>3.5 BreadCrumb</b>	<b>31</b>
<b>3.6 Button</b>	<b>34</b>
<b>3.7 Calendar</b>	<b>37</b>
<b>3.8 Captcha</b>	<b>47</b>
<b>3.9 Carousel</b>	<b>50</b>
<b>3.10 CellEditor</b>	<b>56</b>
<b>3.11 Charts</b>	<b>57</b>
<b><i>3.11.1 Pie Chart</i></b>	<b>57</b>
<b><i>3.11.2 Line Chart</i></b>	<b>61</b>
<b><i>3.11.3 Column Chart</i></b>	<b>65</b>
<b><i>3.11.4 Stacked Column Chart</i></b>	<b>67</b>
<b><i>3.11.5 Bar Chart</i></b>	<b>69</b>

<b>3.11.6 StackedBar Chart</b>	<b>71</b>
<b>3.11.7 Chart Series</b>	<b>73</b>
<b>3.11.8 Skinning Charts</b>	<b>74</b>
<b>3.11.9 Real-Time Charts</b>	<b>77</b>
<b>3.11.10 Interactive Charts</b>	<b>79</b>
<b>3.11.11 Charting Tips</b>	<b>80</b>
<b>3.12 Collector</b>	<b>81</b>
<b>3.13 Color Picker</b>	<b>83</b>
<b>3.14 Column</b>	<b>87</b>
<b>3.15 Columns</b>	<b>89</b>
<b>3.16 ColumnGroup</b>	<b>90</b>
<b>3.17 CommandButton</b>	<b>91</b>
<b>3.18 CommandLink</b>	<b>96</b>
<b>3.19 ConfirmDialog</b>	<b>99</b>
<b>3.20 ContextMenu</b>	<b>103</b>
<b>3.21 Dashboard</b>	<b>106</b>
<b>3.22 DataExporter</b>	<b>111</b>
<b>3.23 DataGrid</b>	<b>114</b>
<b>3.24 DataList</b>	<b>120</b>
<b>3.25 DataTable</b>	<b>125</b>
<b>3.26 Dialog</b>	<b>143</b>
<b>3.27 Divider</b>	<b>148</b>
<b>3.28 Drag&amp;Drop</b>	<b>150</b>
<b>  3.28.1 Draggable</b>	<b>150</b>
<b>  3.28.2 Droppable</b>	<b>154</b>
<b>3.29 Dock</b>	<b>159</b>

<b>3.30 Editor</b>	<b>161</b>
<b>3.31 Effect</b>	<b>165</b>
<b>3.32 Fieldset</b>	<b>168</b>
<b>3.33 FileDownload</b>	<b>172</b>
<b>3.34 FileUpload</b>	<b>174</b>
<b>3.35 Focus</b>	<b>179</b>
<b>3.36 Galleria</b>	<b>181</b>
<b>3.37 GMap</b>	<b>184</b>
<b>3.38 GMapInfoWindow</b>	<b>197</b>
<b>3.39 GraphicImage</b>	<b>198</b>
<b>3.40 GraphicText</b>	<b>203</b>
<b>3.41 Growl</b>	<b>205</b>
<b>3.42 HotKey</b>	<b>208</b>
<b>3.43 IdleMonitor</b>	<b>211</b>
<b>3.44 ImageCompare</b>	<b>214</b>
<b>3.45 ImageCropper</b>	<b>216</b>
<b>3.46 ImageSwitch</b>	<b>220</b>
<b>3.47 Inplace</b>	<b>223</b>
<b>3.48 InputMask</b>	<b>227</b>
<b>3.49 InputText</b>	<b>231</b>
<b>3.50 InputTextarea</b>	<b>234</b>
<b>3.51 Keyboard</b>	<b>238</b>
<b>3.52 Layout</b>	<b>243</b>
<b>3.53 LayoutUnit</b>	<b>250</b>
<b>3.54 LightBox</b>	<b>252</b>

<b>3.55 Media</b>	<b>257</b>
<b>3.56 Menu</b>	<b>259</b>
<b>3.57 Menubar</b>	<b>265</b>
<b>3.58 MenuButton</b>	<b>268</b>
<b>3.59 MenuItem</b>	<b>270</b>
<b>3.60 Message</b>	<b>273</b>
<b>3.61 Messages</b>	<b>275</b>
<b>3.62 NotificationBar</b>	<b>277</b>
<b>3.63 OutputPanel</b>	<b>280</b>
<b>3.64 Panel</b>	<b>282</b>
<b>3.65 Password</b>	<b>287</b>
<b>3.66 PickList</b>	<b>292</b>
<b>3.67 Poll</b>	<b>298</b>
<b>3.68 Printer</b>	<b>301</b>
<b>3.69 ProgressBar</b>	<b>302</b>
<b>3.70 Push</b>	<b>306</b>
<b>3.71 Rating</b>	<b>307</b>
<b>3.72 RemoteCommand</b>	<b>311</b>
<b>3.73 Resizable</b>	<b>313</b>
<b>3.74 Resource</b>	<b>317</b>
<b>3.75 Resources</b>	<b>318</b>
<b>3.76 Row</b>	<b>319</b>
<b>3.77 RowEditor</b>	<b>320</b>
<b>3.78 RowExpansion</b>	<b>321</b>
<b>3.79 RowToggler</b>	<b>322</b>

<b>3.80 Schedule</b>	<b>323</b>
<b>3.81 Separator</b>	<b>335</b>
<b>3.82 Slider</b>	<b>337</b>
<b>3.83 Spacer</b>	<b>342</b>
<b>3.84 Spinner</b>	<b>343</b>
<b>3.85 Submenu</b>	<b>348</b>
<b>3.86 Stack</b>	<b>349</b>
<b>3.87 Tab</b>	<b>351</b>
<b>3.88 TabView</b>	<b>352</b>
<b>3.89 Terminal</b>	<b>357</b>
<b>3.90 ThemeSwitcher</b>	<b>359</b>
<b>3.91 Toolbar</b>	<b>361</b>
<b>3.92 ToolbarGroup</b>	<b>363</b>
<b>3.93 Tooltip</b>	<b>364</b>
<b>3.94 Tree</b>	<b>367</b>
<b>3.95 TreeNode</b>	<b>378</b>
<b>3.96 TreeTable</b>	<b>379</b>
<b>3.97 Watermark</b>	<b>384</b>
<b>3.98 Wizard</b>	<b>386</b>
<b>4. TouchFaces</b>	<b>392</b>
<b>4.1 Getting Started with TouchFaces</b>	<b>392</b>
<b>4.2 Views</b>	<b>394</b>
<b>4.3 Navigations</b>	<b>396</b>
<b>4.4 Ajax Integration</b>	<b>398</b>
<b>4.5 Sample Applications</b>	<b>399</b>

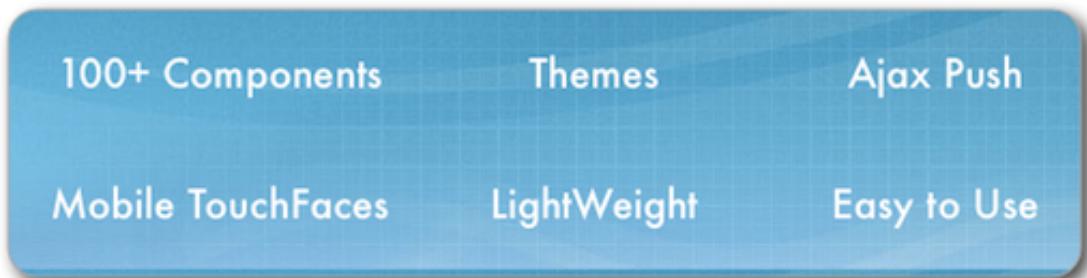
<b>4.6 TouchFaces Components</b>	<b>400</b>
<b>4.6.1 Application</b>	<b>400</b>
<b>4.6.2 NavBarControl</b>	<b>401</b>
<b>4.6.3 RowGroup</b>	<b>402</b>
<b>4.6.4 RowItem</b>	<b>403</b>
<b>4.6.5 Switch</b>	<b>404</b>
<b>4.6.6 TableView</b>	<b>406</b>
<b>4.6.7 View</b>	<b>407</b>
<b>5. Partial Rendering and Processing</b>	<b>408</b>
<b>5.1 Partial Rendering</b>	<b>408</b>
<b>5.1.1 Infrastructure</b>	<b>408</b>
<b>5.1.2 Using IDs</b>	<b>408</b>
<b>5.1.3 Notifying Users</b>	<b>411</b>
<b>5.1.4 Bits&amp;Pieces</b>	<b>411</b>
<b>5.2 Partial Processing</b>	<b>412</b>
<b>5.2.1 Partial Validation</b>	<b>412</b>
<b>5.2.2 Keywords</b>	<b>413</b>
<b>5.2.3 Using Ids</b>	<b>413</b>
<b>6. Ajax Push/Comet</b>	<b>414</b>
<b>6.1 Atmosphere</b>	<b>414</b>
<b>6.2 PrimeFaces Push</b>	<b>415</b>
<b>6.2.1 Setup</b>	<b>415</b>
<b>6.2.2. CometContext</b>	<b>415</b>
<b>6.2.3 Push Component</b>	<b>416</b>
<b>7. Javascript API</b>	<b>418</b>
<b>7.1 PrimeFaces Namespace</b>	<b>418</b>

<b>7.2 Ajax API</b>	<b>419</b>
<b>8. Themes</b>	<b>422</b>
<b>8.1 Applying a Theme</b>	<b>423</b>
<b>8.2 Creating a New Theme</b>	<b>424</b>
<b>8.3 How Themes Work</b>	<b>426</b>
<b>8.4 Theming Tips</b>	<b>427</b>
<b>9. Utilities</b>	<b>428</b>
<b>9.1 RequestContext</b>	<b>428</b>
<b>9.2 EL Functions</b>	<b>431</b>
<b>10. Portlets</b>	<b>433</b>
<b>10.1 Dependencies</b>	<b>433</b>
<b>10.2 Configuration</b>	<b>434</b>
<b>11. Integration with Java EE</b>	<b>437</b>
<b>12. IDE Support</b>	<b>438</b>
<b>12.1 NetBeans</b>	<b>438</b>
<b>12.2 Eclipse</b>	<b>439</b>
<b>13. Project Resources</b>	<b>441</b>
<b>14. FAQ</b>	<b>442</b>

# 1. Introduction

## 1.1 What is PrimeFaces?

PrimeFaces is an open source component suite for Java Server Faces featuring 100+ Ajax powered rich set of JSF components. Additional TouchFaces module features a UI kit for developing mobile web applications. Main goal of PrimeFaces is to create the ultimate component suite for JSF.



- 100+ rich set of components (HtmlEditor, Dialog, AutoComplete, Charts and more).
- Built-in Ajax with Lightweight Partial Page Rendering.
- Native Ajax Push/Comet support.
- Mobile UI kit to create mobile web applications for handheld devices with webkit based browsers.(IPhone, Palm, Android Phones, Nokia S60 and more)
- One jar, zero-configuration and no required dependencies.
- Skinning Framework with 30 pre-designed themes.
- Extensive documentation.

## 1.2 Prime Technology

PrimeFaces is maintained by Prime Technology, a Turkish software development company specialized in Agile and Java EE consulting. Project is led by Çağatay Çivici (aka Optimus Prime), a JSF Expert Group Member.

# 2. Setup

## 2.1 Download

PrimeFaces has a single jar called **primefaces-{version}.jar**. There are two ways to download this jar, you can either download from PrimeFaces homepage or if you are a maven user you can define it as a dependency.

### Download Manually

Three different artifacts are available for each PrimeFaces version, binary, sources and bundle. Bundle contains binary, sources and javadocs.

```
http://www.primefaces.org/downloads.html
```

### Download with Maven

Group id of the dependency is *org.primefaces* and artifact id is *primefaces*.

```
<dependency>
    <groupId>org.primefaces</groupId>
    <artifactId>primefaces</artifactId>
    <version>2.2</version>
</dependency>
```

In addition to the configuration above you also need to add Prime Technology maven repository to the repository list so that maven can download it.

```
<repository>
    <id>prime-repo</id>
    <name>Prime Technology Maven Repository</name>
    <url>http://repository.prime.com.tr</url>
    <layout>default</layout>
</repository>
```

## 2.2 Dependencies

PrimeFaces only requires a JAVA 5+ runtime and a JSF 2.0 implementation as mandatory dependencies. There're some optional libraries for certain features.

Dependency	Version *	Type	Description
JSF runtime	2.0+	Required	Apache MyFaces or Oracle Mojarra
itext	2.1.7	Optional	PDF export support for DataExporter component
apache poi	3.2-FINAL	Optional	Excel export support for DataExporter component
commons-fileupload	1.2.1	Optional	FileUpload
commons-io	1.4	Optional	FileUpload
atmosphere-runtime	0.5.1	Optional	Ajax Push
atmosphere-compat	0.5.1	Optional	Ajax Push

\* Listed versions are tested and known to be working with PrimeFaces, other versions of these dependencies may also work but not tested.

## 2.3 Configuration

PrimeFaces does not require any mandatory configuration.

## 2.4 Hello World

Once you have added the downloaded jar to your classpath, you need to add the PrimeFaces namespace to your page to begin using the components. Here is a simple page;

```
<html xmlns="http://www.w3c.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:p="http://primefaces.prime.com.tr/ui">

<h:head>
</h:head>

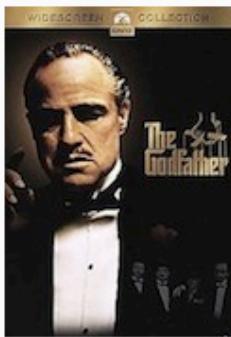
<h:body>
    <p:spinner />
</h:body>
</html>
```

# 3. Component Suite

## 3.1 AccordionPanel

AccordionPanel is a container component that displays content in stacked format.

**▼ Godfather Part I**



The story begins as Don Vito Corleone, the head of a New York Mafia family, oversees his daughter's wedding. His beloved son Michael has just come home from the war, but does not intend to become part of his father's business. Through Michael's life the nature of the family business becomes clear. The business of the family is just like the head of the family, kind and benevolent to those who give respect, but given to ruthless violence whenever anything stands against the good of the family.

**▶ Godfather Part II**

**▶ Godfather Part III**

### Info

Tag	<b>accordionPanel</b>
Component Class	<b>org.primefaces.component.accordionpanel.Accordionpanel</b>
Component Type	<b>org.primefaces.component.AccordionPanel</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.AccordionPanelRenderer</b>
Renderer Class	<b>org.primefaces.component.accordionpanel.AccordionPanelRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	boolean	Boolean value to specify the rendering of the component.
binding	null	Object	An EL expression that maps to a server side UIComponent instance in a backing bean.

Name	Default	Type	Description
activeIndex	0	Integer	Index of the active tab.
style	null	String	Inline style of the container element.
styleClass	null	String	Style class of the container element.
disabled	FALSE	Boolean	Disables or enables the accordion panel.
effect	slide	String	Effect to use when toggling the tabs.
autoHeight	TRUE	Boolean	When enabled, tab with highest content is used to calculate the height.
collapsible	FALSE	Boolean	Defines if accordion panel can be collapsed all together.
fillSpace	FALSE	Boolean	When enabled, accordion panel fills the height of it's parent container.
event	click	String	Client side event to toggle the tabs.
widgetVar	null	String	Name of the widget to access client side api.
tabChangeListener	null	MethodExpr	Server side listener to invoke when active tab changes
onTabChangeUpdate	null	String	Component(s) to update with ajax after dynamic tab change.
onTabChange	null	String	Client side callback to invoke when active tab changes.
dynamic	FALSE	Boolean	Defines the toggle mode.
cache	FALSE	Boolean	Defines if activating a dynamic tab should load the contents from server again.

## Getting Started with Accordion Panel

Accordion panel consists of one or more tabs and each tab can group any other jsf components.

```
<p:accordionPanel>
    <p:tab title="First Tab Title">
        <h:outputText value= "Lorem"/>
        ...More content for first tab
    </p:tab>
    <p:tab title="Second Tab Title">
        <h:outputText value="Ipsum" />
    </p:tab>
    //any number of tabs
</p:accordionPanel>
```

## Toggle Event

By default toggling happens when a tab header is clicked, you can also specify a custom event. For example below, toggling happens when mouse is over the tab headers.

```
<p:accordionPanel effect="hover">
    //..tabs
</p:accordionPanel>
```

## Dynamic Content Loading

AccordionPanel supports lazy loading of tab content, when dynamic option is set true, only active tab contents will be rendered to the client side and clicking an inactive tab header will do an ajax request to load the tab contents. This feature is useful to reduce bandwidth and speed up page loading time. By default activating a previously loaded dynamic tab does not initiate a request to load the contents again as tab is cached. To control this behavior use *cache* option.

```
<p:accordionPanel dynamic="true">
    //..tabs
</p:accordionPanel>
```

## onTabChange

You can use client/server side callbacks to get notified when active tab changes. On client side use *onTabChange* option.

```
<p:accordionPanel onTabChange="handleChange(event, ui)">
    //..tabs
</p:accordionPanel>

<script type="text/javascript">
    function handleChange(event, ui) {
        //Execute custom logic
    }
</script>
```

*ui* object will be passed to your callback containing information about the tab change event.

- *ui.newHeader* = jQuery object representing the header of new tab
- *ui.oldHeader* = jQuery object representing the header of previous tab
- *ui.newContent* = jQuery object representing the content of new tab
- *ui.oldContent* = jQuery object representing the content of previous tab

## TabChangeListener

onTabChange is used on the client side, in case you need to execute logic on server side, use *tabChangeListener* option.

```
<p:accordionPanel tabChangeListener="#{bean.onChange}">
    //..tabs
</p:accordionPanel>
```

```
public void onChange(TabChangeEvent event) {
    //Tab activeTab = event.getTab();
    //...
}
```

Your listener will be invoked with an *org.primefaces.event.TabChangeEvent* instance that contains a reference to the new active tab and the accordion panel itself.

If you'd like to update some parts of your page after your tabChangeListener is invoked, use *onTabChangeUpdate* option. Following example, adds a FacesMessage at listener and displays it using a growl component.

```
<p:growl id="messages" />

<p:accordionPanel tabChangeListener="#{bean.onChange}"
    onTabChangeUpdate="messages">
    //..tabs
</p:accordionPanel>
```

```
public void onChange(TabChangeEvent event) {
    FacesMessage msg = new FacesMessage("Tab Changed",
        "Active Tab:" + event.getTab().getId());
    FacesContext.getCurrentInstance().addMessage(null, msg);
}
```

## Note

For both dynamic loading and tabChangeListener features to work, at least one form needs to present on page, location of the form does not matter.

## Client Side API

Widget: `PrimeFaces.widget.AccordionPanel`

Method	Params	Return Type	Description
<code>select(index)</code>	index: Index of tab to display	void	Activates tab with given index
<code>collapseAll()</code>	-	void	Collapses all tabs.

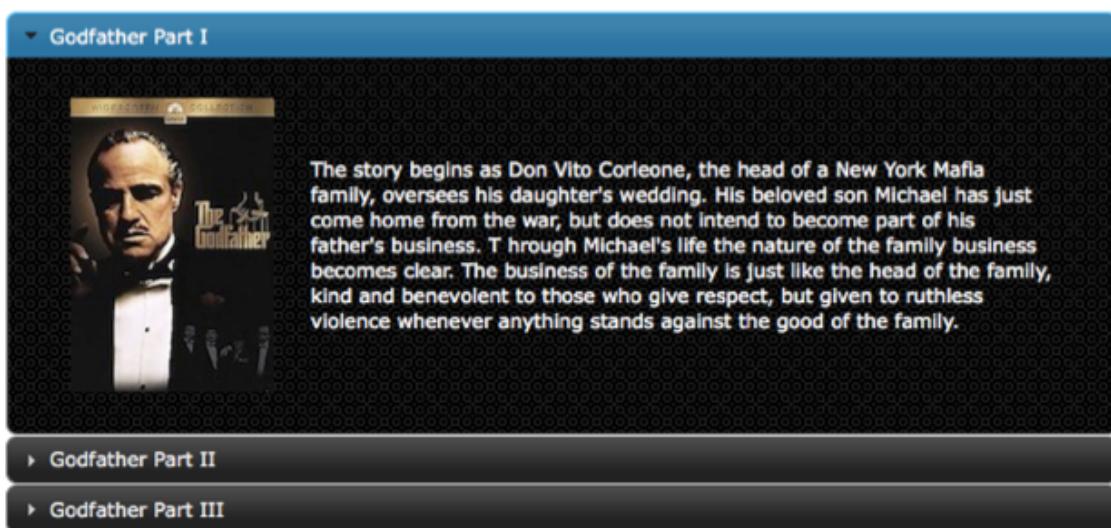
## Skinning

AccordionPanel resides in a main container element which `style` and `styleClass` options apply.

Following is the list of structural style classes;

Class	Applies
<code>.ui-accordion</code>	Main container element
<code>.ui-accordion-header</code>	Tab header
<code>.ui-accordion-content</code>	Tab content
<code>.ui-accordion-content-active</code>	Content of active tab.

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## Tips

- `autoHeight` option provides more consistent animations when enabled.
- Use `c:forEach` to create tabs on the fly, `ui:repeat` will not work as `p:tab` has no Renderer.

## 3.2 AjaxBehavior

AjaxBehavior is applied on components that support client behaviors similar to standard f:ajax behavior.

### Info

Tag	<b>ajax</b>
Behavior Id	<b>org.primefaces.component.AjaxBehavior</b>
Behavior Class	<b>org.primefaces.component.behavior.ajax.AjaxBehavior</b>

### Attributes

Name	Default	Type	Description
listener	null	MethodExpr	Method to process in partial request.
immediate	FALSE	boolean	Boolean value that determines the phaseId, when true actions are processed at apply_request_values, when false at invoke_application phase.
async	FALSE	Boolean	When set to true, ajax requests are not queued.
process	null	String	Component(s) to process in partial request.
update	null	String	Component(s) to update with ajax.
onstart	null	String	Javascript handler to execute before ajax request begins.
oncomplete	null	String	Javascript handler to execute when ajax request is completed.
onsuccess	null	String	Javascript handler to execute when ajax request succeeds.
onerror	null	String	Javascript handler to execute when ajax request fails.
global	TRUE	Boolean	Global ajax requests are listened by ajaxStatus component, setting global to false will not trigger ajaxStatus.
disabled	FALSE	Boolean	Disables ajax behavior.
event	null	String	Client side event to trigger ajax request.

### Getting Started with AjaxBehavior

AjaxBehavior is nested inside the target component;

```
<h:inputText value="#{bean.text}">
    <p:ajax update="out" />
</h:inputText>

<h:outputText id="out" value="#{bean.text}" />
```

In this example, each time the input changes, an ajax request is sent to the server. When the response is received uiajax partially updates the output text with id "out".

## Listener

In case you need to execute a method on a backing bean, define a listener;

```
<h:inputText id="counter">
    <p:ajax update="out" listener="#{counterBean.increment}" />
</h:inputText>

<h:outputText id="out" value="#{counterBean.count}" />
```

```
public class CounterBean {

    private int count;

    public int getCount() {
        return count;
    }

    public void setCount(int count) {
        this.count = count;
    }

    public void increment() {
        count++;
    }
}
```

## Events

Default client side events are defined by components that support client behaviors, for input components it is *onchange* and for command components it is *onclick*. In order to override the dom event to trigger the ajax request use *event* option. In following example, ajax request is triggered when key is up on input field.

```
<h:inputText id="firstname" value="#{bean.text}">
    <p:ajax update="out" event="keyup"/>
</h:inputText>

<h:outputText id="out" value="#{bean.text}" />
```

## Partial Processing

Partial processing is used with *process* option which defaults to @this, meaning the ajaxified component, following example processes all form and validation will fail as there is required field.

```
<h:form>

    <h:inputText required="true" />

    <h:inputText id="firstname" value="#{bean.text}">
        <p:ajax update="out" event="keyup"/>
    </h:inputText>

    <h:outputText id="out" value="#{bean.text}" />

</h:form>
```

## 3.3 AjaxStatus

AjaxStatus is a global notifier for ajax requests made by PrimeFaces components.



### Info

Tag	<b>ajaxStatus</b>
Component Class	<b>org.primefaces.component.ajaxstatus.AjaxStatus</b>
Component Type	<b>org.primefaces.component.AjaxStatus</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.AjaxStatusRenderer</b>
Renderer Class	<b>org.primefaces.component.ajaxstatus.AjaxStatusRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
onstart	null	String	Client side callback to execute after ajax requests start.
oncomplete	null	String	Client side callback to execute after ajax requests complete.
onprestart	null	String	Client side callback to execute before ajax requests start.
onsuccess	null	String	Client side callback to execute after ajax requests completed successfully.
onerror	null	String	Client side callback to execute when an ajax request fails.
style	null	String	Inline style of the container element
styleClass	null	String	Style class of the container element
widgetVar	null	String	Name of the widget to access client side api.

## Getting Started with AjaxStatus

AjaxStatus uses facets to represent the request status. Most common used facets are *start* and *complete*. Start facet will be visible once ajax request begins and stay visible until it's completed. Once the ajax response is received start facet becomes hidden and complete facet shows up.

```
<p:ajaxStatus>
    <f:facet name="start">
        <p:graphicImage value="ajaxloading.gif" />
    </f:facet>

    <f:facet name="complete">
        <h:outputText value="Done!" />
    </f:facet>
</p:ajaxStatus>
```

## Events

Here is the full list of available event names;

- default: Initially visible when page is loaded
- prestart: Before ajax request is sent
- start: After ajax request begins
- success: When ajax response is received without error
- error: When ajax response is received with error
- complete: When everything finishes.

```
<p:ajaxStatus>
    <f:facet name="prestart">
        <h:outputText value="Starting..." />
    </f:facet>

    <f:facet name="error">
        <h:outputText value="Error" />
    </f:facet>

    <f:facet name="success">
        <h:outputText value="Success" />
    </f:facet>

    <f:facet name="default">
        <h:outputText value="Idle" />
    </f:facet>

    <f:facet name="start">
        <h:outputText value="Please Wait" />
    </f:facet>

    <f:facet name="complete">
        <h:outputText value="Done" />
    </f:facet>
</p:ajaxStatus>
```

## Custom Events

Facets are the declarative way to use if you don't like javascript but if you like javascript, you can take advantage of on\* callbacks which are the event handler counterparts of the facets.

```
<p:ajaxStatus onstart="alert('Start')" oncomplete="alert('End')"/>
```

A common usage of programmatic approach is to implement a custom status dialog;

```
<p:ajaxStatus onstart="status.show()" oncomplete="status.hide()"/>

<p:dialog widgetVar="status" modal="true" closable="false">
    Please Wait
</p:dialog>
```

## Client Side API

Widget: *PrimeFaces.widget.AjaxStatus*

Method	Params	Return Type	Description
bindFacet(eventName, facetId)	eventName: Name of status event, facetId: Element id of facet container	void	Binds a facet to an event
bindCallback(eventName, fn)	eventName: Name of status event, fn: function to bind	void	Binds a function to an event

## Skinning

AjaxStatus is equipped with style and styleClass. Styling directly applies to an html div element which contains the facets.

```
<p:ajaxStatus style="width:32px;height:32px" ... />
```

## Tips

- Avoid updating ajaxStatus itself to prevent duplicate facet/callback bindings.
- Provide a fixed width/height to the ajaxStatus to prevent page layout from changing.
- Components like commandButton has an attribute (*global*) to control triggering of AjaxStatus.
- There is an ajax loading gif bundled with PrimeFaces which you can use;

```
<h:graphicImage library="primefaces" name="jquery/ui/ui-anim_basic_16x16.gif" />
```

## 3.4 AutoComplete

AutoComplete provides live suggestions while an input is being typed.



### Info

Tag	<b>autoComplete</b>
Component Class	<b>org.primefaces.component.autocomplete.AutoComplete</b>
Component Type	<b>org.primefaces.component.AutoComplete</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.AutoCompleteRenderer</b>
Renderer Class	<b>org.primefaces.component.autocomplete.AutoCompleteRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
value	null	Object	Value of the component than can be either an EL expression or a literal text.
converter	null	Converter/ String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id.

Name	Default	Type	Description
immediate	FALSE	Boolean	When set true, process validations logic is executed at apply request values phase for this component.
required	FALSE	Boolean	Marks component as required.
validator	null	MethodExpr	A method expression that refers to a method validationg the input.
valueChangeListener	null	MethodExpr	A method expression that refers to a method for handling a valuchangeevent.
requiredMessage	null	String	Message to be displayed when required field validation fails.
converterMessage	null	String	Message to be displayed when conversion fails.
validatorMessage	null	String	Message to be displayed when validation fails.
widgetVar	null	String	Name of the widget to access client side api.
var	null	String	Name of the iterator used in pojo based suggestion.
itemLabel	null	String	Label of the item.
itemValue	null	String	Value of the item.
completeMethod	null	MethodExpr	Method providing suggestions.
maxResults	10	Integer	Maximum number of results to be displayed.
minQueryLength	1	Integer	Number of characters to be typed before starting to query.
queryDelay	300	Integer	Delay to wait in milliseconds before sending each query to the server.
forceSelection	FALSE	Boolean	When enabled, autoComplete only accepts input from the selection list.
selectListener	null	MethodExpr	Server side listener to invoke when an item is selected.
onSelectUpdate	null	String	Component(s) to update with ajax after a suggested item is selected.
onstart	null	String	Client side callback to execute before ajax request to load suggestions begins.
oncomplete	null	String	Client side callback to execute after ajax request to load suggestions completes.
accesskey	null	String	Access key that when pressed transfers focus to the input element.
alt	null	String	Alternate textual description of the input field.
autocomplete	null	String	Controls browser autocomplete behavior.

Name	Default	Type	Description
dir	null	String	Direction indication for text that does not inherit directionality. Valid values are LTR and RTL.
disabled	FALSE	Boolean	Disables input field
label	null	String	A localized user presentable name.
lang	null	String	Code describing the language used in the generated markup for this component.
maxlength	null	Integer	Maximum number of characters that may be entered in this field.
onblur	null	String	Client side callback to execute when input element loses focus.
onchange	null	String	Client side callback to execute when input element loses focus and its value has been modified since gaining focus.
onclick	null	String	Client side callback to execute when input element is clicked.
ondblclick	null	String	Client side callback to execute when input element is double clicked.
onfocus	null	String	Client side callback to execute when input element receives focus.
onkeydown	null	String	Client side callback to execute when a key is pressed down over input element.
onkeypress	null	String	Client side callback to execute when a key is pressed and released over input element.
onkeyup	null	String	Client side callback to execute when a key is released over input element.
onmousedown	null	String	Client side callback to execute when a pointer button is pressed down over input element
onmousemove	null	String	Client side callback to execute when a pointer button is moved within input element.
onmouseout	null	String	Client side callback to execute when a pointer button is moved away from input element.
onmouseover	null	String	Client side callback to execute when a pointer button is moved onto input element.
onmouseup	null	String	Client side callback to execute when a pointer button is released over input element.
onselect	null	String	Client side callback to execute when text within input element is selected by user.
readonly	FALSE	Boolean	Flag indicating that this component will prevent changes by the user.

Name	Default	Type	Description
size	null	Integer	Number of characters used to determine the width of the input element.
style	null	String	Inline style of the input element.
styleClass	null	String	Style class of the input element.
tabindex	null	Integer	Position of the input element in the tabbing order.
title	null	String	Advisory tooltip information.

## Getting Started with AutoComplete

Suggestions are loaded by calling a server side completeMethod that takes a single string parameter which is the text entered. Since autoComplete is an input component, it requires a value as well.

```
<p:autoComplete value="#{bean.text}" completeMethod="#{bean.complete}" />
```

```
public class Bean {
    private String text;

    public List<String> complete(String query) {
        List<String> results = new ArrayList<String>();

        for (int i = 0; i < 10; i++)
            results.add(query + i);

        return results;
    }

    //getter setter
}
```

## Pojo Support

Most of the time, instead of simple strings you would need work with your domain objects, autoComplete supports this common use case with the use of a converter and data iterator.

Following example loads a list of players, itemLabel is the label displayed as a suggestion and itemValue is the submitted value. Note that when working with pojos, you need to plug-in your own converter.

```
<p:autoComplete value="#{playerBean.selectedPlayer}"
    completeMethod="#{playerBean.completePlayer}"
    var="player"
    itemLabel="#{player.name}"
    itemValue="#{player}"
    converter="playerConverter"/>
```

```
import org.primefaces.examples.domain.Player;

public class PlayerBean {

    private Player selectedPlayer;

    public Player getSelectedPlayer() {
        return selectedPlayer;
    }

    public void setSelectedPlayer(Player selectedPlayer) {
        this.selectedPlayer = selectedPlayer;
    }

    public List<Player> complete(String query) {
        List<Player> players = readPlayersFromDatasource(query);

        return players;
    }
}
```

```
public class Player {

    private String name;

    //getter setter
}
```

## Limiting the results

Number of results shown can be limited, by default the limit is 10.

```
<p:autoComplete value="#{bean.text}"
    completeMethod="#{bean.complete}"
    maxResults="5" />
```

## Minimum query length

By default queries are sent to the server and completeMethod is called as soon as users starts typing at the input text. This behavior is tuned using the *minQueryLength* attribute.

```
<p:autoComplete value="#{bean.text}" completeMethod="#{bean.complete}"
    minQueryLength="3" />
```

With this setting, suggestions will start when user types the 3rd character at the input field.

## Query Delay

AutoComplete is optimized using *queryDelay* option, by default autoComplete waits for 300 milliseconds to query a suggestion request, if you'd like to tune the load balance, give a longer value. Following autoComplete waits for 1 second after user types an input.

```
<p:autoComplete value="#{bean.text}" completeMethod="#{bean.complete}"
    queryDelay="1000" />
```

## Instant Ajax Selection

Instead of waiting for user to submit the form manually to process the selected item, you can enable instant ajax selection by using a *selectListener* that takes an *org.primefaces.event.SelectEvent* instance as a parameter containing information about selected item. Optionally other component(s) on page can be updated after selection using *onSelectUpdate* option. Example below demonstrates how to display a facesmessage about the selected item instantly.

```
<p:messages id="messages" />

<p:autoComplete value="#{bean.text}" completeMethod="#{bean.complete}"
    onSelectUpdate="messages" selectListener="#{bean.handleSelect}" />
```

```
public class Bean {

    public void handleSelect(SelectEvent event) {
        Object item = event.getObject();
        FacesMessage msg = new FacesMessage("Selected", "Item:" + item);
    }

    //getter, setter and completeMethod
}
```

## Client Side Callbacks

*onstart* and *oncomplete* are used to execute custom javascript before and after an ajax request to load suggestions.

```
<p:autoComplete value="#{bean.text}" completeMethod="#{bean.complete}"
    onstart="handleStart(request)" oncomplete="handleComplete(response)" />
```

onstart callback gets a *request* parameter and oncomplete gets a *response* parameter, these parameters contain useful information. For example *request.term* is the query string and *response.results* is the suggestion list in json format.

## Client Side API

Widget: *PrimeFaces.widget.AutoComplete*

Method	Params	Return Type	Description
search(value)	value: keyword for search	void	Initiates a search with given value
close()	-	void	Close suggested items menu
disable()	-	void	Disables the input field
enable()	-	void	Enables the input field
deactivate()	-	void	Deactivates search behavior
activate()	-	void	Activates search behavior

## Skinning

Following is the list of structural style classes;

Class	Applies
.ui-autocomplete	Suggestion menu
.ui-autocomplete-input	Input field
.ui-autocomplete .ui-menu-item	Each item in suggestion menu

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## Tips

- Do not forget to use a converter when working with pojos.
- Enable forceSelection if you'd like to accept values only from suggested list.
- Increase query delay to avoid unnecessary load to server as a result of user typing fast.
- AutoComplete supports client behaviors like f:ajax and p:ajax.

## 3.5 BreadCrumb

Breadcrumb is a navigation component that provides contextual information about page hierarchy in the workflow.

```

    <span><a href="#"><img alt="Home icon" /></a></span> <span><a href="#">Sports</a></span> <span><a href="#">Football</a></span> <span><a href="#">Countries</a></span> <span><a href="#">Spain</a></span> <span><a href="#">F.C. Barcelona</a></span> <span><a href="#">Squad</a></span> <span><a href="#">Lionel Messi</a></span>
  
```

### Info

Tag	<b>breadCrumb</b>
Component Class	<b>org.primefaces.component.breadcrumb.BreadCrumb</b>
Component Type	<b>org.primefaces.component.BreadCrumb</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.BreadCrumbRenderer</b>
Renderer Class	<b>org.primefaces.component.breadcrumb.BreadCrumbRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
expandedEndItems	1	Integer	Number of expanded menuitems at the end.
expandedBeginningItems	1	Integer	Number of expanded menuitems at begining.
expandEffectDuration	800	Integer	Expanded effect duration in milliseconds.
collapseEffectDuration	500	Integer	Collapse effect duration in milliseconds.
initialCollapseEffectDuration	600	Integer	Initial collapse effect duration in milliseconds.
previewWidth	5	Integer	Preview width of a collapsed menuitem.
preview	FALSE	Boolean	Specifies preview mode, when set to false menuitems will not collapse.
style	null	String	Style of main container element.

Name	Default	Type	Description
styleClass	null	String	Style class of main container
model	null	MenuModel	MenuModel instance to create menus programmatically

## Getting Started with BreadCrumb

Steps are defined as child menuitem components in breadcrumb.

```
<p:breadCrumb>
    <p:menuItem label="Categories" url="#" />
    <p:menuItem label="Sports" url="#" />
    <p:menuItem label="Football" url="#" />
    <p:menuItem label="Countries" url="#" />
    <p:menuItem label="Spain" url="#" />
    <p:menuItem label="F.C. Barcelona" url="#" />
    <p:menuItem label="Squad" url="#" />
    <p:menuItem label="Lionel Messi" url="#" />
</p:breadCrumb>
```

## Preview

By default all menuitems are expanded, if you have limited space and many menuitems, breadcrumb can collapse/expand menuitems on mouseover. *previewWidth* attribute defines the reveal amount in pixels.

```
<p:breadCrumb preview="true">
    <p:menuItem label="Categories" url="#" />
    <p:menuItem label="Sports" url="#" />
    <p:menuItem label="Football" url="#" />
    <p:menuItem label="Countries" url="#" />
    <p:menuItem label="Spain" url="#" />
    <p:menuItem label="F.C. Barcelona" url="#" />
    <p:menuItem label="Squad" url="#" />
    <p:menuItem label="Lionel Messi" url="#" />
</p:breadCrumb>
```

## Dynamic Menus

Menus can be created programmatically as well, see the dynamic menus part in menu component section for more information and an example.

## Animation Configuration

Duration of effects can be customized using several attributes. Here's an example;

```
<p:breadcrumb preview="true" expandEffectDuration="1000"
               collapseEffectDuration="1000"
               initialCollapseEffectDuration="1000">
```

Durations are defined in milliseconds.

## Skinning

Breadcrumb resides in a main container element which *style* and *styleClass* options apply.

Following is the list of structural style classes;

Style Class	Applies
.ui-breadcrumb	Main breadcrumb container element.
.ui-breadcrumb ul	Container list of each menuitem.
.ui-breadcrumb ul li a	Each menuitem container.
.ui-breadcrumb ul li a	Link element of each menuitem.
.ui-breadcrumb ul li.first a	First element of breadcrumb.
.ui-breadcrumb-chevron	Separator of menuitems.

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;

Home > Sports > Football > Countries > Spain > F.C. Barcelona > Squad > Lionel Messi

## Tips

- If there is a dynamic flow, use model option instead of creating declarative p:menuitem components and bind your MenuModel representing the state of the flow.
- Breadcrumb can do ajax/non-ajax action requests as well since p:menuitem has this option. In this case, breadcrumb must be nested in a form.
- url option is the key for a menuitem, if it is defined, it will work as a simple link. If you'd like to use menuitem to execute command with or without ajax, do not define the url option.

## 3.6 Button

Button is an extension to the standard h:button component with skinning capabilities.



### Info

Tag	<b>button</b>
Component Class	<b>org.primefaces.component.button.Button</b>
Component Type	<b>org.primefaces.component.Button</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.ButtonRenderer</b>
Renderer Class	<b>org.primefaces.component.button.ButtonRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
value	null	Object	Value of the component than can be either an EL expression or a literal text.
widgetVar	null	String	Name of the widget to access client side api.
outcome	null	String	Used to resolve a navigation case.
includeViewParams	FALSE	Boolean	Whether to include page parameters in target URI
fragment	null	String	Identifier of the target page which should be scrolled to.
accesskey	null	String	Access key that when pressed transfers focus to button.
alt	null	String	Alternate textual description.
dir	null	String	Direction indication for text that does not inherit directionality. Valid values are LTR and RTL.
disabled	FALSE	Boolean	Disables button.

Name	Default	Type	Description
lang	null	String	Code describing the language used in the generated markup for this component.
onblur	null	String	Client side callback to execute when button loses focus.
onchange	null	String	Client side callback to execute when button loses focus and its value has been modified since gaining focus.
onclick	null	String	Client side callback to execute when button is clicked.
ondblclick	null	String	Client side callback to execute when button is double clicked.
onfocus	null	String	Client side callback to execute when button receives focus.
onkeydown	null	String	Client side callback to execute when a key is pressed down over button.
onkeypress	null	String	Client side callback to execute when a key is pressed and released over button.
onkeyup	null	String	Client side callback to execute when a key is released over button.
onmousedown	null	String	Client side callback to execute when a pointer button is pressed down over button.
onmousemove	null	String	Client side callback to execute when a pointer button is moved within button
onmouseout	null	String	Client side callback to execute when a pointer button is moved away from button.
onmouseover	null	String	Client side callback to execute when a pointer button is moved onto button.
onmouseup	null	String	Client side callback to execute when a pointer button is released over button.
style	null	String	Inline style of the button.
styleClass	null	String	Style class of the button.
readOnly	FALSE	Boolean	Makes button read only.
tabindex	null	Integer	Position in the tabbing order.
title	null	String	Advisory tooltip information.

## Getting Started with Button

p:button usage is same as standard h:button, an outcome is necessary to navigate using GET requests. Assume you are at source.xhtml and need to navigate target.xhtml.

```
<p:button outcome="target" value="Navigate"/>
```

## Parameters

Parameters in URI are defined with nested <f:param /> tags.

```
<p:button outcome="target" value="Navigate">
    <f:param name="id" value="10" />
</p:button>
```

## Icons

Icons for button are defined via css and *image* attribute, if you use title instead of value, only icon will be displayed and title text will be displayed as tooltip on mouseover.

```
<p:button outcome="target" image="star" value="With Icon"/>
<p:button outcome="target" image="star" title="With Icon"/>
```

```
.star {
    background-image: url("images/star.png");
}
```

## Skinning

Button renders a *button* tag which *style* and *styleClass* applies. Following is the list of structural style classes;

Style Class	Applies
.ui-button	Button element
.ui-button-text-only	Button element when icon is not used
.ui-button-text	Label of button

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.7 Calendar

Calendar is an input component used to provide a date. Other than basic features calendar supports paging, localization, ajax selection and more.



### Info

Tag	<code>calendar</code>
Component Class	<code>org.primefaces.component.calendar.Calendar</code>
Component Type	<code>org.primefaces.component.Calendar</code>
Component Family	<code>org.primefaces.component</code>
Renderer Type	<code>org.primefaces.component.CalendarRenderer</code>
Renderer Class	<code>org.primefaces.component.calendar.CalendarRenderer</code>

### Attributes

Name	Default	Type	Description
<code>id</code>	null	String	Unique identifier of the component
<code>rendered</code>	TRUE	Boolean	Boolean value to specify the rendering of the component.
<code>binding</code>	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
<code>value</code>	null	<code>java.util.Date</code>	Value of the component
<code>converter</code>	null	Converter/String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id

Name	Default	Type	Description
immediate	FALSE	Boolean	When set true, process validations logic is executed at apply request values phase for this component.
required	FALSE	Boolean	Marks component as required
validator	null	MethodExpr	A method expression that refers to a method validationg the input
valueChangeListener	null	MethodExpr	A method expression that refers to a method for handling a valuchangeevent
requiredMessage	null	String	Message to be displayed when required field validation fails.
converterMessage	null	String	Message to be displayed when conversion fails.
validatorMessage	null	String	Message to be displayed when validation fails.
mindate	null	Date or String	Sets calendar's minimum visible date
maxdate	null	Date or String	Sets calendar's maximum visible date
pages	int	1	Enables multiple page rendering.
disabled	FALSE	Boolean	Disables the calendar when set to true.
mode	popup	String	Defines how the calendar will be displayed.
pattern	MM/dd/yyyy	String	DateFormat pattern for localization
locale	null	java.util.Locale or String	Locale to be used for labels and conversion.
popupIcon	null	String	Icon of the popup button
popupIconOnly	FALSE	Boolean	When enabled, popup icon is rendered without the button.
navigator	FALSE	Boolean	Enables month/year navigator
timeZone	null	java.util.TimeZone	String or a java.util.TimeZone instance to specify the timezone used for date conversion, defaults to TimeZone.getDefault()
readOnlyInputText	FALSE	Boolean	Makes input text of a popup calendar readonly.
onSelectUpdate	null	String	Component(s) to update with ajax when a date is selected.
onSelectProcess	@this	String	Component(s) to process with ajax when a date is selected.

Name	Default	Type	Description
	null	MethodExpr	Server side listener to be invoked with ajax when a date is selected.
style	null	String	Style of the main container element.
styleClass	null	String	Style class of the main container element.
inputStyle	null	String	Style of the input element.
inputStyleClass	null	String	Style class of the input element.
showButtonPanel	FALSE	Boolean	Visibility of button panel containing today and done buttons.
effect	null	String	Effect to use when displaying and showing the popup calendar.
effectDuration	normal	String	Duration of the effect.
showOn	both	String	Client side event that displays the popup calendar.
showWeek	FALSE	Boolean	Displays the week number next to each week.
showOtherMonths	FALSE	Boolean	Displays days belonging to other months.
selectOtherMonths	FALSE	Boolean	Enables selection of days belonging to other months.
widgetVar	null	String	Name of the widget to access client side api.
yearRange	null	String	Year range for the navigator, default "c-10:c+10"
accesskey	null	String	Access key that when pressed transfers focus to the input element.
alt	null	String	Alternate textual description of the input field.
autocomplete	null	String	Controls browser autocomplete behavior.
dir	null	String	Direction indication for text that does not inherit directionality. Valid values are LTR and RTL.
label	null	String	A localized user presentable name.
lang	null	String	Code describing the language used in the generated markup for this component.
maxlength	null	Integer	Maximum number of characters that may be entered in this field.
onblur	null	String	Client side callback to execute when input element loses focus.

Name	Default	Type	Description
onchange	null	String	Client side callback to execute when input element loses focus and its value has been modified since gaining focus.
onclick	null	String	Client side callback to execute when input element is clicked.
ondblclick	null	String	Client side callback to execute when input element is double clicked.
onfocus	null	String	Client side callback to execute when input element receives focus.
onkeydown	null	String	Client side callback to execute when a key is pressed down over input element.
onkeypress	null	String	Client side callback to execute when a key is pressed and released over input element.
onkeyup	null	String	Client side callback to execute when a key is released over input element.
onmousedown	null	String	Client side callback to execute when a pointer button is pressed down over input element
onmousemove	null	String	Client side callback to execute when a pointer button is moved within input element.
onmouseout	null	String	Client side callback to execute when a pointer button is moved away from input element.
onmouseover	null	String	Client side callback to execute when a pointer button is moved onto input element.
onmouseup	null	String	Client side callback to execute when a pointer button is released over input element.
onselect	null	String	Client side callback to execute when text within input element is selected by user.
readonly	FALSE	Boolean	Flag indicating that this component will prevent changes by the user.
size	null	Integer	Number of characters used to determine the width of the input element.
tabindex	null	Integer	Position of the input element in the tabbing order.
title	null	String	Advisory tooltip information.

## Getting Started with Calendar

Value of the calendar should be a java.util.Date.

```
<p:calendar value="#{dateBean.date}" />
```

```
public class DateBean {  
    private Date date;  
    //Getter and Setter  
}
```

## Display Modes

Calendar has two main display modes, *popup* (default) and *inline*.

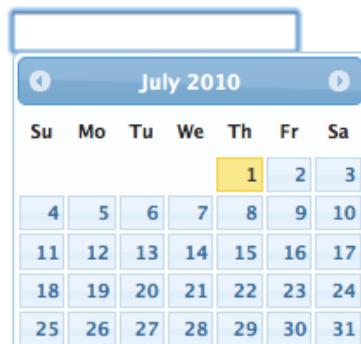
### Inline

```
<p:calendar value="#{dateBean.date}" mode="inline" />
```



### Popup

```
<p:calendar value="#{dateBean.date}" mode="popup" />
```

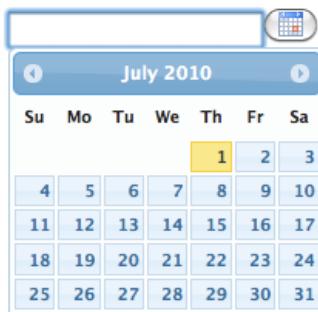


*showOn* option defines the client side event to display the calendar. Valid values are;

- focus: When input field receives focus
- button: When popup button is clicked
- both: Both *focus* and *button* cases

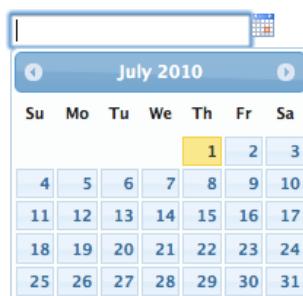
### Popup Button

```
<p:calendar value="#{dateBean.date}" mode="popup" showOn="button" />
```



### Popup Icon Only

```
<p:calendar value="#{dateBean.date}" mode="popup"
            showOn="button" popupIconOnly="true" />
```



### Paging

Calendar can also be rendered in multiple pages where each page corresponds to one month. This feature is tuned with the *pages* attribute.

```
<p:calendar value="#{dateController.date}" pages="3"/>
```

July 2010							August 2010							September 2010						
Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
4	5	6	7	8	9	10	1	2	3	4	5	6	7	1	2	3	4	5	6	7
11	12	13	14	15	16	17	15	16	17	18	19	20	21	12	13	14	15	16	17	18
18	19	20	21	22	23	24	22	23	24	25	26	27	28	19	20	21	22	23	24	25
25	26	27	28	29	30	31	29	30	31					26	27	28	29	30		

## I18N

By default locale information is retrieved from the view's locale and can be overridden by the locale attribute. Locale attribute can take a locale key as a String or a java.util.Locale instance. Following is a Turkish Calendar.

```
<p:calendar value="#{dateController.date}" locale="tr" navigator="true"
    showButtonPanel="true"/>
```



52 languages are supported out of the box;

- |  |  |
|--|--|
| <ul style="list-style-type: none"> <li>• Afrikaans</li> <li>• Albanian</li> <li>• Arabic</li> <li>• Armenian</li> <li>• Azerbaijani</li> <li>• Basque</li> <li>• Bosnian</li> <li>• Bulgarian</li> <li>• Catalan</li> <li>• Chinese Hong Kong</li> <li>• Chinese Simplified</li> <li>• Chinese Traditional</li> <li>• Croatian</li> <li>• Czech</li> <li>• Danish</li> <li>• Dutch</li> <li>• English UK</li> <li>• English US</li> <li>• Esperanto</li> <li>• Estonian</li> <li>• Faroese</li> <li>• Farsi/Persian</li> <li>• Finnish</li> <li>• French</li> <li>• French/Swiss</li> <li>• German</li> <li>• Greek</li> </ul> | <ul style="list-style-type: none"> <li>• Hebrew</li> <li>• Hungarian</li> <li>• Icelandic</li> <li>• Indonesian</li> <li>• Italian</li> <li>• Japanese</li> <li>• Korean</li> <li>• Latvian</li> <li>• Lithuanian</li> <li>• Malaysian</li> <li>• Norwegian</li> <li>• Polish</li> <li>• Portuguese/Brazilian</li> <li>• Romanian</li> <li>• Russian</li> <li>• Serbian</li> <li>• Serbian (sprski jezik)</li> <li>• Slovak</li> <li>• Slovenian</li> <li>• Spanish</li> <li>• Swedish</li> <li>• Tamil</li> <li>• Thai</li> <li>• Turkish</li> <li>• Ukrainian</li> <li>• Vietnamese</li> </ul> |
|--|--|

## L11N

Locale option is used for localizing the labels, not for localizing the date pattern. Default pattern is "MM/dd/yyyy" and pattern attribute is used to provide a custom date pattern.

```
<p:calendar value="#{dateController.date1}" pattern="dd.MM.yyyy"/>
<p:calendar value="#{dateController.date2}" pattern="yy, M, d"/>
<p:calendar value="#{dateController.date3}" pattern="EEE, dd MMM, yyyy"/>
```

dd.MM.yyyy	<input type="text" value="06.07.2010"/>
yy, M, d	<input type="text" value="10, 7, 13"/>
EEE, dd MMM, yyyy	<input type="text" value="Fri, 23 Jul, 2010"/>

## Effects

Various effects can be used when showing and hiding the popup calendar, options are;

- show
- slideDown
- fadeIn
- blind
- bounce
- clip
- drop
- fold
- slide

## Ajax Selection

Calendar supports instant ajax selection which means whenever a date is selected a server side selectListener can be invoked with an *org.primefaces.event.DateSelectEvent* instance as a parameter. Optional *onSelectUpdate* option allows updating other component(s) on page.

```
<p:calendar value="#{calendarBean.date}" onSelectUpdate="messages"
            selectListener="#{calendarBean.handleDateSelect}" />
<p:messages id="messages" />
```

```
public void handleDateSelect(DateSelectEvent event) {
    Date date = event.getDate();
    //Add facesmessage
}
```

## Date Ranges

Using mindate and maxdate options, selectable dates can be restricted. Values for these attributes can either be a string or a java.util.Date.

```
<p:calendar value="#{dateBean.date}" mode="inline"
    mindate="07/10/2010" maxdate="07/15/2010"/>
```



## Navigator

Navigator is an easy way to jump between months/years quickly.

```
<p:calendar value="#{dateBean.date}" mode="inline" navigator="true" />
```



## Client Side API

Widget: *PrimeFaces.widget.Calendar*

Method	Params	Return Type	Description
getDate()	-	Date	Return selected date
setDate(date)	date: Date to display	void	Sets display date
disable()	-	void	Disables calendar
enable()	-	void	Enables calendar

## Skinning

Calendar resides in a container element which *style* and *styleClass* options apply.

Following is the list of structural style classes;

Style Class	Applies
.ui-datepicker	Main container
.ui-datepicker-header	Header container
.ui-datepicker-prev	Previous month navigator
.ui-datepicker-next	Next month navigator
.ui-datepicker-title	Title
.ui-datepicker-month	Month display
.ui-datepicker-table	Date table
.ui-datepicker-week-end	Label of weekends
.ui-datepicker-other-month	Dates belonging to other months
.ui-datepicker td	Each cell date
.ui-datepicker-buttonpane	Button panel
.ui-datepicker-current	Today button
.ui-datepicker-close	Close button

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## Tips

- Navigator year range can be modified using *yearRange* option, defaults to c-10:c+10
- Calendar supports client behaviors like f:ajax and p:ajax.

## 3.8 Captcha

Captcha is a form validation component based on Recaptcha API.



### Info

Tag	<code>captcha</code>
Component Class	<code>org.primefaces.component.captcha.Captcha</code>
Component Type	<code>org.primefaces.component.Captcha</code>
Component Family	<code>org.primefaces.component</code>
Renderer Type	<code>org.primefaces.component.CaptchaRenderer</code>
Renderer Class	<code>org.primefaces.component.captcha.CaptchaRenderer</code>

### Attributes

Name	Default	Type	Description
<code>id</code>	null	String	Unique identifier of the component.
<code>rendered</code>	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
<code>binding</code>	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
<code>value</code>	null	Object	Value of the component than can be either an EL expression of a literal text.
<code>converter</code>	null	Converter/ String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id.
<code>immediate</code>	FALSE	Boolean	When set true, process validations logic is executed at apply request values phase for this component.
<code>required</code>	FALSE	Boolean	Marks component as required.

Name	Default	Type	Description
validator	null	MethodExpr	A method binding expression that refers to a method validationg the input.
valueChangeListener	null	ValueChange Listener	A method binding expression that refers to a method for handling a valuchangeevent.
requiredMessage	null	String	Message to be displayed when required field validation fails.
converterMessage	null	String	Message to be displayed when conversion fails.
validatorMessage	null	String	Message to be displayed when validation fields.
publicKey	null	String	Public recaptcha key for a specific domain (deprecated)
theme	red	String	Theme of the captcha.
language	en	String	Key of the supported languages.
tabindex	null	Integer	Position of the input element in the tabbing order.
label	null	String	A localized user presentable name.
secure	FALSE	Boolean	Enables https support

## Getting Started with Captcha

Catpcha is implemented as an input component with a built-in validator that is integrated with reCaptcha. First thing to do is to sign up to reCaptcha to get public&private keys. Once you have the keys for your domain, add them to web.xml as follows;

```
<context-param>
    <param-name>primefaces.PRIVATE_CAPTCHA_KEY</param-name>
    <param-value>YOUR_PRIVATE_KEY</param-value>
</context-param>

<context-param>
    <param-name>primefaces.PUBLIC_CAPTCHA_KEY</param-name>
    <param-value>YOUR_PUBLIC_KEY</param-value>
</context-param>
```

That is it, now you can use captcha as follows;

```
<p:captcha />
```

## Themes

Captcha features following built-in themes for look and feel customization.

- red (default)
- white
- blackglass
- clean

Themes are applied via the theme attribute.

```
<p:captcha theme="white"/>
```



## Languages

Text instructions displayed on captcha is customized with the *language* attribute. Below is a captcha with Turkish text.

```
<p:captcha language="tr"/>
```

## Overriding Validation Messages

By default captcha displays its own validation messages, this can be easily overridden by the JSF message bundle mechanism. Corresponding keys are;

Summary	primefaces.captcha.INVALID
Detail	primefaces.captcha.INVALID_detail

## Tips

- Use label option to provide readable error messages in case validation fails.
- Enable *secure* option to support https otherwise browsers will give warnings.
- See <http://www.google.com/recaptcha/learnmore> to learn more about how reCaptcha works.

## 3.9 Carousel

Carousel is a multi purpose component to display a set of data or general content with slide effects.



### Info

Tag	<b>carousel</b>
Component Class	<b>org.primefaces.component.carousel.Carousel</b>
Component Type	<b>org.primefaces.component.Carousel</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.CarouselRenderer</b>
Renderer Class	<b>org.primefaces.component.carousel.CarouselRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	A value expression that refers to a collection
var	null	String	Name of the request scoped iterator
rows	3	Integer	Number of visible items per page
first	0	Integer	Index of the first element to be displayed
scrollIncrement	1	Integer	Number of items to pass in each scroll
circular	FALSE	Boolean	Sets continuous scrolling
vertical	FALSE	Boolean	Sets vertical scrolling

Name	Default	Type	Description
autoPlayInterval	0	Integer	Sets the time in milliseconds to have Carousel start scrolling automatically after being initialized
revealAmount	0	Integer	The percentage of the previous and next item of the current item to be revealed
animate	TRUE	Boolean	When enabled scrolling is animated, animation is turned on by default
speed	0.5	Double	Sets the speed of the scrolling animation
effect	null	String	Name of the animation effect
pagerPrefix	null	String	Prefix text of the pager dropdown.
style	null	String	Inline style of the main container.
styleClass	null	String	Style class of the main container.
itemStyle	null	String	Inline style of each item container.
itemStyleClass	null	String	Style class of each item container.
widgetVar	null	String	Name of the widget to access client side api

## Getting Started with Carousel

Calendar has two main use-cases; data and general content display. To begin with data iteration let's use a list of cars to display with carousel.

```
public class Car {
    private String model;
    private int year;
    private String manufacturer;
    private String color;
    ...
}
```

```
public class CarBean {
    private List<Car> cars;

    public CarListController() {
        cars = new ArrayList<Car>();
        cars.add(new Car("myModel", 2005, "ManufacturerX", "blue"));
        //add more cars
    }

    //getter setter
}
```

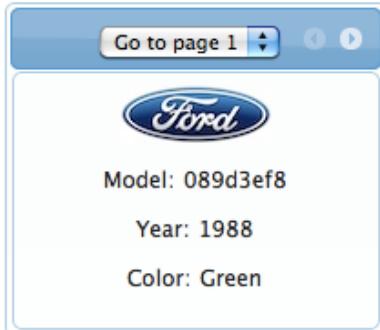
```
<p:carousel value="#{carBean.cars}" var="car">
    <p:graphicImage value="/images/cars/#{car.manufacturer}.jpg"/>
    <h:outputText value="Model: #{car.model}" />
    <h:outputText value="Year: #{car.year}" />
    <h:outputText value="Color: #{car.color}" />
</p:carousel>
```

Carousel iterates through the cars collection and renders it's children for each car.

## Limiting Visible Items

Bu default carousel lists it's items in pages with size 3. This is customizable with the rows attribute.

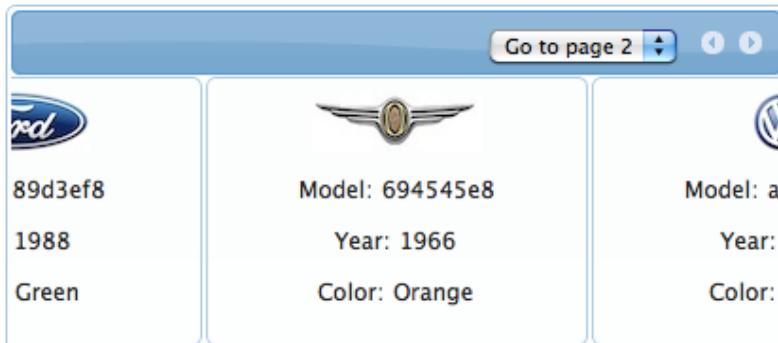
```
<p:carousel value="#{carBean.cars}" var="car" rows="1">
    ...
</p:carousel>
```



## Reveal Amount

Reveal amount is the percentage of the next and previous item to be shown, it can be tuned by the *revealAmount* attribute. Example above reveals %20 of the next and previous items.

```
<p:carousel value="#{carBean.cars}" var="car" revealAmount="20">
    ...
</p:carousel>
```



## Effects

Paging happens with a slider effect by default and following easing options are supported.

- backBoth
- backIn
- backOut
- bounceBoth
- bounceIn
- bounceOut
- easeBoth
- easeBothStrong
- easeIn
- easeInStrong
- easeNone
- easeOut
- easeOutStrong
- elasticBoth
- elasticIn
- elasticOut

**Note:** Effect names are case sensitive and incorrect usage may result in javascript errors

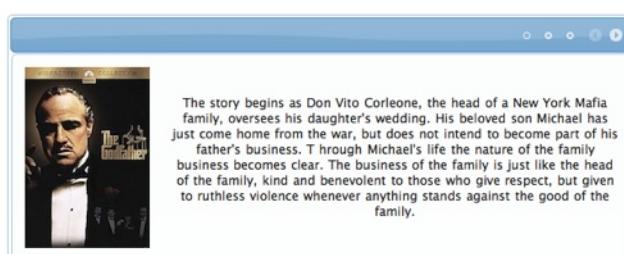
## SlideShow

Carousel can display the contents in a slideshow, for this purpose *autoPlayInterval* and *circular* attributes are used. Following carousel displays a collection of images as a slideshow.

```
<p:carousel itemStyleClass="carItem" autoPlayInterval="2000"
            rows="1" effect="easeInStrong" circular="true">
    <p:graphicImage value="/images/nature1.jpg"/>
    <p:graphicImage value="/images/nature2.jpg"/>
    <p:graphicImage value="/images/nature3.jpg"/>
    <p:graphicImage value="/images/nature4.jpg"/>
</p:carousel>
```

## Content Display

Another use case of carousel is tab based content display.



```
<p:carousel rows="1" itemStyle="height:200px; width:600px;">
    <p:tab title="Godfather Part I">
        <h:panelGrid columns="2" cellpadding="10">
            <p:graphicImage value="/images/godfather/godfather1.jpg" />
            <h:outputText value="The story begins as Don Vito ..." />
        </h:panelGrid>
    </p:tab>
    <p:tab title="Godfather Part II">
        <h:panelGrid columns="2" cellpadding="10">
            <p:graphicImage value="/images/godfather/godfather2.jpg" />
            <h:outputText value="Francis Ford Coppola's ..." />
        </h:panelGrid>
    </p:tab>
    <p:tab title="Godfather Part III">
        <h:panelGrid columns="2" cellpadding="10">
            <p:graphicImage value="/images/godfather/godfather3.jpg" />
            <h:outputText value="After a break of ..." />
        </h:panelGrid>
    </p:tab>
</p:carousel>
```

## Item Selection

When selecting an item from a carousel with a command component, p:column is necessary to process selection properly. Following example displays selected car contents within a dialog;

```
<h:form prependId="false">
    <p:carousel value="#{carBean.cars}" var="car">
        <p:column>
            <p:graphicImage value="/images/cars/#{car.manufacturer}.jpg"/>
            <p:commandLink update="detail" oncomplete="dlg.show()">
                <h:outputText value="Model: #{car.model}" />
                <f:setPropertyActionListener value="#{car}" target="#{carBean.selected}" />
            </p:commandLink>
        </p:column>
    </p:carousel>

    <p:dialog widgetVar="dlg">
        <h:outputText id="detail" value="#{carBean.selected}" />
    </p:dialog>
</h:form>
```

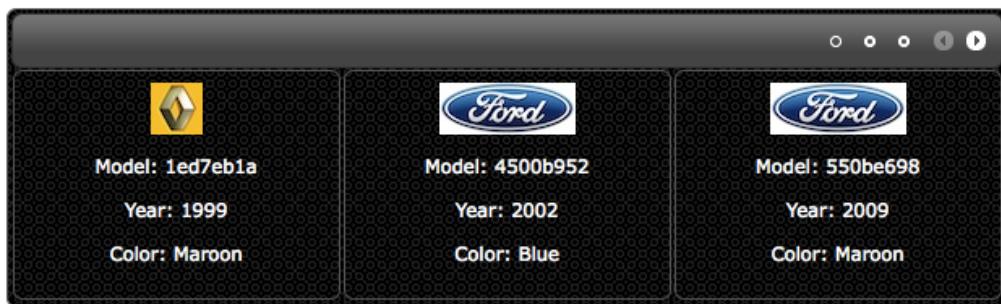
```
public class CarBean {
    private List<Car> cars;
    private Car selected;
    //getters and setters
}
```

## Skinning

Carousel resides in a container element which *style* and *styleClass* options apply. *itemStyle* and *itemStyleClass* attributes apply to each item displayed by carousel. Following is the list of structural style classes;

Style Class	Applies
.ui-carousel	Main container
.ui-carousel-nav	Header container
.ui-carousel-content	Content container
.ui-carousel-nav-first-page	First page of paginator
.ui-carousel-nav-page-selected	Current page of paginator
.ui-carousel-button	Navigation buttons
.ui-carousel-first-button	First navigation button of paginator
.ui-carousel-next-button	Next navigation button of paginator
.ui-carousel-element	Item container list
.ui-carousel-element li	Each item

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## Tips

- To avoid cross browser issues, provide dimensions of each item with *itemStyle* or *itemStyleClass* attributes.
- When selecting an item with a command component like `commandLink`, place carousel contents in a `p:column` to process selecting properly.

## 3.10 CellEditor

CellEditor is a helper component of datatable used for incell editing.

### Info

Tag	<b>cellEditor</b>
Component Class	<b>org.primefaces.component.celleditor.CellEditor</b>
Component Type	<b>org.primefaces.component.CellEditor</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.CellEditorRenderer</b>
Renderer Class	<b>org.primefaces.component.celleditor.CellEditorRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean

### Getting Started with CellEditor

See inline editing section in datatable documentation for more information about usage.

## 3.11 Charts

Charts are flash based JSF components to display graphical data. There're various chart types like pie, column, line and more. Charts can also display real-time data and fire server side events as a response to user interaction.

### 3.11.1 Pie Chart

Pie chart displays category-data pairs in a pie graphic.

#### Info

Tag	<b>pieChart</b>
Component Class	<b>org.primefaces.component.chart.pie.PieChart</b>
Component Type	<b>org.primefaces.component.chart.PieChart</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.chart.PieChartRenderer</b>
Renderer Class	<b>org.primefaces.component.chart.pie.PieChartRenderer</b>

#### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Collection	Datasource to be displayed on the chart
var	null	String	Name of the data iterator
categoryField	null	Object	Pie category field
dataField	null	Object	Pie data field
live	FALSE	Boolean	When a chart is live, the data is refreshed based on the refreshInterval period.
refreshInterval	3000	Integer	Refresh period of a live chart data in milliseconds
update	null	String	Component(s) to be updated after item selection.
oncomplete	null	String	Client side callback to execute when ajax request for item select event is completed.

Name	Default	Type	Description
itemSelectListener	null	MethodExpr	Method expression to listen chart series item select events
styleClass	null	String	Style to apply to chart container element
style	null	String	Javascript variable name representing the style
seriesStyle	null	String	Javascript variable name representing the series styles
width	500px	String	Width of the chart.
height	350px	String	Height of the chart.
dataTipFunction	null	String	Name of the javascript function to customize datatips.
wmode	null	String	wmode property of the flash object
model	null	ChartModel	ChartModel instance to create chart from.
widgetVar	null	String	Name of the client side widget

## Getting started with PieChart

There are two ways to create a PieChart, declarative way is to provide a java.util.Collection with your domain objects as the value and programmatic way is to create a ChartModel.

### Declarative (Deprecated)

In declarative way, charts needs a datasource like a java.util.List to display the data, in addition to the datasource *categoryField* is used to identify the pie section and *dataField* is used to hold the value of the corresponding categoryField. As an example, suppose there are 4 brands and each brand has made x amount of sales last year. Let's begin with creating the sale class to represent this model.

```
public class Sale {
    private String brand;
    private int amount;

    public Sale(String brand, int amount) {
        this.brand = brand;
        this.amount = amount;
    }

    //getters and setters
}
```

In SaleDisplay bean, a java.util.List holds sale data of the 4 brands.

```
public class ChartBean {

    private List<Sale> sales;

    public SaleDisplayBean() {
        sales = new ArrayList<Sale>();
        sales.add(new Sale("Brand 1", 540));
        sales.add(new Sale("Brand 2", 325));
        sales.add(new Sale("Brand 3", 702));
        sales.add(new Sale("Brand 4", 421));
    }

    public List<Sale> getSales() {
        return sales;
    }
}
```

That's all the information needed for the pieChart to start working. Sales list can be visualized as follows;

```
<p:pieChart value="#{chartBean.sales}" var="sale"
            categoryField="#{sale.brand}" dataField="#{sale.amount}" />
```

## ChartModel (Suggested)

Pie chart can also be created programmatically with an *org.primefaces.model.chart.PieChartModel* instance. Same sales chart can be implemented as follows;

```
public class SaleDisplayBean {

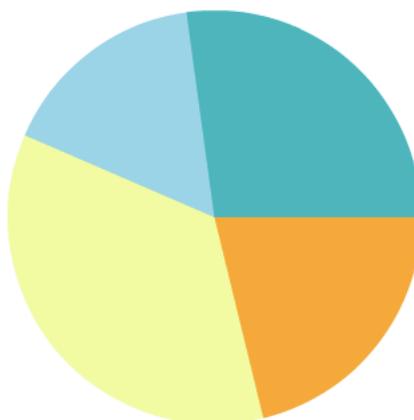
    private PieChartModel model;

    public SaleDisplayBean() {
        model = new PieChartModel();
        model.set("Brand 1", 540);
        model.set("Brand 2", 325);
        model.set("Brand 3", 702);
        model.set("Brand 4", 421);
    }

    public PieChartModel getModel() {
        return model;
    }
}
```

```
<p:pieChart model="#{chartBean.model}" />
```

Whether it is created declaratively or via a ChartModel, output would be;

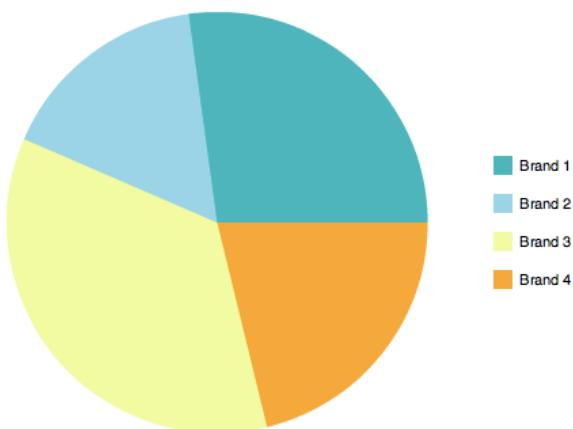


## PieChart Style

As chart is a flash component, it cannot be styled with CSS, instead use style attribute to define a javascript variable, following example demonstrates how to add legend to the pie chart.

```
<p:pieChart model="#{chartBean.model}" style="piechartStyle" />
```

```
<script type="text/javascript">
    var piechartStyle = {
        padding:20,
        legend: {
            display:"right"
            ,spacing:10
        }
    };
</script>
```



For more information about chart skinning, see main Charts Skinning section.

### 3.11.2 Line Chart

Line chart visualizes one or more series of data in a line graph.

#### Info

Tag	<b>lineChart</b>
Component Class	<b>org.primefaces.component.chart.line.LineChart</b>
Component Type	<b>org.primefaces.component.chart.LineChart</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.chart.LineChartRenderer</b>
Renderer Class	<b>org.primefaces.component.chart.line.LineChartRenderer</b>

#### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Collection	Datasource to be displayed on the chart
var	null	String	Name of the data iterator
xField	null	Object	Field for x-axis
live	FALSE	Boolean	When a chart is live, the data is refreshed based on the refreshInterval period.
refreshInterval	3000	Integer	Refresh period of a live chart data in milliseconds
update	null	String	Component(s) to be updated after item selection.
oncomplete	null	String	Client side callback to execute when ajax request for item select event is completed.
itemSelectListener	null	MethodExpr	Method expression to listen chart series item select events
styleClass	null	String	Style to apply to chart container element
style	null	String	Javascript variable name representing the styles
minY	null	Double	Minimum boundary value for y-axis.

Name	Default	Type	Description
maxY	null	Double	Maximum boundary value for y-axis.
width	500px	String	Width of the chart.
height	350px	String	Height of the chart.
dataTipFunction	null	String	Name of the javascript function to customize datatips.
labelFunctionX	null	String	Name of the javascript function to format x-axis labels.
labelFunctionY	null	String	Name of the javascript function to format y-axis labels.
titleX	null	String	Title of the x-axis
titleY	null	String	Title of the y-axis
wmode	null	String	wmode property of the flash object
model	null	ChartModel	ChartModel instance to create chart from.
widgetVar	null	String	Name of the client side widget

## Getting started with LineChart

LineChart can be created in two ways, one way is declaratively with representing each series using p:chartSeries component and other way is programmatic approach using a ChartModel.

### Declarative (Deprecated)

In declarative way, chart needs a collection of data as the value, the xField for the x-axis and one or more series data each corresponding to a line on the graph. As an example, let's display and compare the number of boys and girls year by year who were born last year at some place on earth. To model this, we need the Birth class.

```
public class Birth {
    private int year, boys, girls;

    public Birth(int year, int boys, int girls) {
        this.year = year;
        this.boys = boys;
        this.girls = girls;
    }
    //getters and setters for fields
}
```

Next thing to do is to prepare the data year by year in ChartBean.

```

public class ChartBean {

    private List<Birth> births;

    public ChartBean() {
        births = new ArrayList<Birth>();
        births.add(new Birth(2004, 120, 52));
        births.add(new Birth(2005, 100, 60));
        births.add(new Birth(2006, 44, 110));
        births.add(new Birth(2007, 150, 135));
        births.add(new Birth(2008, 125, 120));
    }

    public List<Birth> getBirths() {
        return births;
    }
}

```

A linechart can visualize this births collection as;

```

<p:lineChart value="#{chartBean.births}" var="birth" xfield="#{birth.year}">
    <p:chartSeries label="Boys" value="#{birth.boys}" />
    <p:chartSeries label="Girls" value="#{birth.girls}" />
</p:lineChart>

```

## ChartModel (Suggested)

Programmatic way is to create an instance of *org.primefaces.model.chart.CartesianChartModel* and bind it as the value of the linechart.

```

public class ChartBean {

    private CartesianChartModel model;

    public ChartBean() {
        model = new CartesianChartModel();

        ChartSeries boys = new ChartSeries();
        boys.setLabel("Boys");

        boys.set("2004", 120);
        boys.set("2005", 100);
        boys.set("2006", 44);
        boys.set("2007", 150);
        boys.set("2008", 25);
    }

    public CartesianChartModel getModel() {
        return model;
    }
}

```

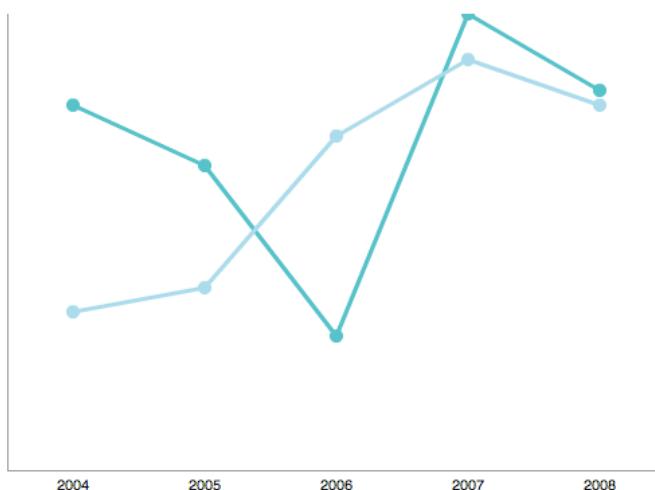
```
ChartSeries girls = new ChartSeries();
boys.setLabel("Boys");

girls.set("2004", 52);
girls.set("2005", 60);
girls.set("2006", 110);
girls.set("2007", 135);
girls.set("2008", 120);

}

public CartesianChartModel getModel() {
    return model;
}
}
```

Whether lineChart is created declaratively or programmatically, output would be same;



### 3.11.3 Column Chart

Column chart visualizes one or more series of data using vertical bars.

#### Info

Tag	<b>columnChart</b>
Component Class	<b>org.primefaces.component.chart.column.ColumnChart</b>
Component Type	<b>org.primefaces.component.chart.ColumnChart</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.chart.ColumnChartRenderer</b>
Renderer Class	<b>org.primefaces.component.chart.column.ColumnChartRenderer</b>

#### Attributes

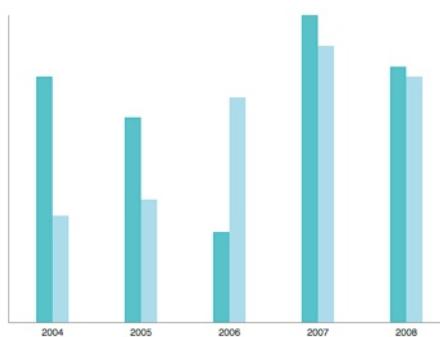
Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Collection	Datasource to be displayed on the chart
var	null	String	Name of the data iterator
xField	null	Object	Data of the x-axis
live	FALSE	boolean	When a chart is live, the data is refreshed based on the refreshInterval period.
refreshInterval	3000	Integer	Refresh period of a live chart data in milliseconds
update	null	String	Component(s) to be updated after item selection.
oncomplete	null	String	Client side callback to execute when ajax request for item select event is completed.
itemSelectListener	null	MethodExpr	Method expression to listen chart series item select events
styleClass	null	String	Style to apply to chart container element
style	null	String	Javascript variable name representing the styles
minY	null	Double	Minimum boundary value for y-axis.

Name	Default	Type	Description
maxY	null	double	Maximum boundary value for y-axis.
width	500px	String	Width of the chart.
height	350px	String	Height of the chart.
dataTipFunction	null	String	Name of the javascript function to customize datatips.
labelFunctionX	null	String	Name of the javascript function to format x-axis labels.
labelFunctionY	null	String	Name of the javascript function to format y-axis labels.
titleX	null	String	Title of the x-axis
titleY	null	String	Title of the y-axis
wmode	null	String	wmode property of the flash object
model	null	ChartModel	ChartModel instance to create chart from.
widgetVar	null	String	Name of the client side widget

## Getting Started with Column Chart

As column chart is a Cartesian chart, usage is same as linechart, as an example chart below displays the births collection example given in line chart section.

```
<p:columnChart value="#{chartBean.births}" var="birth" xfield="#{birth.year}">
    <p:chartSeries label="Boys" value="#{birth.boys}" />
    <p:chartSeries label="Girls" value="#{birth.girls}" />
</p:lineChart>
```



Column chart can be created with a chart model as well.

```
<p:columnChart model="#{chartBean.model}" />
```

### 3.11.4 Stacked Column Chart

Stacked Column chart is similar to column chart but series are displayed as stacked.

#### Info

Tag	<b>stackedColumnChart</b>
Component Class	<b>org.primefaces.component.chart.stackedcolumn.StackedColumnChart</b>
Component Type	<b>org.primefaces.component.chart.StackedColumnChart</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.chart.StackedColumnChartRenderer</b>
Renderer Class	<b>org.primefaces.component.chart.stackedcolumn.StackedColumnChartRenderer</b>

#### Attributes

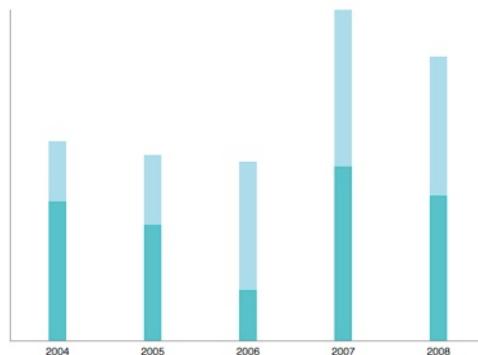
Name	Default	Type	Description
id	Assigned by JSF	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Collection	Datasource to be displayed on the chart
var	null	String	Name of the data iterator
xField	null	Object	Data of the x-axis
live	FALSE	Boolean	When a chart is live, the data is refreshed based on the refreshInterval period.
refreshInterval	3000	Integer	Refresh period of a live chart data in milliseconds
update	null	String	Component(s) to be updated after item selection.
oncomplete	null	String	Client side callback to execute when ajax request for item select event is completed.
itemSelectListener	null	MethodExpr	Method expression to listen chart series item select events
styleClass	null	String	Style to apply to chart container element
style	null	String	Javascript variable name representing the styles

Name	Default	Type	Description
minY	null	Double	Minimum boundary value for y-axis.
maxY	null	Double	Maximum boundary value for y-axis.
width	500px	String	Width of the chart.
height	350px	String	Height of the chart.
dataTipFunction	null	String	Name of the javascript function to customize datatips.
labelFunctionX	null	String	Name of the javascript function to format x-axis labels.
labelFunctionY	null	String	Name of the javascript function to format y-axis labels.
wmode	null	String	wmode property of the flash object
model	null	ChartModel	ChartModel instance to create chart from.
widgetVar	null	String	Name of the client side widget

## Getting started with Stacked Column Chart

As column chart is a Cartesian chart, usage is same as linechart, as an example chart below displays the births collection example given in line chart section.

```
<p:stackedColumnChart value="#{chartBean.births}"
    var="birth" xfield="#{birth.month}">
    <p:chartSeries label="Boys" value="#{birth.boys}" />
    <p:chartSeries label="Girls" value="#{birth.girls}" />
</p:stackedColumnChart>
```



Stacked column chart can be created with a chart model as well.

```
<p:stackedColumnChart model="#{chartBean.model}" />
```

### 3.11.5 Bar Chart

Bar Chart is similar to the column chart but bars are displayed horizontally.

#### Info

Tag	<b>barChart</b>
Component Class	<b>org.primefaces.component.chart.bar.BarChart</b>
Component Type	<b>org.primefaces.component.chart.BarChart</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.chart.BarChartRenderer</b>
Renderer Class	<b>org.primefaces.component.chart.bar.BarChartRenderer</b>

#### Attributes

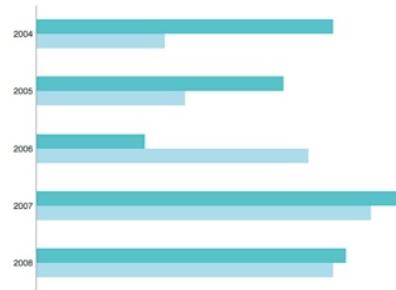
Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Collection	Datasource to be displayed on the chart
var	null	String	Name of the data iterator
yField	null	Object	Data of the y-axis
live	FALSE	Boolean	When a chart is live, the data is refreshed based on the refreshInterval period.
refreshInterval	3000	Integer	Refresh period of a live chart data in milliseconds
update	null	String	Component(s) to be updated after item selection.
oncomplete	null	String	Client side callback to execute when ajax request for item select event is completed.
itemSelectListener	null	MethodExpr	Method expression to listen chart series item select events
styleClass	null	String	Style to apply to chart container element
style	null	String	Javascript variable name representing the styles
minX	null	Double	Minimum boundary value for x-axis.

Name	Default	Type	Description
maxX	null	Double	Maximum boundary value for x-axis.
width	500px	String	Width of the chart.
height	350px	String	Height of the chart.
dataTipFunction	null	String	Name of the javascript function to customize datatips.
labelFunctionX	null	String	Name of the javascript function to format x-axis labels.
labelFunctionY	null	String	Name of the javascript function to format y-axis labels.
titleX	null	String	Title of the x-axis
titleY	null	String	Title of the y-axis
wmode	null	String	wmode property of the flash object
model	null	ChartModel	ChartModel instance to create chart from.
widgetVar	null	String	Name of the client side widget

## Getting Started with Bar Chart

As column chart is a Cartesian chart, usage is similar to linechart, important difference is that yfield is used instead of xfield. As an example chart below displays the births collection example given in line chart section.

```
<p:barChart value="#{chartBean.births}" var="birth" yfield="#{birth.month}">
    <p:chartSeries label="Boys" value="#{birth.boys}" />
    <p:chartSeries label="Girls" value="#{birth.girls}" />
</p:barChart>
```



Column chart can be created with a cartesian chart model as well.

```
<p:barChart model="#{chartBean.model}" />
```

### 3.11.6 StackedBar Chart

Stacked Bar chart is similar to bar chart but the series are displayed as stacked.

#### Info

Tag	<b>stackedBarChart</b>
Component Class	<b>org.primefaces.component.chart.stackedbar.StackedBarChart</b>
Component Type	<b>org.primefaces.component.chart.StackedBarChart</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.chart.StackedBarChartRenderer</b>
Renderer Class	<b>org.primefaces.component.chart.stackedbar.StackedBarChartRenderer</b>

#### Attributes

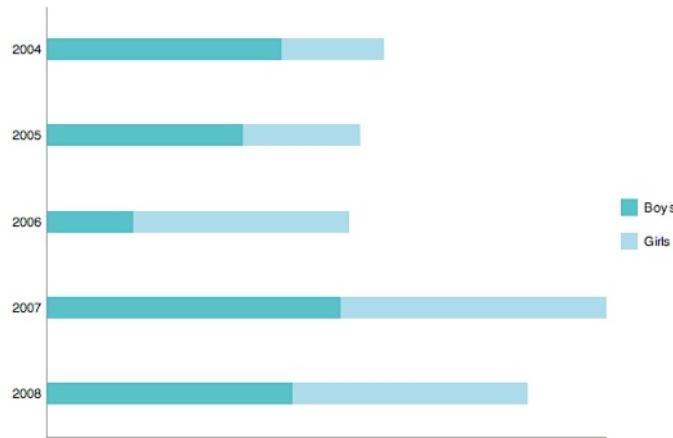
Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Collection	Datasource to be displayed on the chart
var	null	String	Name of the data iterator
yField	null	Object	Data of the y-axis
live	FALSE	Boolean	When a chart is live, the data is refreshed based on the refreshInterval period.
refreshInterval	3000	Integer	Refresh period of a live chart data in milliseconds
update	null	String	Component(s) to be updated after item selection.
oncomplete	null	String	Client side callback to execute when ajax request for item select event is completed.
itemSelectListener	null	MethodExpr	Method expression to listen chart series item select events
styleClass	null	String	Style to apply to chart container element
style	null	String	Javascript variable name representing the styles
minX	null	Double	Minimum boundary value for x-axis.

Name	Default	Type	Description
maxX	null	Double	Maximum boundary value for x-axis.
width	500px	String	Width of the chart.
height	350px	String	Height of the chart.
dataTipFunction	null	String	Name of the javascript function to customize datatips.
labelFunctionX	null	String	Name of the javascript function to format x-axis labels.
labelFunctionY	null	String	Name of the javascript function to format y-axis labels.
wmode	null	String	wmode property of the flash object
model	null	ChartModel	ChartModel instance to create chart from.
widgetVar	null	String	Name of the client side widget

## Getting Started with StackedBar Chart

StackedBar chart usage is very same as bar chart.

```
<p:stackedBarChart value="#{chartBean.births}" var="birth" yfield="#{birth.month}">
    <p:chartSeries label="Boys" value="#{birth.boys}" />
    <p:chartSeries label="Girls" value="#{birth.girls}" />
</p:stackedBarChart>
```



Column chart can be created with a cartesian chart model as well.

```
<p:stackedBarChart model="#{chartBean.model}" />
```

### 3.11.7 Chart Series

ChartSeries is used to define a series in a chart declaratively.

#### Info

Tag	<b>chartSeries</b>
Tag Class	<b>org.primefaces.component.chart.series.ChartSeriesTag</b>
Component Class	<b>org.primefaces.component.chart.series.ChartSeries</b>
Component Type	<b>org.primefaces.component.ChartSeries</b>
Component Family	<b>org.primefaces.component</b>

#### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Value to be displayed on the series
converter	null	Converter	Output converter to be used if any.
label	null	String	Label of the series
style	null	String	Javascript variable name representing the styles

#### Getting started with ChartSeries

ChartSeries is nested inside a chart component, you can have as many series as you want on a chart by nesting multiple series. See the other chart component documentations to see the usage of chartSeries.

### 3.11.8 Skinning Charts

Charts are highly customizable in terms of skinning however they are flash based so regular CSS styling is not possible. Charts are styled through Javascript and the object is passed to the chart's style attribute.

There are two attributes in chart components related to skinning.

*styleClass* : Each chart resides in an html div element, style class applies to this container element. Style class is mainly useful for setting the width and height of the chart.

```
<style type="text/css">
    .chartClass {
        width:700px;
        height:400px;
    }
</style>
```

*style* : Style should be a javascript object variable name, as a simple example to start with; Style below effects chart padding, border and legend. See the full list of style selectors link for the complete list of selectors.

```
var chartStyle = {
    padding : 20,
    border: {color: 0x96acb4, size: 8},
    legend: {
        display: "right"
    }
};
```

### Skinning Series

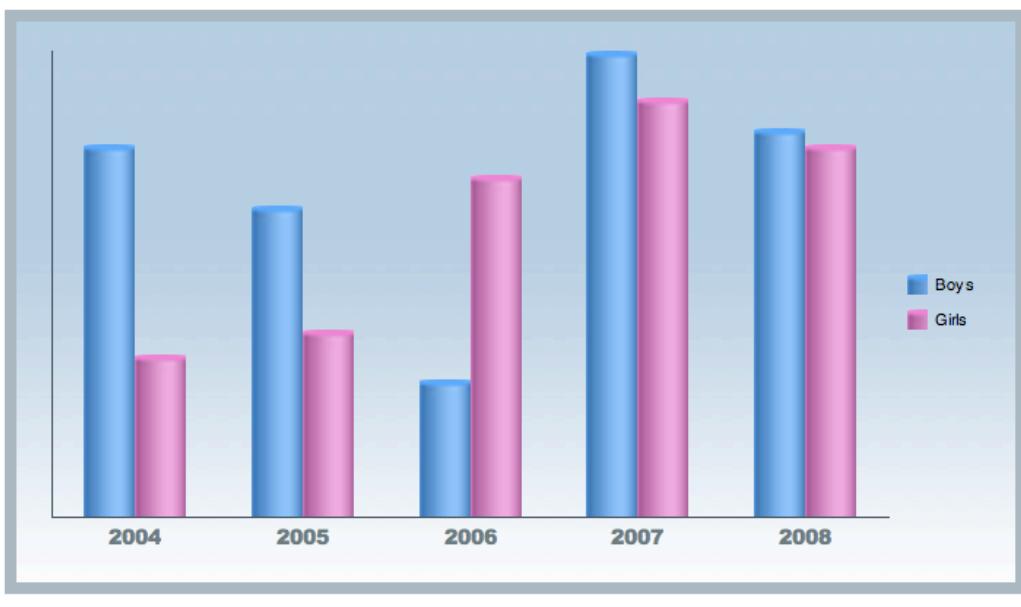
ChartSeries can be styled individually using the style attribute. Styling is same as charts and done via javascript.

```
var boysStyle = {
    color: 0x3399FF,
    size: 35
};
```

```
<p:chartSeries value="#{birth.boys}" label="Boys" style="boysStyle" />
```

## Extreme Makeover

To give a complete styling example, we'll skin the chart described in colum chart section. In the end after the extreme makeover chart will look like;



```

<style type="text/css">
    .chartClass {
        width:700px;
        height:400px;
    }
</style>

<script type="text/javascript">
    var chartStyle = {
        border: {color: 0x96acb4, size: 12},
        background: {
            image : "../design/bg.jpg"
        },
        font: {name: "Arial Black", size: 14, color: 0x586b71},
        dataTip: {
            border: {color: 0x2e434d, size: 2},
            font: {name: "Arial Black", size: 13, color: 0x586b71}
        },
        xAxis: {
            color: 0x2e434d
        },
        yAxis: {
            color: 0x2e434d,
            majorTicks: {color: 0x2e434d, length: 4},
            minorTicks: {color: 0x2e434d, length: 2},
            majorGridLines: {size: 0}
        }
    };
</script>

```

```
var boysSeriesStyle =  
{  
    image: "../design/column.png",  
    mode: "no-repeat",  
    color: 0x3399FF,  
    size: 35  
};  
  
var girlsSeriesStyle =  
{  
    image: "../design/column.png",  
    mode: "no-repeat",  
    color: 0xFF66CC,  
    size: 35  
};
```

```
<p:columnChart value="#{chartBean.births}" var="birth" xfield="#{birth.year}"  
    styleClass="column" style="chartStyle">  
    <p:chartSeries label="Boys" value="#{birth.boys}" style="boysSeriesStyle"/>  
    <p:chartSeries label="Girls" value="#{birth.girls}" style="girlsSeriesStyle"/>  
</p:columnChart>
```

Full list of skinning properties are available at

<http://developer.yahoo.com/yui/charts/#basicstyles>

### 3.11.9 Real-Time Charts

Charts have built-in support live data display using ajax polling. As an example suppose there's an ongoing vote between two candidates. Let's start with creating the *Vote* class representing the voting model.

```
public class Vote {
    private String candidate;
    private int count;
    public Vote() {
        //NoOp
    }
    public Vote(String candidate, int count) {
        this.candidate = candidate;
        this.count = count;
    }
    public void add(int count) {
        this.count = this.count + count;
    }
    //getters and setters
}
```

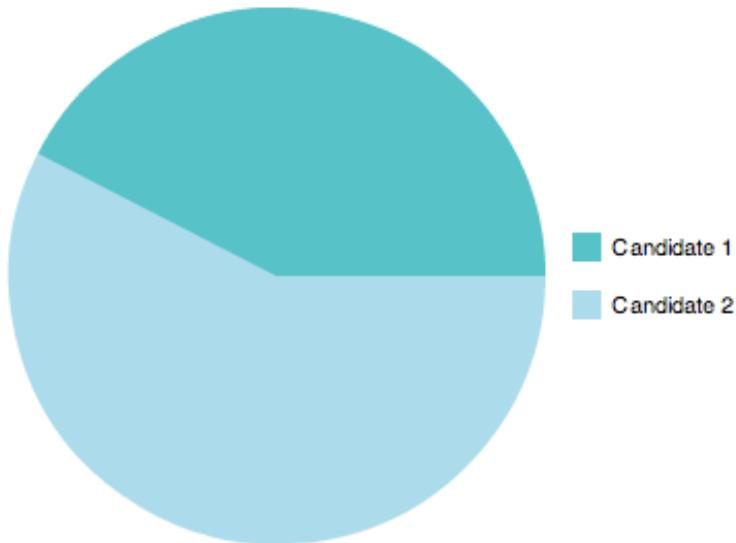
Next step is to provide the data;

```
public class ChartBean implements Serializable {
    private List<Vote> votes;
    public ChartBean() {
        votes = new ArrayList<Vote>();
        votes.add(new Vote("Candidate 1", 100));
        votes.add(new Vote("Candidate 2", 100));
    }
    public List<Vote> getVotes() {
        int random1 = (int)(Math.random() * 1000);
        int random2 = (int)(Math.random() * 1000);
        votes.get(0).add(random1);
        votes.get(1).add(random2);
        return votes;
    }
}
```

For displaying the voting, we'll be using a pie chart as follows;

```
<p:pieChart id="votes" value="#{chartBean.votes}" var="vote"
             live="true" refreshInterval="5000"
             categoryField="#{vote.candidate}"
             dataField="#{vote.count}" />
```

This live piechart is almost same as a static pie chart, except *live* attribute is set to true. When a chart is live, the collection bind to the value is read periodically in a specified interval. In this example, `getVotes()` would be called continuously in 5 seconds interval. Polling interval is tuned using the *refreshInterval* attribute which is set to 3000 milliseconds.



### 3.11.10 Interactive Charts

Charts are interactive components, they can respond to events like series item selection. When a series item is clicked an ajax request can be sent to the server notifying an itemSelectListener by passing an itemSelectEvent. ItemSelectEvent contains useful information about the selected item like series index and item index.

Chart components also use PrimeFaces Partial Page Rendering mechanism so using the update attribute, it's possible to refresh other components on the page. In the example below, message outputText is refreshed with the message provided in itemSelectListener.

```
<p:pieChart id="votes" value="#{chartBean.votes}" var="vote"
            itemSelectListener="#{chartBean.itemSelect}"
            update="msgs"
            categoryField="#{vote.candidate}"
            dataField="#{vote.count}" />

<p:messages id="msgs" value="#{chartBean.message}" />
```

```
public class ChartBean implements Serializable {

    //Data creation omitted

    public void itemSelect(ItemSelectEvent event) {
        FacesMessage msg = new FacesMessage();
        msg.setSummary("Item Index: " + event.getItemIndex());
        msg.setDetail("Series Index:" + event.getSeriesIndex());

        FacesContext.getCurrentInstance().addMessage(null, msg);
    }
}
```

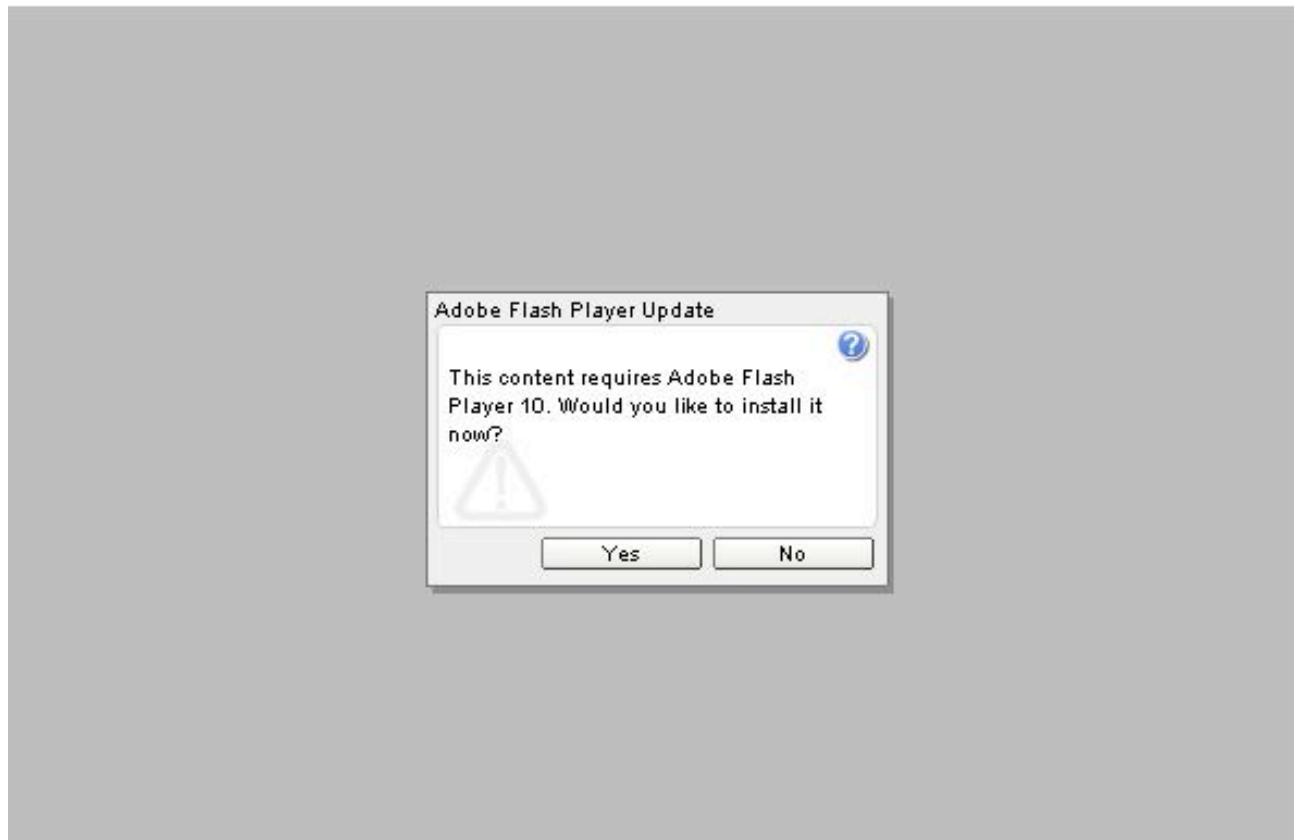
### 3.11.11 Charting Tips

#### Flash Version

Chart components require flash player version 9.0.45 or higher.

#### Express Install

In case the users of your application use an older unsupported version of flash player, chart components will automatically prompt to install or update users' flash players. The screen would look like this for these users.



#### Non Flash

If you like to use static image charts instead of flash based charts, see the JFreeChart integration example at `graphicImage` section. Static images charts are not rich as flash charts but guaranteed to work at environments with no flash installed.

#### Declarative vs Programmatic

Charts can be created in two ways, declarative way is deprecated and new programmatic way via a `ChartModel` is the suggested one.

#### Live and Interactive Charts

Live or an Interactive Chart must be nested inside a form.

## 3.12 Collector

Collector is a simple utility component to manage collections declaratively.

### Info

Tag	<b>collector</b>
Tag Class	<b>org.primefaces.component.collector.CollectorTag</b>
ActionListener Class	<b>org.primefaces.component.collector.Collector</b>

### Attributes

Name	Default	Type	Description
value	null	Object	Value to be used in collection operation
addTo	null	java.util.Collection	Reference to the Collection instance
removeFrom	null	java.util.Collection	Reference to the Collection instance

### Getting started with Collector

Collector requires a collection and a value to work with. It's important to override equals and hashCode methods of the value object to make collector work.

```
public class CreateBookBean {

    private Book book = new Book();

    private List<Book> books;

    public CreateBookBean() {
        books = new ArrayList<Book>();
    }

    public String createNew() {
        book = new Book(); //reset form

        return null;
    }

    //getters and setters
}
```

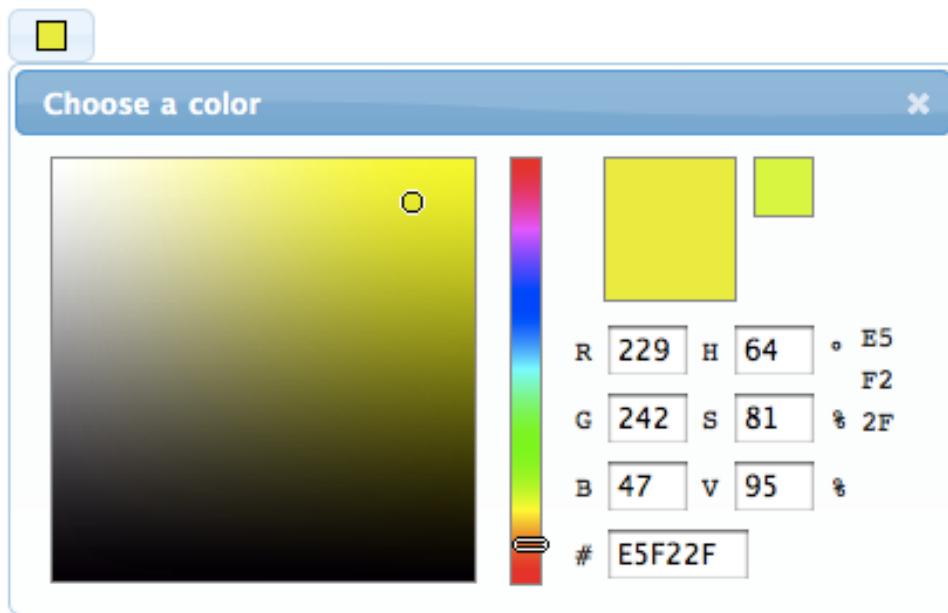
*value* attribute is required and sets the object to be added or removed to/from a collection.

```
<p:commandButton value="Add" action="#{createBookBean.createNew}">
    <p:collector value="#{createBookBean.book}"
        addTo="#{createBookBean.books}" />
</p:commandButton>
```

```
<p:commandLink value="Remove">
    <p value="#{book}" removeFrom="#{createBookBean.books}" />
</p:commandLink>
```

## 3.13 Color Picker

ColorPicker is an input component with a color palette.



### Info

Tag	<code>colorPicker</code>
Component Class	<code>org.primefaces.component.colorpicker.ColorPicker</code>
Component Type	<code>org.primefaces.component.ColorPicker</code>
Component Family	<code>org.primefaces.component</code>
Renderer Type	<code>org.primefaces.component.ColorPickerRenderer</code>
Renderer Class	<code>org.primefaces.component.colorpicker.ColorPickerRenderer</code>

### Attributes

Name	Default	Type	Description
<code>id</code>	<code>null</code>	<code>String</code>	Unique identifier of the component
<code>rendered</code>	<code>TRUE</code>	<code>Boolean</code>	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
<code>binding</code>	<code>null</code>	<code>Object</code>	An el expression that maps to a server side UIComponent instance in a backing bean

Name	Default	Type	Description
value	null	Object	Value of the component.
converter	null	Converter/String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id
immediate	FALSE	Boolean	When set true, process validations logic is executed at apply request values phase for this component.
required	FALSE	Boolean	Marks component as required.
validator	null	MethodExpr	A method expression that refers to a method for validation the input.
valueChangeListener	null	ValueChangeListener	A method binding expression that refers to a method for handling a valuchangeevent.
requiredMessage	null	String	Message to be displayed when required field validation fails.
converterMessage	null	String	Message to be displayed when conversion fails.
validatorMessage	null	String	Message to be displayed when validation fields.
widgetVar	null	String	Name of the client side widget.
header	Choose a color	String	Header text for the color picker title.
showControls	TRUE	String	Sets visibility of whole set of controls.
showHexControls	TRUE	String	Sets visibility of hex controls.
showHexSummary	TRUE	String	Sets visibility of hex summary.
showHsvControls	FALSE	String	Sets visibility of hsv controls.
showRGBControls	TRUE	String	Sets visibility of rgb controls.
showWebSafe	TRUE	String	Sets visibility of web safe controls.

## Getting started with ColorPicker

ColorPicker requires a `java.awt.Color` reference.

```
import java.awt.Color;

public class ColorPickerController {

    private Color selectedColor;

    //getter and setter
}
```

```
<p:colorPicker value="#{colorBean.color}" />
```

## Converter

In case you don't prefer to use `java.awt.Color`, you can plug your custom converter. Following example uses a converter to define colors in RGB string format such as '250, 214, 255'.

```
<p:colorPicker value="#{colorBean.color}">
    <f:converter converterId="colorPickerConverer" />
</p:colorPicker>
```

```
public class ColorPickerConverter implements Converter {

    public Object getAsObject(FacesContext facesContext, UIComponent component, String submittedValue) {
        return submittedValue; //just return the rgb value as string
    }

    public String getAsString(FacesContext facesContext, UIComponent component, Object value) {
        //value is a comma separated string in "R,G,B" format.
        return value == null ? null : value.toString();
    }
}
```

```
public class ColorBean {

    private String selectedColor;

    //getter and setter
}
```

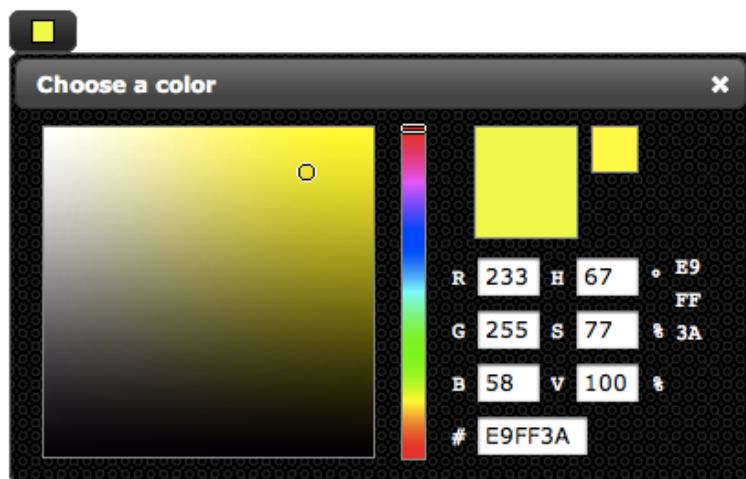
## Client Side API

Color Picker is based on a YUI widget, see link below to access client side API documentation.

[http://developer.yahoo.com/yui/docs/module\\_colorpicker.html](http://developer.yahoo.com/yui/docs/module_colorpicker.html)

## Skinning

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.14 Column

Column is an extended version of the standard column used by various PrimeFaces components like datatable, treetable and more.

### Info

Tag	<b>column</b>
Component Class	<b>org.primefaces.component.column.Column</b>
Component Type	<b>org.primefaces.component.Column</b>
Component Family	<b>org.primefaces.component</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
style	null	String	Inline style of the column.
styleClass	null	String	Style class of the column.
sortBy	null	ValueExpr	Property to be used for sorting.
sortFunction	null	String/MethodExpr	Custom pluggable sortFunction.
filterBy	null	ValueExpr	Property to be used for filtering.
filterEvent	keyup	String	Dom event to trigger a filter request
filterStyle	null	String	Inline style of the filter element
filterStyleClass	null	String	Style class of the filter element
filterOptions	null	Object	A collection of selectitems for filter dropdown.
filterMatchMode	startsWith	String	Match mode for filtering.
rowspan	1	Integer	Defines the number of rows the column spans.
colspan	1	Integer	Defines the number of columns the column spans.
headerText	null	String	Shortcut for header facet.
footerText	null	String	Shortcut for footer facet.

Name	Default	Type	Description
selectionMode	null	String	Enables selection mode.

## Note

Not all attributes of column are utilized by the components that use column.

## Getting Started with Column

As column is a reused component, see documentation of components that use a column.

## 3.15 Columns

Columns is used by datatable to create columns programmatically.

### Info

Tag	<b>columns</b>
Component Class	<b>org.primefaces.component.column.Columns</b>
Component Type	<b>org.primefaces.component.Columns</b>
Component Family	<b>org.primefaces.component</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Data to represent columns.
var	null	String	Request scoped iterator to access a column.
columnIndexVar	null	String	Request scoped iterator to access a column index.

### Getting Started with Columns

See dynamic columns section in datatable documentation for detailed information.

## 3.16 ColumnGroup

ColumnGroup is used by datatable for grouping.

### Info

Tag	<b>columnGroup</b>
Component Class	<b>org.primefaces.component.columngroup.ColumnGroup</b>
Component Type	<b>org.primefaces.component. ColumnGroup</b>
Component Family	<b>org.primefaces.component</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
type	null	String	Type of group, valid values are <i>header</i> and <i>footer</i> .

### Getting Started with ColumnGroup

See grouping section in datatable documentation for detailed information.

## 3.17 CommandButton

CommandButton is an extended version of standard JSF commandButton with ajax and skinning features.



### Info

Tag	<b>commandButton</b>
Component Class	<b>org.primefaces.component.commandbutton.CommandButton</b>
Component Type	<b>org.primefaces.component.CommandButton</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.CommandButtonRenderer</b>
Renderer Class	<b>org.primefaces.component.commandbutton.CommandButtonRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	String	Label for the button
action	null	MethodExpr/String	A method expression or a String outcome that'd be processed when button is clicked.
actionListener	null	MethodExpr	An actionlistener that'd be processed when button is clicked.
immediate	FALSE	Boolean	Boolean value that determines the phaseId, when true actions are processed at apply_request_values, when false at invoke_application phase.
type	submit	String	Sets the behavior of the button.
async	FALSE	Boolean	When set to true, ajax requests are not queued.
process	null	String	Component(s) to process partially instead of whole view.

Name	Default	Type	Description
ajax	TRUE	Boolean	Specifies the submit mode, when set to true (default), submit would be made with Ajax.
update	null	String	Component(s) to be updated with ajax.
onstart	null	String	Client side callback to execute before ajax request is begins.
oncomplete	null	String	Client side callback to execute when ajax request is completed.
onsuccess	null	String	Client side callback to execute when ajax request succeeds.
onerror	null	String	Client side callback to execute when ajax request fails.
global	TRUE	Boolean	Defines whether to trigger ajaxStatus or not.
style	null	String	Inline style of the button element.
styleClass	null	String	StyleClass of the button element.
onblur	null	String	Client side callback to execute when button loses focus.
onchange	null	String	Client side callback to execute when button loses focus and its value has been modified since gaining focus.
onclick	null	String	Client side callback to execute when button is clicked.
ondblclick	null	String	Client side callback to execute when button is double clicked.
onfocus	null	String	Client side callback to execute when button receives focus.
onkeydown	null	String	Client side callback to execute when a key is pressed down over button.
onkeypress	null	String	Client side callback to execute when a key is pressed and released over button.
onkeyup	null	String	Client side callback to execute when a key is released over button.
onmousedown	null	String	Client side callback to execute when a pointer button is pressed down over button.
onmousemove	null	String	Client side callback to execute when a pointer button is moved within button.
onmouseout	null	String	Client side callback to execute when a pointer button is moved away from button.
onmouseover	null	String	Client side callback to execute when a pointer button is moved onto button.

Name	Default	Type	Description
onmouseup	null	String	Client side callback to execute when a pointer button is released over button.
onselect	null	String	Client side callback to execute when text within button is selected by user.
accesskey	null	String	Access key that when pressed transfers focus to the button.
alt	null	String	Alternate textual description of the button.
dir	null	String	Direction indication for text that does not inherit directionality. Valid values are LTR and RTL.
disabled	FALSE	Boolean	Disables the button.
image	null	String	Style class representing the button icon.
label	null	String	A localized user presentable name.
lang	null	String	Code describing the language used in the generated markup for this component.
tabindex	null	Integer	Position of the button element in the tabbing order.
title	null	String	Advisory tooltip information.
readonly	FALSE	Boolean	Flag indicating that this component will prevent changes by the user.
widgetVar	null	String	Name of the client side widget.

## Getting started with CommandButton

CommandButton usage is similar to standard h:commandButton, by default commandButton submits its enclosing form with ajax.

```
<p:commandButton value="Save" actionListener="#{bookBean.saveBook}" />
```

```
public class BookBean {

    public void saveBook() {
        //Save book
    }
}
```

## Reset Buttons

Reset buttons do not submit the form, just resets the form contents.

```
<p:commandButton type="reset" value="Reset" />
```

## Push Buttons

Push buttons are used to execute custom javascript without causing an ajax/non-ajax request. To create a push button set type as "button".

```
<p:commandButton type="button" value="Alert" onclick="alert('Prime')"/>
```

## AJAX and Non-AJAX

CommandButton has built-in ajax capabilities, ajax submit is enabled by default and configured using *ajax* attribute. When ajax attribute is set to false, form is submitted with a regular full page refresh.

The *update* attribute is used to partially update other component(s) after the ajax response is received. Update attribute takes a comma or white-space separated list of JSF component ids to be updated. Basically any JSF component, not just primefaces components should be updated with the Ajax response.

In the following example, form is submitted with ajax and *display* outputText is updated with the ajax response.

```
<h:form>
    <h:inputText value="#{bean.text}" />
    <p:commandButton value="Submit" update="display"/>
    <h:outputText value="#{bean.text}" id="display" />
</h:form>
```

**Tip:** You can use the *ajaxStatus* component to notify users about the ajax request.

## Icons

An icon on a button is provided using CSS and *image* attribute.

```
<p:commandButton value="With Icon" image="disk"/>
<p:commandButton image="disk"/>
```



.disk is a simple css class with a background property;

```
.disk {
    background-image: url('disk.png') !important;
}
```

## Client Side API

Widget: *PrimeFaces.widget.CommandButton*

Method	Params	Return Type	Description
disable()	-	void	Disables button
enable()	-	void	Enables button

## Skinning

CommandButton renders a *button* tag which *style* and *styleClass* applies.

Following is the list of structural style classes;

Style Class	Applies
.ui-button	Button element
.ui-button-text-only	Button element when icon is not used
.ui-button-text	Label of button

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;

## 3.18 CommandLink

CommandLink extends standard JSF commandLink with Ajax capabilities.

### Info

Tag	<b>commandLink</b>
Component Class	<b>org.primefaces.component.commandlink.CommandLink</b>
Component Type	<b>org.primefaces.component.CommandLink</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.CommandLinkRenderer</b>
Renderer Class	<b>org.primefaces.component.commandlink.CommandLinkRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	String	Href value of the rendered anchor.
action	null	MethodExpr/String	A method expression or a String outcome that'd be processed when link is clicked.
actionListener	null	MethodExpr	An actionlistener that'd be processed when link is clicked.
immediate	FALSE	Boolean	Boolean value that determines the phaseId, when true actions are processed at apply_request_values, when false at invoke_application phase.
async	FALSE	Boolean	When set to true, ajax requests are not queued.
process	null	String	Component(s) to process partially instead of whole view.
ajax	TRUE	Boolean	Specifies the submit mode, when set to true (default), submit would be made with Ajax.
update	null	String	Component(s) to be updated with ajax.
onstart	null	String	Client side callback to execute before ajax request is begins.

Name	Default	Type	Description
oncomplete	null	String	Client side callback to execute when ajax request is completed.
onsuccess	null	String	Client side callback to execute when ajax request succeeds.
onerror	null	String	Client side callback to execute when ajax request fails.
global	TRUE	Boolean	Defines whether to trigger ajaxStatus or not.
style	null	String	Style to be applied on the anchor element
styleClass	null	String	StyleClass to be applied on the anchor element
onblur	null	String	Client side callback to execute when link loses focus.
onclick	null	String	Client side callback to execute when link is clicked.
ondblclick	null	String	Client side callback to execute when link is double clicked.
onfocus	null	String	Client side callback to execute when link receives focus.
onkeydown	null	String	Client side callback to execute when a key is pressed down over link.
onkeypress	null	String	Client side callback to execute when a key is pressed and released over link.
onkeyup	null	String	Client side callback to execute when a key is released over link.
onmousedown	null	String	Client side callback to execute when a pointer button is pressed down over link.
onmousemove	null	String	Client side callback to execute when a pointer button is moved within link.
onmouseout	null	String	Client side callback to execute when a pointer button is moved away from link.
onmouseover	null	String	Client side callback to execute when a pointer button is moved onto link.
onmouseup	null	String	Client side callback to execute when a pointer button is released over link.
accesskey	null	String	Access key that when pressed transfers focus to the link.
charset	null	String	Character encoding of the resource designated by this hyperlink.
coords	null	String	Position and shape of the hot spot on the screen for client use in image maps.

Name	Default	Type	Description
dir	null	String	Direction indication for text that does not inherit directionality. Valid values are LTR and RTL.
disabled	null	Boolean	Disables the link
hreflang	null	String	Language code of the resource designated by the link.
rel	null	String	Relationship from the current document to the anchor specified by the link, values are provided by a space-separated list of link types.
rev	null	String	A reverse link from the anchor specified by this link to the current document, values are provided by a space-separated list of link types.
shape	null	String	Shape of hot spot on the screen, valid values are default, rect, circle and poly.
tabindex	null	Integer	Position of the button element in the tabbing order.
target	null	String	Name of a frame where the resource targeted by this link will be displayed.
title	null	String	Advisory tooltip information.
type	null	String	Type of resource referenced by the link.

## Getting Started with CommandLink

CommandLink is used just like the standard h:commandLink, difference is form is submitted with ajax by default.

```
public class BookBean {

    public void saveBook() {
        //Save book
    }
}
```

```
<p:commandLink actionListener="#{bookBean.saveBook}" update="text">
    <h:outputText value="Save" />
</p:commandLink>
```

## Skinning

CommandLink renders an html anchor element that *style* and *styleClass* attributes apply.

## 3.19 ConfirmDialog

ConfirmDialog is a replacement to the legacy javascript confirmation box. Skinning, customization and avoiding popup blockers are notable advantages over classic javascript confirmation.



### Info

Tag	<code>confirmDialog</code>
Component Class	<code>org.primefaces.component.confirmdialog.ConfirmDialog</code>
Component Type	<code>org.primefaces.component.ConfirmDialog</code>
Component Family	<code>org.primefaces.component</code>
Renderer Type	<code>org.primefaces.component.ConfirmDialogRenderer</code>
Renderer Class	<code>org.primefaces.component.confirmdialog.ConfirmDialogRenderer</code>

### Attributes

Name	Default	Type	Description
<code>id</code>	null	String	Unique identifier of the component
<code>rendered</code>	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
<code>binding</code>	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
<code>widgetVar</code>	null	String	Name of the client side widget.
<code>message</code>	null	String	Text to be displayed in body.
<code>header</code>	null	String	Text for the header.
<code>severity</code>	null	String	Message severity for the displayed icon.
<code>draggable</code>	TRUE	Boolean	Controls draggability

Name	Default	Type	Description
modal	FALSE	Boolean	Boolean value that specifies whether the document should be shielded with a partially transparent mask to require the user to close the Panel before being able to activate any elements in the document.
width	300	Integer	Width of the dialog in pixels
height	auto	Integer	Width of the dialog in pixels
zindex	1000	Integer	zindex property to control overlapping with other elements.
styleClass	null	String	Style class of the dialog container
showEffect	null	String	Effect to use when showing the dialog
hideEffect	null	String	Effect to use when hiding the dialog
position	null	String	Defines where the dialog should be displayed
closeOnEscape	TRUE	Boolean	Defines if dialog should be closed when escape key is pressed.
closable	TRUE	Boolean	Defines if close icon should be displayed or not
appendToBody	FALSE	Boolean	Appends dialog as a child of document body.

## Getting started with ConfirmDialog

ConfirmDialog has a simple client side api, `show()` and `hide()` functions are used to display and close the dialog respectively. You can call these functions to display a confirmation from any component like commandButton, commandLink, menuitem and more.

```
<h:form>
    <p:commandButton type="button" onclick="cd.show()" />

    <p:confirmDialog message="Are you sure about destroying the world?"
        header="Initiating destroy process" severity="alert"
        widgetVar="cd">

        <p:commandButton value="Yes Sure" actionListener="#{buttonBean.destroyWorld}"
            update="messages" oncomplete="confirmation.hide()"/>

        <p:commandButton value="Not Yet" onclick="confirmation.hide(); type="button" />

    </p:confirmDialog>
</h:form>
```

## Message

Message can be defined in two ways, either via message option or message facet. Message facet is useful if you need to place custom content instead of simple text.

```
<p:confirmDialog widgetVar="cd">
    <f:facet name="message">
        <h:outputText value="Are you sure?" />
    </f:facet>

    //...
</p:confirmDialog>
```

## Effects

There are various effect options to be used when displaying and closing the dialog. Use *showEffect* and *hideEffect* options to apply these effects;

- blind
- bounce
- clip
- drop
- explode
- fade
- fold
- highlight
- puff
- pulsate
- scale
- shake
- size
- slide
- transfer

```
<p:confirmDialog showEffect="fade" hideEffect="explode" ...>
    //...
</p:confirmDialog>
```

## Severity

Severity defines the icon to display next to the message, default severity is *alert* and the other option is *info*.

## Position

By default dialog is positioned at center of the viewport and *position* option is used to change the location of the dialog. Possible values are;

- Single string value like '*center*', '*left*', '*right*', '*top*', '*bottom*' representing the position within viewport.
- Comma separated x and y coordinate values like *200, 500*
- Comma separated position values like '*top*', '*right*'. (Use single quotes when using a combination)

Some examples are described below;

```
<p:confirmDialog position="top" ...>
```

```
<p:confirmDialog position="'right','top'" ...>
```

```
<p:confirmDialog position="200,500" ...>
```

## Client Side API

Widget: *PrimeFaces.widget.ConfirmDialog*

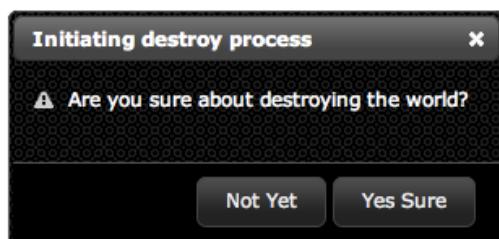
Method	Params	Return Type	Description
show()	-	void	Displays dialog.
hide()	-	void	Closes dialog.

## Skinning

ConfirmDialog resides in a main container element which the *styleClass* option apply. Following is the list of structural style classes;

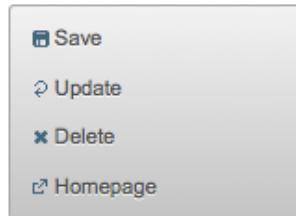
Style Class	Applies
.ui-dialog	Container element of dialog
.ui-dialog-titlebar	Title bar
.ui-dialog-title-dialog	Header text
.ui-dialog-titlebar-close	Close icon
.ui-dialog-content	Dialog body
.ui-dialog-buttonpane	Footer button panel

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.20 ContextMenu

ContextMenu provides an overlay menu displayed on mouse right-click event.



### Info

Tag	<code>contextMenu</code>
Component Class	<code>org.primefaces.component.contextmenu.ContextMenu</code>
Component Type	<code>org.primefaces.component.ContextMenu</code>
Component Family	<code>org.primefaces.component</code>
Renderer Type	<code>org.primefaces.component.ContextMenuRenderer</code>
Renderer Class	<code>org.primefaces.component.contextmenu.ContextMenuRenderer</code>

### Attributes

Name	Default	Type	Description
<code>id</code>	null	String	Unique identifier of the component
<code>rendered</code>	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
<code>binding</code>	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
<code>widgetVar</code>	null	String	Name of the client side widget.
<code>for</code>	null	String	Id of the component to attach to
<code>style</code>	null	String	Style of the main container element
<code>styleClass</code>	null	String	Style class of the main container element
<code>zindex</code>	null	Integer	zindex property to control overlapping with other elements.
<code>effect</code>	fade	String	Sets the effect for the menu display.
<code>effectDuration</code>	400	Integer	Sets the effect duration in milliseconds.
<code>model</code>	null	MenuModel	Menu model instance to create menu programmatically.

## Getting started with ContextMenu

Just like any other PrimeFaces menu component, contextMenu is created with menuitems. Optional for attribute defines which component the contextMenu is attached to. When for is not defined, contextMenu is attached to the page meaning, right-click on anywhere on page will display the menu.

```
<p:contextMenu>
    <p:menuItem value="Save" actionListener="#{bean.save}" update="msg"/>
    <p:menuItem value="Delete" actionListener="#{bean.delete}" ajax="false"/>
    <p:menuItem value="Go Home" url="www.primefaces.org" target="_blank"/>
</p:contextMenu>
```

ContextMenu example above is attached to the whole page and consists of three different menuitems with different use cases. First menuItem triggers an ajax action, second one triggers a non-ajax action and third one is used for navigation.

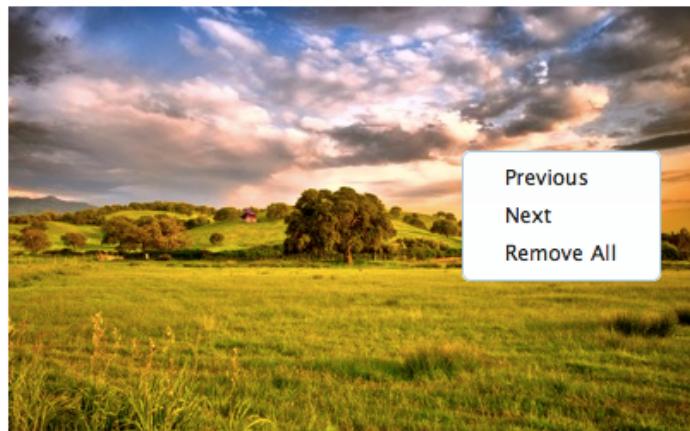
## Attachment

ContextMenu can be attached to any JSF component, this means right clicking only on the attached component will display the contextMenu. Following example demonstrates an integration between contextMenu and imageSwitcher, contextMenu here is used to navigate between images.

```
<p:imageSwitch id="images" widgetVar="gallery" slideshowAuto="false">
    <p:graphicImage value="/images/nature1.jpg" />
    <p:graphicImage value="/images/nature2.jpg" />
    <p:graphicImage value="/images/nature3.jpg" />
    <p:graphicImage value="/images/nature4.jpg" />
</p:imageSwitch>

<p:contextMenu for="images">
    <p:menuItem value="Previous" url="#" onclick="gallery.previous()" />
    <p:menuItem value="Next" url="#" onclick="gallery.next()" />
</p:contextMenu>
```

Now right-clicking anywhere on an image will display the contextMenu like;



## Data Components

Data components like datatable, datagrid and tree needs special treatment, special integration with contextMenu and these components will be available in PrimeFaces 3.0.

## Effects

ContextMenu has a built-in animation to use when displaying&hiding itself. This animation is customizable using attributes like *effect* and *effectDuration*. Available animations are *fade* or *slide*, effectDuration is defined in milliseconds defaulting to 400.

## Dynamic Menus

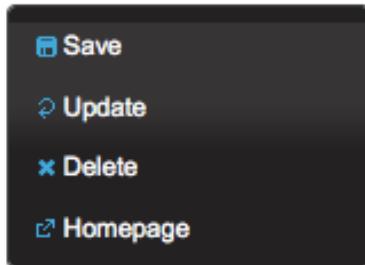
ContextMenus can be created programmatically as well, see the dynamic menus part in menu component section for more information and an example.

## Skinning

ContextMenu resides in a main container which *style* and *styleClass* attributes apply. Following is the list of structural style classes;

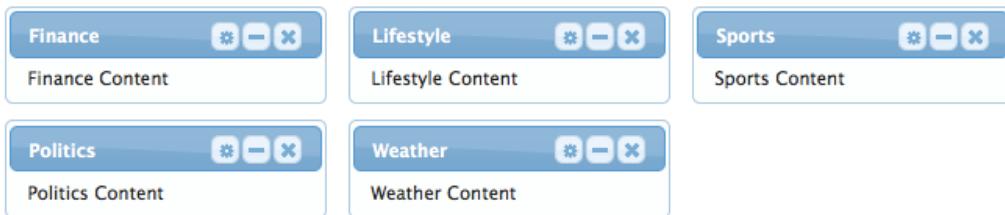
Style Class	Applies
.wijmo-wijmenu	Container element of menu
.wijmo-wijmenu-list	List container
.wijmo-wijmenu-item	Each menu item
.wijmo-wijmenu-link	Anchor element in a link item
.wijmo-wijmenu-text	Text element in an item

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.21 Dashboard

Dashboard provides a portal like layout with drag&drop based reorder capabilities.



### Info

Tag	<b>dashboard</b>
Component Class	<b>org.primefaces.component.dashboard.Dashboard</b>
Component Type	<b>org.primefaces.component.Dashboard</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.DashboardRenderer</b>
Renderer Class	<b>org.primefaces.component.dashboard.DashboardRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
widgetVar	null	String	Name of the client side widget
model	null	String	Dashboard model instance representing the layout of the UI.
disabled	FALSE	Boolean	Disables reordering
onReorderUpdate	null	String	Component(s) to update after ajax reorder event is processed.
reorderListener	null	MethodExpression	A server side listener to invoke when widgets are reordered
style	null	String	Inline style of the dashboard container
styleClass	null	String	Style class of the dashboard container

## Getting started with Dashboard

Dashboard is backed by a DashboardModel and consists of panel components.

```
<p:dashboard model="#{dashboardBean.model}">
    <p:panel id="sports">
        //Sports Content
    </p:panel>
    <p:panel id="finance">
        //Finance Content
    </p:panel>

    //more panels like lifestyle, weather, politics...
</p:dashboard>
```

Dashboard model simply defines the number of columns and the widgets to be placed in each column. See the end of this section for the detailed Dashboard API.

```
public class DashboardBean {

    private DashboardModel model;

    public DashboardBean() {
        model = new DefaultDashboardModel();
        DashboardColumn column1 = new DefaultDashboardColumn();
        DashboardColumn column2 = new DefaultDashboardColumn();
        DashboardColumn column3 = new DefaultDashboardColumn();

        column1.addWidget("sports");
        column1.addWidget("finance");
        column2.addWidget("lifestyle");
        column2.addWidget("weather");
        column3.addWidget("politics");

        model.addColumn(column1);
        model.addColumn(column2);
        model.addColumn(column3);
    }
}
```

## State

Dashboard is a stateful component, whenever a widget is reordered dashboard model will be updated, by persisting this information, you can easily create a stateful dashboard so if your application allows users to change the layout, next time a user logs in you can present the dashboard layout based on the user preferences.

## Reorder Listener

As most of other PrimeFaces components, dashboard provides flexible callbacks for page authors to invoke custom logic. Ajax reorderListener is one of them, optionally you can update a certain part of your page with onReorderUpdate option.

```
<p:dashboard model="#{dashboardBean.model}"
    reorderListener="#{dashboardBean.handleReorder}"
    onReorderUpdate="messages">

    //panels

</p:dashboard>

<p:growl id="messages" />
```

This dashboard displays a facesmessage added at reorderlistener using growl.

```
public class DashboardBean {

    ...

    public void handleReorder(DashboardReorderEvent event) {
        String widgetId = event.getWidgetId();
        int widgetIndex = event.getItemIndex();
        int columnIndex = event.getColumnIndex();
        int senderColumnIndex = event.getSenderColumnIndex();

        //Add facesmessage
    }
}
```

If a widget is reordered in the same column, senderColumnIndex will be null. This field is populated only when widget is moved to a column from another column. Also when reorderListener is invoked, dashboard has already updated it's model, reorderListener is useful for custom logic like persisting the model.

**Note:** At least one form needs to be present on page to use ajax reorderListener.

## Disabling Dashboard

If you'd like to disable reordering, set *disabled* option to true.

```
<p:dashboard disabled="true" ...>
    //panels
</p:dashboard>
```

## Toggle, Close and Options Menu

Widgets presented in dashboard can be closable, toggleable and have options menu as well, dashboard doesn't implement these by itself as these features are already provided by the panel component. See panel component section for more information.

```
<p:dashboard model="#{dashboardBean.model}">
    <p:panel id="sports" closable="true" toggleable="true">
        //Sports Content
    </p:panel>
</p:dashboard>
```

## New Widgets

Draggable component is used to add new widgets to the dashboard. This way you can add new panels from outside of the dashboard.

```
<p:dashboard model="#{dashboardBean.model}" id="board">
    //panels
</p:dashboard>

<p:panel id="newwidget" />

<p:draggable for="newwidget" helper="clone" dashboard="board" />
```

## Skinning

Dashboard resides in a container element which style and styleClass options apply. Following is the list of structural style classes;

Style Class	Applies
.ui-dashboard	Container element of dashboard
.ui-dashboard-column	Each column in dashboard
div.ui-state-hover	Placeholder

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## Tips

- Provide a column width using *ui-dashboard-column* style class otherwise empty columns might not receive new widgets.

## Dashboard Model API

*org.primefaces.model.DashboardModel* (*org.primefaces.model.map.DefaultDashboardModel* is the default implementation)

Method	Description
void addColumn(DashboardColumn column)	Adds a column to the dashboard
List<DashboardColumn> getColumns()	Returns all columns in dashboard
int getColumnCount()	Returns the number of columns in dashboard
DashboardColumn getColumn(int index)	Returns the dashboard column at given index
void transferWidget(DashboardColumn from, DashboardColumn to, String widgetId, int index)	Relocates the widget identified with widget id to the given index of the new column from old column.

*org.primefaces.model.DashboardColumn* (*org.primefaces.model.map.DefaultDashboardModel* is the default implementation)

Method	Description
void removeWidget(String widgetId)	Removes the widget with the given id
List<String> getWidgets()	Returns the ids of widgets in column
int getWidgetCount()	Returns the count of widgets in column
String getWidget(int index)	Returns the widget id with the given index
void addWidget(String widgetId)	Adds a new widget with the given id
void addWidget(int index, String widgetId)	Adds a new widget at given index
void reorderWidget(int index, String widgetId)	Updates the index of widget in column

## 3.22 DataExporter

DataExporter is handy for exporting data listed using a Primefaces Datatable to various formats such as excel, pdf, csv and xml.

### Info

Tag	<b>dataExporter</b>
Tag Class	<b>org.primefaces.component.export.DataExporterTag</b>
ActionListener Class	<b>org.primefaces.component.export.DataExporter</b>

### Attributes

Name	Default	Type	Description
type	null	String	Export type: "xls", "pdf", "csv", "xml"
target	null	String	Server side id of the datatable whose date would be exported
fileName	null	String	Filename of the generated export file, defaults to datatable server side id
excludeColumns	null	String	Comma separated list(if more than one) of column indexes to be excluded from export
pageOnly	FALSE	String	Exports only current page instead of whole dataset
encoding	UTF-8	Boolean	Character encoding to use
preProcessor	null	MethodExpr	PreProcessor for the exported document.
postProcessor	null	MethodExpr	PostProcessor for the exported document.

### Getting Started with DataExporter

DataExporter is nested in a UICommand component such as commandButton or commandLink. For pdf exporting **iText** and for xls exporting **poi** libraries are required in the classpath. Target must point to a PrimeFaces Datatable. Assume the table to be exported is defined as;

```
<p:DataTable id="tableId" ...>
    //columns
</p:DataTable>
```

### *Excel export*

```
<p:commandButton value="Export as Excel" ajax="false">
    <p:dataExporter type="xls" target="tableId" fileName="cars"/>
</p:commandButton>
```

### *PDF export*

```
<p:commandButton value="Export as PDF" ajax="false" >
    <p:dataExporter type="pdf" target="tableId" fileName="cars"/>
</p:commandButton>
```

### *CSV export*

```
<p:commandButton value="Export as CSV" ajax="false" >
    <p:dataExporter type="csv" target="tableId" fileName="cars"/>
</p:commandButton>
```

### *XML export*

```
<p:commandButton value="Export as XML" ajax="false" >
    <p:dataExporter type="xml" target="tableId" fileName="cars"/>
</p:commandLink>
```

## **PageOnly**

By default dataExporter works on whole dataset, if you'd like export only the data displayed on current page, set pageOnly to true.

```
<p:dataExporter type="pdf" target="tableId" fileName="cars" pageOnly="true"/>
```

## **Excluding Columns**

Usually datatable listings contain command components like buttons or links that need to be excluded from the exported data. For this purpose optional excludeColumns property is used to define the column indexes to be omitted during data export.

Exporter below ignores first column, to exclude more than one column define the indexes as a comma separated string (excludeColumns="0,2,6").

```
<p:dataExporter type="pdf" target="tableId" excludeColumns="0"/>
```

## Pre and Post Processors

In case you need to customize the exported document (add logo, caption ...), use the processor method expressions. PreProcessors are executed before the data is exported and PostProcessors are processed after data is included in the document. Processors are simple java methods taking the document as a parameter.

### *Change Excel Table Header*

First example of processors changes the background color of the exported excel's headers.

```
<h:commandButton value="Export as XLS">
    <p:dataExporter type="xls" target="tableId" fileName="cars"
        postProcessor="#{bean.postProcessXLS}"/>
</h:commandButton>
```

```
public void postProcessXLS(Object document) {
    HSSFWorkbook wb = (HSSFWorkbook) document;
    HSSFSheet sheet = wb.getSheetAt(0);
    HSSFRow header = sheet.getRow(0);
    HSSFCCellStyle cellStyle = wb.createCellStyle();
    cellStyle.setFillForegroundColor(HSSFColor.GREEN.index);
    cellStyle.setFillPattern(HSSFCCellStyle.SOLID_FOREGROUND);

    for(int i=0; i < header.getPhysicalNumberOfCells();i++) {
        header.getCell(i).setCellStyle(cellStyle);
    }
}
```

### *Add Logo to PDF*

This example adds a logo to the PDF before exporting begins.

```
<h:commandButton value="Export as PDF">
    <p:dataExporter type="pdf" target="tableId" fileName="cars"
        preProcessor="#{bean.preProcessPDF}"/>
</h:commandButton>
```

```
public void preProcessPDF(Object document) throws IOException,
    BadElementException, DocumentException {
    Document pdf = (Document) document;
    ServletContext servletContext = (ServletContext)
    FacesContext.getCurrentInstance().getExternalContext().getContext();
    String logo = servletContext.getRealPath("") + File.separator + "images" +
    File.separator + "prime_logo.png";
    pdf.add(Image.getInstance(logo));
}
```

## 3.23 DataGrid

DataGrid displays a collection of data in grid layout. Ajax Pagination is a built-in feature and paginator UI is fully customizable via various options like paginatorTemplate, rowPerPageOptions, pageLinks and more.



### Info

Tag	<b>dataGrid</b>
Component Class	<b>org.primefaces.component.datagrid.DataGrid</b>
Component Type	<b>org.primefaces.component.DataGrid</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.DataGridRenderer</b>
Renderer Class	<b>org.primefaces.component.datagrid.DataGridRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean

Name	Default	Type	Description
value	null	Object	Data to display.
var	null	String	Name of the request-scoped variable used to refer each data.
rows	null	Integer	Number of rows to display per page.
first	0	Integer	Index of the first row to be displayed
columns	3	Integer	Number of columns of grid.
widgetVar	null	String	Variable name of the javascript widget.
paginator	FALSE	boolean	Enables pagination.
paginatorTemplate	null	String	Template of the paginator.
rowsPerPageTemplate	null	String	Template of the rowsPerPage dropdown.
currentPageReportTemplate	null	String	Template of the currentPageReport UI.
pageLinks	10	Integer	Maximum number of page links to display.
paginatorPosition	both	String	Position of the paginator.
paginatorAlwaysVisible	TRUE	Boolean	Defines if paginator should be hidden if total data count is less than number of rows per page.
page	1	Integer	Index of the current page
effect	TRUE	Boolean	Displays a fade animation during pagination.
effectSpeed	normale	String	Speed of the pagination effect.
style	null	String	Inline style of the main container.
styleClass	Null	String	Style class of the main container.

## Getting started with the DataGrid

We will be using a list of cars to display throughout the datagrid examples.

```
public class Car {

    private String model;
    private int year;
    private String manufacturer;
    private String color;
    ...

}
```

The code for CarBean that would be used to bind the datagrid to the car list.

```
public class CarBean {

    private List<Car> cars;

    public CarBean() {
        cars = new ArrayList<Car>();
        cars.add(new Car("myModel", 2005, "ManufacturerX", "blue"));
        //add more cars
    }

    public List<Car> getCars() {
        return cars;
    }
}
```

```
<p:datagrid var="car" value="#{carBean.cars}" columns="3" rows="12">

    <p:column>
        <p:panel header="#{car.model}">
            <h:panelGrid columns="1">
                <p:graphicImage value="/images/cars/#{car.manufacturer}.jpg"/>

                <h:outputText value="#{car.year}" />
            </h:panelGrid>
        </p:panel>
    </p:column>
</p:datagrid>
```

This datagrid has 3 columns and 12 rows. As datagrid extends from standard UIData, rows correspond to the number of data to display not the number of rows to render so the actual number of rows to render is rows/columns = 4. As a result datagrid is displayed as;

<b>5a0e3ce6</b>  1978	<b>c0a66869</b>  1991	<b>cd25ac27</b>  1991
<b>68d039c4</b>  1992	<b>0c2874f1</b>  1992	<b>0a32e04e</b>  2002
<b>518a6446</b>  2009	<b>be52e4d7</b>  1969	<b>6192c9e2</b>  1987
<b>c2e29105</b>  1992	<b>957c4405</b>  2008	<b>b3b3cbe8</b>  1983

## Ajax Pagination

DataGrid has a built-in paginator that is enabled by setting paginator option to true.

```
<p:dataGrid var="car" value="#{carBean.cars}" columns="3" rows="12"
    paginator="true">
    ...
</p:dataGrid>
```

## Paginator Template

Paginator is customized using paginatorTemplateOption that accepts various keys of UI controls.

- FirstPageLink
- LastPageLink
- PreviousPageLink
- NextPageLink
- PageLinks
- CurrentPageReport
- RowsPerPageDropDown

Note that {RowsPerPageDropDown} has it's own template, options to display is provided via rowsPerPageTemplate attribute (e.g. rowsPerPageTemplate="9,12,15").

Default UI is;

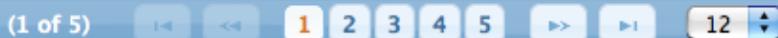


which corresponds to the following template.

```
"{FirstPageLink} {PreviousPageLink} {PageLinks} {NextPageLink} {LastPageLink}"
```

Here are more examples based on different templates;

```
" {CurrentPageReport} {FirstPageLink} {PreviousPageLink} {PageLinks} {NextPageLink}
 {LastPageLink} {RowsPerPageDropDown}"
```



```
" {PreviousPageLink} {CurrentPageReport} {NextPageLink}"
```



## Paginator Position

Paginator can be positioned using *paginatorPosition* attribute in three different locations, "top", "bottom" or "both" (default).

## Pagination Effect

A Fade animation is displayed during ajax paging, you can specify the speed of this animation using effectSpeed option or disable it at all by setting effect option to true.

```
<p:dataGrid var="car" value="#{carBean.cars}" columns="3" rows="12"
    paginator="true" effect="true" effectSpeed="fast">
    ...
</p:dataGrid>
```

## Selecting Data

Selection of data displayed in datagrid is very similar to row selection in datatable, you can access the current data using the var reference. Important point is to place datagrid contents in a p:column which is a child of datapgrid. Here is an example to demonstrate how to select data from datagrid and display withing a dialog.

```
<h:form id="carForm">
    <p:dataGrid var="car" value="#{carBean.cars}" columns="3" rows="12">
        <p:column>
            <p:panel header="#{car.model}">
                <p:commandLink update="carForm:display" oncomplete="dlg.show()">
                    <f:setPropertyActionListener value="#{car}" target="#{carBean.selectedCar}" />
                    <h:outputText value="#{car.model}" />
                </p:commandLink>
            </p:panel>
        </p:column>
    </p:dataGrid>

    <p:dialog modal="true" widgetVar="dlg">
        <h:panelGrid id="display" columns="2">
            <f:facet name="header">
                <p:graphicImage value="/images/cars/#{car.manufacturer}.jpg"/>
            </f:facet>
            <h:outputText value="Model:" />
            <h:outputText value="#{carBean.selectedCar.year}" />

            //more selectedCar properties
        </h:panelGrid>
    </p:dialog>
</h:form>
```

```
public class CarBean {

    private List<Car> cars;
    private Car selectedCar;

    //getters and setters
}
```

## Client Side API

Widget: *PrimeFaces.widget.DataGrid*

Method	Params	Return Type	Description
getPaginator()	-	Paginator	Returns the datagrid paginator instance.

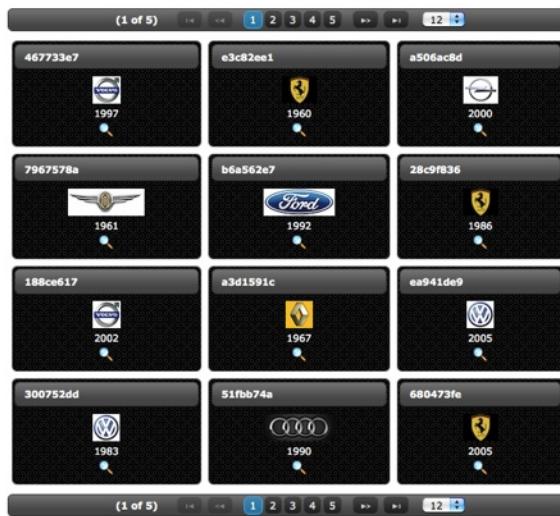
## Skinning

DataGrid resides in a main div container which style and styleClass attributes apply.

Following is the list of structural style classes;

Class	Applies
.ui-datagrid	Main container element
.ui-datagrid-content	Content container.
.ui-datagrid-data	Table element containing data
.ui-datagrid-row	A row in grid
.ui-datagrid-column	A column in grid

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;

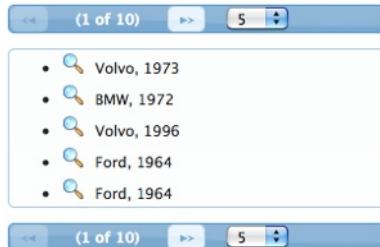


## Tips

- DataGrid uses a YUI paginator widget which you can retrieve via `widgetVar.getPaginator()`. It has handy client side methods like `getTotalRecords()`, `setPage` and more. For api documentation, see [http://developer.yahoo.com/yui/docs/module\\_paginator.html](http://developer.yahoo.com/yui/docs/module_paginator.html).

## 3.24 DataList

DataList presents a collection of data in list layout with several display types. Ajax Pagination is a built-in feature and paginator UI is fully customizable via various options like paginatorTemplate, rowsPerPageOptions, pageLinks and more.



### Info

Tag	<b>dataList</b>
Component Class	<b>org.primefaces.component.datalist.DataList</b>
Component Type	<b>org.primefaces.component.DataList.DataListTag</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.DataListRenderer</b>
Renderer Class	<b>org.primefaces.component.datalist.DataListRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Data to display.
var	null	String	Name of the request-scoped variable used to refer each data.
rows	null	Integer	Number of rows to display per page.
first	0	Integer	Index of the first row to be displayed
type	unordered	String	Type of the list, valid values are "unordered", "ordered" and "definition".
itemType	null	String	Specifies the list item type.

Name	Default	Type	Description
widgetVar	null	String	Variable name of the javascript widget.
paginator	FALSE	boolean	Enables pagination.
paginatorTemplate	null	String	Template of the paginator.
rowsPerPageTemplate	null	String	Template of the rowsPerPage dropdown.
currentPageReportTemplate	null	String	Template of the currentPageReport UI.
pageLinks	10	Integer	Maximum number of page links to display.
paginatorPosition	both	String	Position of the paginator.
paginatorAlwaysVisible	TRUE	Boolean	Defines if paginator should be hidden if total data count is less than number of rows per page.
page	1	Integer	Index of the current page
effect	TRUE	Boolean	Displays a fade animation during pagination.
effectSpeed	normale	String	Speed of the pagination effect.
style	null	String	Inline style of the main container.
styleClass	Null	String	Style class of the main container.

## Getting started with the DataList

We will be using the same Car and CarBean classes described in DataGrid section.

```
<p: dataList value="#{carBean.cars}" var="car" itemType="disc">
    #{car.manufacturer}, #{car.year}
</p: dataList>
```

Since DataList is a data iteration component, it renders its children for each data represented with *var* option. See itemType section for more information about the possible values.

## Ordered Lists

DataList displays the data in unordered format by default, if you'd like to use ordered display set type option to "ordered".

```
<p: dataList value="#{carBean.cars}" var="car" type="ordered">
    #{car.manufacturer}, #{car.year}
</p: dataList>
```

Output of this datalist would be;

1. Ferrari, 1960
2. Renault, 1985
3. Ford, 2003
4. Audi, 1976
5. Opel, 1983
6. Ferrari, 1974
7. Chrysler, 1980
8. Audi, 1980
9. Chrysler, 1983

## Item Type

*itemType* defines the bullet type of each item.

For ordered lists, following item types are available;

A	a	i
A. Ferrari, 1960 B. Renault, 1985 C. Ford, 2003 D. Audi, 1976 E. Opel, 1983 F. Ferrari, 1974 G. Chrysler, 1980 H. Audi, 1980 I. Chrysler, 1983	a. Ferrari, 1960 b. Renault, 1985 c. Ford, 2003 d. Audi, 1976 e. Opel, 1983 f. Ferrari, 1974 g. Chrysler, 1980 h. Audi, 1980 i. Chrysler, 1983	i. Ferrari, 1960 ii. Renault, 1985 iii. Ford, 2003 iv. Audi, 1976 v. Opel, 1983 vi. Ferrari, 1974 vii. Chrysler, 1980 viii. Audi, 1980 ix. Chrysler, 1983

And for unordered lists, available values are;

disc	circle	square
<ul style="list-style-type: none"> <li>• Opel, 1980</li> <li>• Chrysler, 1966</li> <li>• Volvo, 1962</li> <li>• Audi, 1990</li> <li>• Ford, 1972</li> <li>• Mercedes, 2003</li> <li>• BMW, 1984</li> <li>• Audi, 1975</li> <li>• Volvo, 1973</li> </ul>	<ul style="list-style-type: none"> <li>◦ Opel, 1980</li> <li>◦ Chrysler, 1966</li> <li>◦ Volvo, 1962</li> <li>◦ Audi, 1990</li> <li>◦ Ford, 1972</li> <li>◦ Mercedes, 2003</li> <li>◦ BMW, 1984</li> <li>◦ Audi, 1975</li> <li>◦ Volvo, 1973</li> </ul>	<ul style="list-style-type: none"> <li>■ Opel, 1980</li> <li>■ Chrysler, 1966</li> <li>■ Volvo, 1962</li> <li>■ Audi, 1990</li> <li>■ Ford, 1972</li> <li>■ Mercedes, 2003</li> <li>■ BMW, 1984</li> <li>■ Audi, 1975</li> <li>■ Volvo, 1973</li> </ul>

## Definition Lists

Third type of dataList is definition lists that display inline description for each item, to use definition list set *type* option to "*definition*".

Detail content is provided with the facet called "*description*".

```
<p: dataList value="#{carBean.cars}" var="car" type="definition">
    Model: #{car.model}, Year: #{car.year}
    <f: facet name="description">
        <p: graphicImage value="/images/cars/#{car.manufacturer}.jpg"/>
    </f: facet>
</p: dataList>
```



## Ajax Pagination

DataList has a built-in paginator that is enabled by setting paginator option to true.

```
<p: dataList value="#{carBean.cars}" var="car" paginator="true" rows="10">
    #{car.manufacturer}, #{car.year}
</p: dataList>
```

Pagination configuration and usage is same as dataGrid, see pagination section in dataGrid documentation for more information and examples.

## Selecting Data

Data selection can be implemented same as in dataGrid, see selecting data section in dataGrid documentation for more information and examples.

## Client Side API

Widget: *PrimeFaces.widget.DataList*

Method	Params	Return Type	Description
getPaginator()	-	Paginator	Returns the datagrid paginator instance.

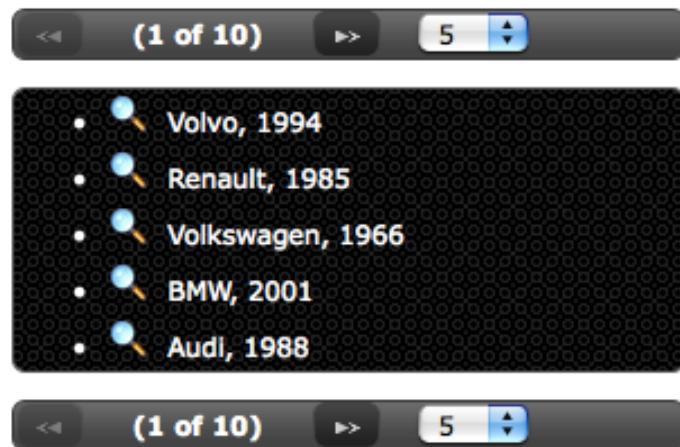
## Skinning

DataList resides in a main div container which style and styleClass attributes apply.

Following is the list of structural style classes;

Class	Applies
.ui-datalist	Main container element
.ui-datalist-content	Content container
.ui-datalist-data	Data container
.ui-datalist-item	Each item in list

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## Tips

- DataGrid uses a YUI paginator widget which you can retrieve via `widgetVar.getPaginator()`. It has handy client side methods like `getTotalRecords()`, `setPage` and more. For api documentation, see [http://developer.yahoo.com/yui/docs/module\\_paginator.html](http://developer.yahoo.com/yui/docs/module_paginator.html).

## 3.25 DataTable

DataTable is an enhanced version of the standard Datatable that provides built-in solutions to many commons use cases like paging, sorting, selection, lazy loading, filtering and more.

Model	Year	Manufacturer	Color
62da385e	2000	Opel	Green
9429b69f	2007	Mercedes	Brown
0b1db0d0	2001	BMW	Orange
1edc33c9	1977	Audi	Red
c9a6048e	2009	Opel	Blue
272e6dc3	1964	Ford	Brown
cbde87b3	2007	Volkswagen	Yellow
c0f941e5	1998	Opel	White
41a1c297	2003	Volvo	Maroon

### Info

Tag	<b>dataTable</b>
Component Class	<b>org.primefaces.component.datatable.DataTable</b>
Component Type	<b>org.primefaces.component.DataTable</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.DataTableRenderer</b>
Renderer Class	<b>org.primefaces.component.datatable.DataTableRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Data to display.
var	null	String	Name of the request-scoped variable used to refer each data.
rows	null	Integer	Number of rows to display per page.

Name	Default	Type	Description
first	0	Integer	Index of the first row to be displayed
widgetVar	null	String	Variable name of the javascript widget.
paginator	FALSE	boolean	Enables pagination.
paginatorTemplate	null	String	Template of the paginator.
rowsPerPageTemplate	null	String	Template of the rowsPerPage dropdown.
currentPageReportTemplate	null	String	Template of the currentPageReport UI.
pageLinks	10	Integer	Maximum number of page links to display.
paginatorPosition	both	String	Position of the paginator.
paginatorAlwaysVisible	TRUE	Boolean	Defines if paginator should be hidden if total data count is less than number of rows per page.
page	1	Integer	Index of the current page
style	null	String	Inline style of the main container.
styleClass	null	String	Style class of the main container.
scrollable	FALSE	Boolean	Makes data scrollable with fixed header.
height	null	Integer	Height in pixels for scrollable data.
selectionMode	null	String	Enables data selection, valid values are <i>single</i> , <i>multiple</i> , <i>singlcell</i> and <i>multiplecell</i> .
selection	null	Object	Reference to the selection data.
lazy	FALSE	Boolean	Enables lazy loading.
rowIndexVar	null	String	Name of iterator to refer each row index.
emptyMessage	No records found.	String	Text to display when there is no data to display.
update	null	String	Deprecated use onRowSelectUpdate instead.
onRowSelectUpdate	null	String	Component(s) to update instantly after a row is selected.
rowSelectListener	null	MethodExpr	Server side listener to invoke when a row is selected.
rowUnselectListener	null	MethodExpr	Server side listener to invoke when a row is unselected.
onRowUnselectUpdate	null	String	Component(s) to update instantly after a row is unselected.

Name	Default	Type	Description
onselectStart	null	String	Client side callback to execute before ajax request to select a row begins.
onselectComplete	null	String	Client side callback to execute before ajax request to select a row begins.
dblClickSelect	FALSE	Boolean	Client side callback to execute after ajax request to select a row ends.
liveScroll	FALSE	Boolean	Enables live scrolling.
rowStyleClass	null	String	Style class for each row.
rowEditListener	null	MethodExpr	Server side listener to invoke when an incell editor row is saved.
onRowEditUpdate	null	String	Component(s) to update after rowEditListener is invoked.
onExpandStart	null	String	Client side callback to execute before row expansion.

## Getting started with the DataTable

We will be using the same Car and CarBean classes described in DataGrid section.

```
<p:dataTable var="car" value="#{carBean.cars}">
    <p:column>
        <f:facet name="header">
            <h:outputText value="Model" />
        </f:facet>
        <h:outputText value="#{car.model}" />
    </p:column>
    <p:column>
        <f:facet name="header">
            <h:outputText value="Year" />
        </f:facet>
        <h:outputText value="#{car.year}" />
    </p:column>
    <p:column>
        <f:facet name="header">
            <h:outputText value="Manufacturer" />
        </f:facet>
        <h:outputText value="#{car.manufacturer}" />
    </p:column>
    <p:column>
        <f:facet name="header">
            <h:outputText value="Color" />
        </f:facet>
        <h:outputText value="#{car.color}" />
    </p:column>
</p:dataTable>
```

## Header and Footer

Both datatable itself and columns can have headers and footers.

```
<p:datatable var="car" value="#{carBean.cars}" widgetVar="carsTable">

    <f:facet name="header">
        List of Cars
    </f:facet>

    <p:column>
        <f:facet name="header">
            Model
        </f:facet>
        #{car.model}
        <f:facet name="footer">
            8 digit code
        </f:facet>
    </p:column>

    <p:column headerText="Year" footerText="1960-2010">
        #{car.year}
    </p:column>

    //more columns

    <f:facet name="header">
        In total there are #{fn:length(carBean.cars)} cars.
    </f:facet>

</p:datatable>
```

List of Cars			
Model	Manufacturer	Color	Year
16c9b7c6	Mercedes	Maroon	1979
de0e4475	Volkswagen	Maroon	1994
d17a0cac	Ford	Black	1998
0db0095d	Ford	Red	1983
c09b2d08	Renault	Red	1962
a5e3c203	Volkswagen	Green	2007
196bd9e9	Ford	White	1994
111db4d2	Ford	Silver	1994
73b17bd0	Volvo	Blue	1973
8 digit code			1960-2010
In total there are 9 cars.			

*headerText* and *footerText* attributes on column are handy shortcuts for *header* and *footer* facets that just display plain texts.

## Pagination

DataTable has a built-in ajax based paginator that is enabled by setting paginator option to true.

```
<p: dataTable var="car" value="#{carBean.cars}" paginator="true" rows="10">
    //columns
</p: dataTable>
```

Pagination configuration and usage is same as dataGrid, see pagination section in dataGrid documentation for more information and examples.

## Sorting

Sorting is specified at column level, defining *sortBy* attribute enables ajax based sorting on that particular column.

```
<p: dataTable var="car" value="#{carBean.cars}">

    <p: column sortBy="#{car.model}">
        <f: facet name="header">
            <h: outputText value="Model" />
        </f: facet>
        <h: outputText value="#{car.model}" />
    </p: column>

    ...more columns
</p: dataTable>
```

Instead of using the default sorting algorithm which uses a java comparator, you can plug-in your own sort method.

```
<p: dataTable var="car" value="#{carBean.cars}" dynamic="true">
    <p: column sortBy="#{car.model}" sortFunction="#{carBean.sortByModel}">
        <f: facet name="header">
            <h: outputText value="Model" />
        </f: facet>
        <h: outputText value="#{car.model}" />
    </p: column>

    ...more columns
</p: dataTable>
```

```
public int sortByModel(Car car1, Car car2) {
    //return -1, 0 , 1 if car1 is less than, equal to or greater than car2
}
```

## Data Filtering

Similar to sorting, ajax based filtering is enabled at column level by setting *filterBy* option.

```
<p:dataTable var="car" value="#{carBean.cars}">

    <p:column filterBy="#{car.model}">
        <f:facet name="header">
            <h:outputText value="Model" />
        </f:facet>
        <h:outputText value="#{car.model}" />
    </p:column>

    ...more columns

</p:dataTable>
```

By default filtering is triggered with keyup event, this is configurable using *filterEvent* attribute. Filter inputs can be styled using *filterStyle* and *filterStyleClass* attributes. If you'd like to use a dropdown instead of an inputtext to only allow predefined filter values use *filterOptions* attribute and a collection/array of selectitems as value. In addition, *filterMatchMode* defines the built-in matcher which is *startsWith* by default. Following is an advanced filtering datatable with these options demonstrated.

```
<p:dataTable var="car" value="#{carBean.cars}" widgetVar="carsTable">

    <f:facet name="header">
        <p:outputPanel>
            <h:outputText value="Search all fields:" />
            <h:inputText id="globalFilter" onkeyup="carsTable.filter()" />
        </p:outputPanel>
    </f:facet>

    <p:column filterBy="#{car.model}" headerText="Model" filterMatchMode="contains">
        <h:outputText value="#{car.model}" />
    </p:column>

    <p:column filterBy="#{car.year}" headerText="Year" footerText="startsWith">
        <h:outputText value="#{car.year}" />
    </p:column>

    <p:column filterBy="#{car.manufacturer}" headerText="Manufacturer"
        filterOptions="#{carBean.manufacturerOptions}" filterMatchMode="exact">
        <h:outputText value="#{car.manufacturer}" />
    </p:column>

    <p:column filterBy="#{car.color}" headerText="Color" filterMatchMode="endsWith">
        <h:outputText value="#{car.color}" />
    </p:column>

</p:dataTable>
```

<input type="text" value="Search all fields:"/>			
Model	Year	Manufacturer	Color
9f1e82ad	1989	Volkswagen	Black
c1362b1d	1968	Mercedes	Blue
eclf0bb1	1962	Renault	Green
9b0b3fe3	2001	Mercedes	Yellow
de0517b3	2002	Volkswagen	Green
3e702116	1972	BMW	Blue
49612134	1994	Ford	Black
bf19778d	1983	Audi	Red
4ecd938b	1962	Opel	Yellow
<a href="#">contains</a>	<a href="#">startsWith</a>	<a href="#">exact</a>	<a href="#">endsWith</a>

Filter located at header is a global one applying on all fields, this is implemented by calling client side api method called `filter()`, important part is to specify the id of the input text as `globalFilter` which is a reserved identifier for datatable.

## Row Selection

There are several ways to select row(s) from datatable. Let's begin by adding a Car reference for single selection and a Car array for multiple selection to the CarBean to hold the selected data.

```
public class CarBean {

    private List<Car> cars;

    private Car selectedCar;

    private Car[] selectedCars;

    public CarBean() {
        cars = new ArrayList<Car>();
        cars.add(new Car("myModel",2005,"ManufacturerX","blue"));
        //add more cars
    }

    //getters and setters
}
```

### Single Selection with a Command Component

This method is implemented with a command component such as `commandLink` or `commandButton`. Selected row can be set to a server side instance by passing as a parameter if you are using EL 2.2 or using `f:setPropertyActionListener`.

```
<p:dataTable var="car" value="#{carBean.cars}">

    <p:column>
        <p:commandButton value="Select">
            <f:setPropertyActionListener value="#{car}" target="#{carBean.selectedCar}" />
        </p:commandButton>
    </p:column>
    ...
</p:dataTable>
```

### *Single Selection with Row Click*

Previous method works when the button is clicked, if you'd like to enable selection wherever the row is clicked, use *selectionMode* option.

```
<p:dataTable var="car" value="#{carBean.cars}" selectionMode="single"
    selection="#{carBean.selectedCar}">
    ...
</p:dataTable>
```

### *Multiple Selection with Row Click*

Multiple row selection is similar to single selection but selection should reference an array of the domain object displayed.

```
<p:dataTable var="car" value="#{carBean.cars}" selectionMode="multiple"
    selection="#{carBean.selectedCars}">
    ...
</p:dataTable>
```

### *Instant Selection with Row Click*

In both single and multiple selection options described before, enclosing form needs to be submitted by user so that the selections would be processed and set to the selection value reference. If you'd like to execute custom logic whenever the row is selected instantly bind a *rowSelectListener* or define *onRowSelectUpdate* option. If you'd like to get notified when a row is unselected instantly, use *rowUnselectListener*.

Following example displays a dialog with the selected car information once a row is selected;

```

<h:form id="carListForm">
    <p:dataTable var="car" value="#{carBean.cars}" selectionMode="single"
        selection="#{carBean.selectedCar}"
        onRowSelectUpdate="carListForm:dialogContent"
        rowSelectListener="#{carBean.onCarSelect}"
        onRowSelectComplete="carDialog.show()">

        ...
        .columns

    </p:dataTable>

    <p:dialog widgetVar="carDialog" header="Car Info">
        <p:outputPanel id="dialogContent">
            <h:outputText value="Model: #{carBean.selectedCar.model}" />
            //more information about selected car
        </p:outputPanel>
    </p:dialog>
</p:outputPanel>
</h:form>

```

```

public void onCarSelect(SelectEvent event) {
    Car car = (Car) event.getObject();
}

```

`rowSelectListener` is handy in case you need to execute custom logic on selected row data, an `org.primefaces.event.SelectEvent` will be passed providing reference to the selected object. It is also aware of JSF navigations, for example you can navigate to another page to display information about selected data instead of staying on same page. In this case, return an outcome;

```

public String onCarSelect(SelectEvent event) {
    FacesContext.getCurrentInstance().getExternalContext().getFlash().put
("selectedCar", event.getObject());

    return "carDetail?faces-redirect=true";
}

```

### Selection on Double Click

By default, row based selection is enabled by click event, enable `dblClickSelect` so that clicking double on a row does the selection.

### Single Selection with RadioButton

Selection a row with a radio button placed on each row is a common case, datatable has built-in support for this method so that you don't need to deal with `h:selectOneRadio`'s and low level bits. In order to enable this feature, define a column with `selectionMode` set as `single`.

```
<p:dataTable var="car" value="#{carBean.cars}" selection="#{carBean.selectedCar}">
    <p:column selectionMode="single"/>
    ...
</p:dataTable>
```

### *Multiple Selection with Checkboxes*

Similar to how radio buttons are enabled, define a selection column with a multiple selectionMode. DataTable will also provide a selectAll checkbox at column header.

```
<p:dataTable var="car" value="#{carBean.cars}" selection="#{carBean.selectedCars}">
    <p:column selectionMode="multiple"/>
    ...
</p:dataTable>
```

### **Cell Selection**

Cell selection is used to select particular cell(s) in datatable, two different modes are supported; ‘singlecell’ and ‘multiplecell’. Selected cells are passed to the backing bean as *org.primefaces.model.Cell* instances.

```
<p:dataTable var="car" value="#{carBean.cars}"
    selection="#{carBean.selectedCell}"
    selectionMode="singlecell">
    ...
</p:dataTable>
```

```
public class CarBean {
    private List<Car> cars;
    private Cell selectedCell;
    //getters and setters
}
```

*org.primefaces.model.Cell* class has the following properties about the selected cell;

Property	Type	Description
rowData	Object	Row data the cell belongs to.
columnId	String	Id of the cell column
value	Object	Value displayed in cell.

For multiple cell selection use "multiple" selection mode, in this case selection should be a Cell[] reference instead of a single Cell.

## Dynamic Columns

Dynamic columns is handy in case you don't know how many columns to render. Columns component is used to define the columns programmatically. It requires a collection as the value, two iterator variables called *var* and *columnIndexVar*. Following example displays cars of each brand dynamically;

```
<p: dataTable var="cars" value="#{tableBean.dynamicCars}" id="carsTable">
    <p:columns value="#{tableBean.columns}" var="column" columnIndexVar="colIndex">
        <f:facet name="header">
            #{column}
        </f:facet>

        <h:outputText value="#{cars[colIndex].model}" /> <br />
        <h:outputText value="#{cars[colIndex].year}" /> <br />
        <h:outputText value="#{cars[colIndex].color}" />
    </p:columns>
</p: dataTable>
```

```
public class CarBean {

    private List<String> columns;

    private List<Car[]> dynamicCars;

    public CarBean() {
        populateColumns();
        populateCars();
    }

    public void populateColumns() {
        columns = new ArrayList();

        for(int i = 0; i < 3; i++) {
            columns.add("Brand:" + i);
        }
    }
}
```

```

public void populateCars() {
    dynamicCars = new ArrayList<Car[]>();

    for(int i=0; i < 9; i++) {
        Car[] cars = new Car[columns.size()];

        for(int j = 0; j < columns.size(); j++) {
            cars[j] = //Create car object
        }

        dynamicCars.add(cars);
    }
}

```

## Grouping

Grouping is defined by ColumnGroup component used to combine datatable header and footers.

```

<p:dataTable var="sale" value="#{carBean.sales}">

    <p:columnGroup type="header">
        <p:row>
            <p:column rowspan="3" headerText="Manufacturer" />
            <p:column colspan="4" headerText="Sales" />
        </p:row>
        <p:row>
            <p:column colspan="2" headerText="Sales Count" />
            <p:column colspan="2" headerText="Profit" />
        </p:row>
        <p:row>
            <p:column headerText="Last Year" />
            <p:column headerText="This Year" />
            <p:column headerText="Last Year" />
            <p:column headerText="This Year" />
        </p:row>
    </p:columnGroup>

    <p:column>
        #{sale.manufacturer}
    </p:column>
    <p:column>
        #{sale.lastYearProfit}%
    </p:column>
    <p:column>
        #{sale.thisYearProfit}%
    </p:column>
    <p:column>
        #{sale.lastYearSale}$
    </p:column>
    <p:column>
        #{sale.thisYearSale}$
    </p:column>

```

```
<p:columnGroup type="footer">
    <p:row>
        <p:column colspan="3" style="text-align:right" footerText="Totals:"/>
        <p:column footerText="#{tableBean.lastYearTotal}$" />
        <p:column footerText="#{tableBean.thisYearTotal}$" />
    </p:row>
</p:columnGroup>

</p:dataTable>
```

```
public class CarBean {

    private List<Manufacturer> sales;

    public CarBean() {
        sales = //create a list of BrandSale objects
    }

    public List<ManufacturerSale> getSales() {
        return this.sales;
    }
}
```

Manufacturer	Sales			
	Sales Count		Profit	
	Last Year	This Year	Last Year	This Year
Mercedes	90%	8%	28031\$	25102\$
BMW	14%	91%	18640\$	28023\$
Volvo	82%	24%	130\$	77724\$
Audi	7%	40%	2272\$	33672\$
Renault	10%	54%	98115\$	40664\$
Opel	63%	28%	10549\$	93746\$
Volkswagen	67%	38%	38242\$	19063\$
Chrysler	40%	63%	10146\$	7697\$
Ferrari	26%	70%	40384\$	62298\$
Ford	14%	94%	96052\$	42233\$
Totals:		342561\$	430222\$	

## Scrolling

Scrolling is a way to display data with fixed headers, in order to enable simple scrolling set scrollable option to true, define a fixed height in pixels and set a fixed width to each column.

```
<p:dataTable var="car" value="#{carBean.cars}" scrollable="true" height="150">

    <p:column style="width:100px" ...
    //columns
</p:dataTable>
```

Model	Year	Manufacturer	Color
069794d7	1991	Volvo	Silver
4aeeec6c	1993	Ford	Green
09cbc05c	1983	Chrysler	Maroon
2d374a04	1964	Ferrari	Red
9c09bc54	1987	Volkswagen	Blue
25d45a08	1993	Opel	White
Model	Year	Year	Year

Simple scrolling renders all data to client and places a scrollbar, live scrolling is necessary to deal with huge data, in this case data is fetched whenever the scrollbar reaches bottom. Set *liveScroll* to enable this option;

```
<p: dataTable var="car" value="#{carBean.cars}" scrollable="true" height="150"
    liveScroll="true">

    <p:column style="width:100px" ...
    //columns
</p: dataTable>
```

## Expandable Rows

*RowToggler* and *RowExpansion* components are used to implement expandable rows.

```
<p: dataTable var="car" value="#{carBean.cars}">

    <f: facet name="header">
        Expand rows to see detailed information
    </f: facet>

    <p: column>
        <p: rowToggler />
    </p: column>

    //columns

    <p: rowExpansion>
        //Detailed content of a car
    </p: rowExpansion>

</p: dataTable>
```

p:rowToggler component places an expand/collapse icon, clicking on a collapsed row loads expanded content with ajax.

Expand rows to see detailed information		
	Model	Year
0b8313c2		1976
2be34a8c		1995
08e342c4		2004
b5d03231		1998
 Model: b5d03231 Year: 1998 Manufacturer: Mercedes Color: Red		
b50b6dcc		1974
db39801c		1995
f76c474f		1989
2c9b67a2		2005
94fb553f		1973

In case you need to execute custom javascript before row expansion begins, use *onExpandStart* option, your function will be executed with row element as parameter named *row*.

## Incell Editing

Incell editing provides an easy way to display editable data. *p:cellEditor* is used to define the cell editor of a particular column and *p:rowEditor* is used to toggle edit/view displays of a row.

```
<p:dataTable var="car" value="#{carBean.cars}">

  <f:facet name="header">
    In-Cell Editing
  </f:facet>

  <p:column headerText="Model">

    <p:cellEditor>
      <f:facet name="output">
        <h:outputText value="#{car.model}" />
      </f:facet>

      <f:facet name="input">
        <h:inputText value="#{car.model}" />
      </f:facet>

    </p:cellEditor>
  </p:column>

  //more columns with cell editors

  <p:column>
    <p:rowEditor />
  </p:column>

</p:dataTable>
```

In-Cell Editing				
Model	Year	Manufacturer	Color	Options
824641ad	1976	Volvo	Yellow	
a9bf1625	1961	Volkswagen	Orange	
d859a7ba	1977	Ferrari	Brown	
9379f6f5	1961	Renault	Silver	
744a8017	1960	Chrysler	Silver	
80feefe5	2000	Opel	Yellow	
9e0c7267	1982	Opel	Red	
33124250	1984	Ford	Red	
0349899f	1977	Renault	Red	

When pencil icon is clicked, row is displayed in editable mode meaning input facets are displayed and output facets are hidden. Clicking tick icon only saves that particular row and cancel icon reverts the changes, both options are implemented with ajax.

Most of the time, you'd need a callback to do something with the edited data like persisting changes, in this case define a *rowEditListener* with org.primefaces.event.RowEditEvent as parameter;

```
<p: dataTable var="car" value="#{carBean.cars}"
    rowEditListener="#{carBean.onEditRow}">

    //editable columns

</p: dataTable>
```

```
public void onEditRow(RowEditEvent event) {
    Car editedCar = (Car) event.getObject();
    //persist to database
}
```

Optionally *onRowEditUpdate* attribute is used to update other components on page after successful row editing. Note that if validation fails, row will still be displayed as editable.

## Lazy Loading

Lazy Loading is a built-in feature of datatable to deal with huge datasets efficiently, regular ajax based pagination works by rendering only a particular page but still requires all data to be loaded into memory. Lazy loading datatable renders a particular page similarly but also only loads the page data into memory not the whole dataset. In order to implement this, you'd need to bind a org.primefaces.model.LazyDataModel as the value and implement one single method called *load*.

```
<p:dataTable var="car" value="#{carBean.model}" lazy="true"
    paginator="true" rows="10">

    //columns

</p:dataTable>
```

```
public class CarBean {

    private LazyDataModel model;

    public CarBean() {
        model = new LazyDataModel() {

            @Override
            public void load(int first, int pageSize, String sortField,
                boolean sortOrder, Map<String, String> filters) {

                //load lazy data
            }
        };

        int totalRowCount = //total row count based on a count query
        model.setRowCount(totalRowCount);
    }

    public LazyDataModel getModel() {
        return model;
    }
}
```

DataTable calls your load implementation whenever a paging, sorting or filtering occurs with following parameters;

- first: Offset of first data to start from
- pageSize: Number of data to load
- sortField: Name of sort field (e.g. "model" for sortBy="#{car.model}")
- sortOrder: True for ascending and False for descending
- filter: Filter map with field name as key (e.g. "model" for filterBy="#{car.model}") and value.

In addition to load method, totalRowCount needs to be provided so that paginator can display itself according to the logical number of rows to display.

## Client Side API

Widget: *PrimeFaces.widget.DataTable*

Method	Params	Return Type	Description
getPaginator()	-	Paginator	Returns the datagrid paginator instance.
clearFilters()	-	void	Clears all column filters

## Skinning

DataTable resides in a main container element which *style* and *styleClass* options apply.

Following is the list of structural style classes;

Class	Applies
.ui-datatable	Main container element
.ui-datatable-data	Table body
.ui-datatable-data-empty	Table body when there is no data
.ui-datatable-header	Table header
.ui-datatable-footer	Table footer
.ui-sortable-column	Sortable columns
.ui-sortable-column-icon	Icon of a sortable icon
.ui-expanded-row-content	Content of an expanded row
.ui-row-toggler	Row toggler for row expansion
.ui-editable-column	Columns with a cell editor
.ui-cell-editor	Container of input and output controls of an editable cell
.ui-cell-editor-input	Container of input control of an editable cell
.ui-cell-editor-output	Container of output control of an editable cell
.ui-datatable-even	Even numbered rows
.ui-datatable-odd	Odd numbered rows

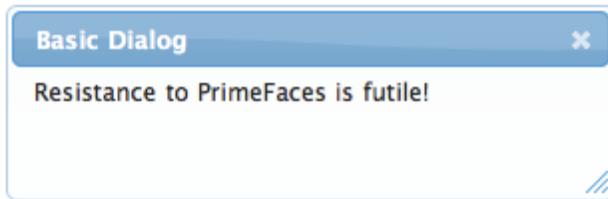
As skinning style classes are global, see the main Skinning section for more information.

## Tips

- JPA Query API has methods like *setFirstResult(int first)* and *setMaxResults(int limit)* you can use when implementing lazy loading as values like first result and max results are passed in load method.
- Filtering caches the filtered data, when updating a filtered datatable, use *resetValue()* method on datatable to clear cache. There are also useful methods like *reset()* and *clearLazyCache()*.

## 3.26 Dialog

Dialog is a panel component overlaying other elements. Dialog avoids popup blockers, provides customization, resizing, modality, ajax interactions and more.



### Info

Tag	<b>dialog</b>
Component Class	<b>org.primefaces.component.dialog.Dialog</b>
Component Type	<b>org.primefaces.component.Dialog</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.DialogRenderer</b>
Renderer Class	<b>org.primefaces.component.dialog.DialogRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
header	null	String	Text of the header
draggable	TRUE	Boolean	Specifies draggability
resizable	TRUE	Boolean	Specifies resizability
modal	FALSE	Boolean	Boolean value that specifies whether the document should be shielded with a partially transparent mask to require the user to close the Panel before being able to activate any elements in the document.
visible	FALSE	Boolean	Set's dialogs visibility

Name	Default	Type	Description
width	150	Integer	Width of the dialog
height	auto	Integer	Width of the dialog
zindex	1000	Integer	Specifies zindex property.
minWidth	150	Integer	Minimum width of a resizable dialog.
minHeight	0	Integer	Minimum height of a resizable dialog.
styleClass	null	String	Style class of the dialog
closeListener	null	MethodExpr	Server side listener to invoke when dialog is closed.
onCloseUpdate	null	String	Components to update after dialog is closed and closeListener is processed with ajax.
showEffect	null	String	Effect to use when showing the dialog
hideEffect	null	String	Effect to use when hiding the dialog
position	null	String	Defines where the dialog should be displayed
closeOnEscape	TRUE	Boolean	Defines if dialog should be closed when escape key is pressed.
closable	TRUE	Boolean	Defines if close icon should be displayed or not
onShow	null	String	Client side callback to execute when dialog is displayed.
onHide	null	String	Client side callback to execute when dialog is hidden.
appendToBody	FALSE	Boolean	Appends dialog as a child of document body.
widgetVar	null	String	Name of the client side widget

## Getting started with the Dialog

Dialog is a panel component containing other components, note that by default dialog is not visible.

```
<p:dialog>
    <h:outputText value="Resistance to PrimeFaces is Futile!" />
    //Other content
</p:dialog>
```

## Show and Hide

Showing and hiding the dialog is easy using the client side api.

```
<p:dialog header="Header Text" widgetVar="dlg">
    //Content
</p:dialog>

<p:commandButton value="Show" type="button" onclick="dlg.show()" />
<p:commandButton value="Hide" type="button" onclick="dlg.hide()" />
```

## Effects

There are various effect options to be used when displaying and closing the dialog. Use *showEffect* and *hideEffect* options to apply these effects;

- blind
- bounce
- clip
- drop
- explode
- fade
- fold
- highlight
- puff
- pulsate
- scale
- shake
- size
- slide
- transfer

```
<p:dialog showEffect="fade" hideEffect="explode" ...>
    //...
</p:dialog>
```

## Position

By default dialog is positioned at center of the viewport and *position* option is used to change the location of the dialog. Possible values are;

- Single string value like '*center*', '*left*', '*right*', '*top*', '*bottom*' representing the position within viewport.
  - Comma separated x and y coordinate values like *200, 500*
  - Comma separated position values like '*top*', '*right*'. (Use single quotes when using a combination)
- Some examples are described below;

```
<p:dialog position="top" ...>
```

```
<p:dialog position="'right','top'" ...>
```

```
<p:dialog position="200,500" ...>
```

## Ajax Interaction

Dialog can also be used for ajax interaction. In the following example when the dialog is shown, it displays a form to enter a text, once submit button is clicked, dialog is hidden and outputText with id="txt" is partially updated.

```
<h:form>
    <h:outputText id="txt" value="Text: #{bean.text}"/>
    <h:outputLink value="#" onclick="dlg.show()">Enter Text</h:outputLink>

    <p:dialog header="Enter FirstName" widgetVar="dlg">
        <h:panelGrid columns="2" style="margin-bottom:10px">
            <h:outputLabel for="text" value="Text:" />
            <h:inputText id="text" value="#{bean.text}" />

            <p:commandButton value="Reset" type="reset"/>
            <p:commandButton value="Submit" update="txt"
                oncomplete="dlg.hide();"/>
        </h:panelGrid>
    </p:dialog>
</h:form>
```

## Ajax CloseListener

If you'd like to execute custom logic on server side when dialog is closed, bind a *closeListener*. Your listener will be invoked passing an *org.primefaces.event.CloseEvent* instance. Optionally you can update other components on page after dialog is closed and ajax closelistener is invoked.

Example below adds a FacesMessage when dialog is closed and updates the messages component to display the added message.

```
<p:dialog closeListener="#{dialogBean.handleClose}" onCloseUpdate="msg">
    //Content
</p:dialog>

<p:messages id="msg" />
```

```
public class DialogBean {

    public void handleClose(CloseEvent event) {
        //Add facesmessage
    }
}
```

## Client Side Callbacks

Similar to close listener, onShow and onHide are handy callbacks for client side in case you need to execute custom javascript.

```
<p:dialog onShow="alert('Visible')" onHide="alert('Hidden')">
    //Content
</p:dialog>
```

## Client Side API

Widget: *PrimeFaces.widget.Dialog*

Method	Params	Return Type	Description
show()	-	void	Displays dialog.
hide()	-	void	Closes dialog.

## Skinning

Dialog resides in a main container element which *styleClass* option apply. Following is the list of structural style classes;

Style Class	Applies
.ui-dialog	Container element of dialog
.ui-dialog-titlebar	Title bar
.ui-dialog-title-dialog	Header text
.ui-dialog-titlebar-close	Close icon
.ui-dialog-content	Dialog body

As skinning style classes are global, see the main Skinning section for more information.

## Tips

- Avoid updating the dialog itself for better performance, instead update a container within a dialog.

## 3.27 Divider

Divider is used to separate elements in a toolbar.

### Info

Tag	<b>divider</b>
Component Class	<b>org.primefaces.component.divider.Divider</b>
Component Type	<b>org.primefaces.component.Divider</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.DividerRenderer</b>
Renderer Class	<b>org.primefaces.component.divider.DividerRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
style	null	String	Inline style of the divider.
styleClass	null	String	Style class of the divider.
type	dotted	String	Visual style of the divider.

### Getting started with Divider

See toolbar section for more information about how divider is used within a toolbar.

### Type

Divider has two visual styles;

dotted	solid
⋮	

## Skinning

Divider element resides in a main container element which *style* and *styleClass* options apply.

Following is the list of structural style classes;

Class	Applies
.ui-divider	Divider element

As skinning style classes are global, see the main Skinning section for more information.

## 3.28 Drag&Drop

Drag&Drop utilities of PrimeFaces consists of two components; Draggable and Droppable.

### 3.28.1 Draggable

#### Info

Tag	<b>draggable</b>
Component Class	<b>org.primefaces.component.dnd.Draggable</b>
Component Type	<b>org.primefaces.component.Draggable</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.DraggableRenderer</b>
Renderer Class	<b>org.primefaces.component.dnd.DraggableRenderer</b>

#### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
widgetVar	null	String	Name of the client side widget
for	null	String	Id of the component to add draggable behavior
disabled	FALSE	Boolean	Disables or enables dragging
axis	null	String	Specifies drag axis, valid values are 'x' and 'y'.
containment	null	String	Constraints dragging within the boundaries of containment element
helper	null	String	Helper element to display when dragging
revert	FALSE	Boolean	Reverts draggable to it's original position when not dropped onto a valid droppable
snap	FALSE	Boolean	Draggable will snap to edge of near elements
snapMode	null	String	Specifies the snap mode. Valid values are 'both', 'inner' and 'outer'.

Name	Default	Type	Description
snapTolerance	20	Integer	Distance from the snap element in pixels to trigger snap.
zindex	null	Integer	ZIndex to apply during dragging.
handle	null	String	Specifies a handle for dragging.
opacity	1	Double	Defines the opacity of the helper during dragging.
stack	null	String	In stack mode, draggable overlap is controlled automatically using the provided selector, dragged item always overlays other draggables.
grid	null	String	Dragging happens in every x and y pixels.
scope	null	String	Scope key to match draggables and droppables.
cursor	crosshair	String	CSS cursor to display in dragging.
dashboard	null	String	Id of the dashboard to connect with.

## Getting started with Draggable

Any component can be enhanced with draggable behavior, basically this is achieved by defining the id of component using the *for* attribute of draggable.

```
<p:panel id="pnl" header="Draggable Panel">
    <h:outputText value="This is actually a regular panel" />
</p:panel>

<p:draggable for="pnl"/>
```

If you omit the for attribute, parent component will be selected as the draggable target.

```
<h:graphicImage id="campnou" value="/images/campnou.jpg">
    <p:draggable />
</h:graphicImage>
```

## Handle

By default any point in dragged component can be used as handle, if you need a specific handle, you can define it with handle option. Following panel is dragged using its header only.

```
<p:panel id="pnl" header="Draggable Panel">
    <h:outputText value="I can only be dragged using my header" />
</p:panel>
<p:draggable for="pnl" handle="div.ui-panel-titlebar"/>
```

## Drag Axis

Dragging can be limited to either horizontally or vertically.

```
<p:panel id="pnl" header="Draggable Panel">
    <h:outputText value="I am dragged on an axis only" />
</p:panel>

<p:draggable for="pnl" axis="x or y"/>
```

## Clone

By default, actual component is used as the drag indicator, if you need to keep the component at it's original location, use a clone helper.

```
<p:panel id="pnl" header="Draggable Panel">
    <h:outputText value="I am cloned" />
</p:panel>

<p:draggable for="pnl" helper="clone"/>
```

## Revert

When a draggable is not dropped onto a matching droppable, revert option enables the component to move back to it's original position with an animation.

```
<p:panel id="pnl" header="Draggable Panel">
    <h:outputText value="I will be reverted back to my original position" />
</p:panel>

<p:draggable for="pnl" revert="true"/>
```

## Opacity

During dragging, opacity option can be used to give visual feedback, helper of following panel's opacity is reduced in dragging.

```
<p:panel id="pnl" header="Draggable Panel">
    <h:outputText value="My opacity is lower during dragging" />
</p:panel>

<p:draggable for="pnl" opacity="0.5"/>
```

## Grid

Defining a grid enables dragging in specific pixels. This value takes a comma separated dimensions in x,y format.

```
<p:panel id=" pnl " header=" Draggable Panel ">
  <h:outputText value=" I am dragged in grid mode " />
</p:panel>

<p:draggable for=" pnl " grid=" 20,40 "/>
```

## Containment

A draggable can be restricted to a certain section on page, following draggable cannot go outside of it's parent.

## 3.28.2 Droppable

### Info

Tag	<b>droppable</b>
Component Class	<b>org.primefaces.component.dnd.Droppable</b>
Component Type	<b>org.primefaces.component.Droppable</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.DroppableRenderer</b>
Renderer Class	<b>org.primefaces.component.dnd.DroppableRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
widgetVar	null	String	Variable name of the client side widget
dropListener	null	MethodExpr	A server side listener to process a DragDrop event.
for	null	String	Id of the component to add droppable behavior
disabled	FALSE	Boolean	Disables or enables droppable behavior
hoverStyleClass	null	String	Style class to apply when an acceptable draggable is dragged over.
activeStyleClass	null	String	Style class to apply when an acceptable draggable is being dragged.
onDropUpdate	null	String	Component(s) to update with ajax after a draggable is dropped.
onDrop	null	String	Client side callback to execute when a draggable is dropped.
accept	null	String	Selector to define the accepted draggables.
scope	null	String	Scope key to match draggables and dropables.
tolerance	null	String	Specifies the intersection mode to accept a draggable.
datasource	null	String	Id of a UIData component to connect with.

## Getting Started with Droppable

Usage of droppable is very similar to draggable, droppable behavior can be added to any component specified with the for attribute.

```
<p:outputPanel id="slot" styleClass="slot" />  
<p:droppable for="slot" />
```

slot styleClass represents a small rectangle.

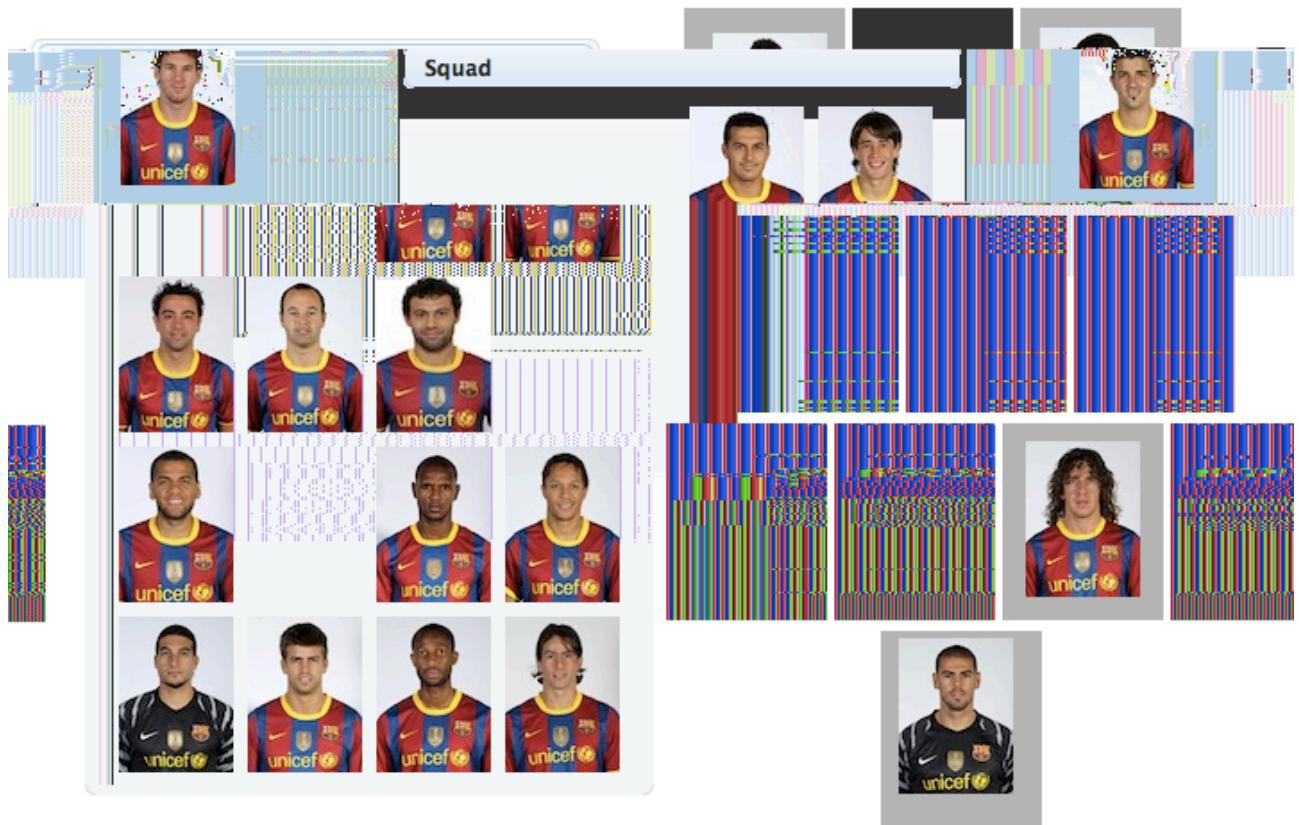
```
<style type="text/css">  
    .slot {  
        background:#FF9900;  
        width:64px;  
        height:96px;  
        display:block;  
    }  
</style>
```

If for attribute is omitted, parent component becomes droppable.

```
<p:outputPanel id="slot" styleClass="slot">  
    <p:droppable />  
</p:outputPanel>
```

## Drop Listener

PrimeFaces component showcase demo contains a functional example to setup tactical formation of F.C. Barcelona, see the source code for more information.



In addition to the ajax *dropListener*, *onDropUpdate* attribute is used to define which components to update after dropListener is processed.

### onDrop

onDrop is a client side callback that is invoked when a draggable is dropped, it gets two parameters event and ui object holding information about the drag drop event.

```
<p:outputPanel id="zone" styleClass="slot" />
<p:droppable for="zone" onDrop="handleDrop"/>
```

```
function handleDrop(event, ui) {
    var draggable = ui.draggable,      //draggable element, a jQuery object
        helper = ui.helper,           //helper element of draggable, a jQuery object
        position = ui.position,       //position of draggable helper
        offset = ui.offset;          //absolute position of draggable helper
}
```

## DataSource

Droppable has special care for data elements that extend from UIData(e.g. datatable, datagrid), in order to connect a droppable to accept data from a data component define datasource option as the id of the data component. Example below show how to drag data from datagrid and drop onto a droppable to implement a dragdrop based selection. Dropped cars are displayed with a datatable.

```

<h:form id="carForm">
    <p:fieldset legend="AvailableCars">
        <p:datagrid id="availableCars" var="car"
            value="#{tableBean.availableCars}" columns="3">
            <p:column>
                <p:panel id=" pnl " header="#{car.model}" style="text-align:center">
                    <p:graphicImage value="/images/cars/#{car.manufacturer}.jpg" />
                </p:panel>
                <p:draggable for=" pnl " revert="true" handle=".ui-panel-titlebar" stack=".ui-panel"/>
            </p:column>
        </p:datagrid>
    </p:fieldset>

    <p:fieldset id="selectedCars" legend="Selected Cars" style="margin-top:20px">
        <p:outputPanel id="dropArea">

            <h:outputText value="!!!Drop here!!!"
                rendered="#{empty tableBean.droppedCars}" style="font-size:24px;" />

            <p:datatable var="car" value="#{tableBean.droppedCars}"
                rendered="#{not empty tableBean.droppedCars}">
                <p:column headerText="Model">
                    <h:outputText value="#{car.model}" />
                </p:column>
                <p:column headerText="Year">
                    <h:outputText value="#{car.year}" />
                </p:column>
                <p:column headerText="Manufacturer">
                    <h:outputText value="#{car.manufacturer}" />
                </p:column>
                <p:column headerText="Color">
                    <h:outputText value="#{car.color}" />
                </p:column>
            </p:datatable>
        </p:outputPanel>
    </p:fieldset>

    <p:droppable for="selectedCars" tolerance="touch" activeStyleClass="ui-state-highlight" datasource="availableCars" dropListener="#{tableBean.onCarDrop}"
        onDropUpdate="dropArea availableCars" onDrop="handleDrop"/>

</h:form>

<script type="text/javascript">
    function handleDrop(event, ui) {
        ui.draggable.fadeOut('fast');           //fade out the dropped item
    }
</script>

```

```

public class TableBean {

    private List<Car> availableCars;
    private List<Car> droppedCars;

    public TableBean() {
        availableCars = //populate data
    }

    //getters and setters

    public void onCarDrop(DragDropEvent event) {
        Car car = ((Car) ddEvent.getData());
        droppedCars.add(car);
        availableCars.remove(car);
    }
}

```

## Tolerance

There are four different tolerance modes that define the way of accepting a draggable.

Mode	Description
fit	draggable should overlap the droppable entirely
intersect	draggable should overlap the droppable at least 50%
pointer	pointer of mouse should overlap the droppable
touch	droppable should overlap the droppable at any amount

## Acceptance

You can limit which draggables can be dropped onto droppables using scope attribute which a draggable also has. Following example has two images, only first image can be accepted by droppable.

```

<p:graphicImage id="messi" value="barca/messi_thumb.jpg" />
<p:draggable for="messi" scope="forward"/>

<p:graphicImage id="xavi" value="barca/xavi_thumb.jpg" />
<p:draggable for="xavi" scope="midfield"/>

<p:outputPanel id="forwardsonly" styleClass="slot" scope="forward" />
<p:droppable for="forwardsonly" />

```

## Skinning

*hoverStyleClass* and *activeStyleClass* attributes are used to change the style of the droppable when interacting with a draggable.

## 3.29 Dock

Dock component mimics the well known dock interface of Mac OS X.



### Info

Tag	<b>dock</b>
Component Class	<b>org.primefaces.component.dock.Dock</b>
Component Type	<b>org.primefaces.component.Dock</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.DockRenderer</b>
Renderer Class	<b>org.primefaces.component.dock.DockRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
position	bottom	String	Position of the dock, <i>bottom</i> or <i>top</i> .
itemWidth	40	Integer	Initial width of items.
maxWidth	50	Integer	Maximum width of items.
proximity	90	Integer	Distance to enlarge.
halign	center	String	Horizontal alignment,
model	null	MenuModel	MenuModel instance to create menus programmatically
widgetVar	null	String	Name of the client side widget.

## Getting started with the Dock

A dock is composed of menuitems.

```
<p:dock>
    <p:menuitem value="Home" icon="/images/dock/home.png" url="#" />
    <p:menuitem value="Music" icon="/images/dock/music.png" url="#" />
    <p:menuitem value="Video" icon="/images/dock/video.png" url="#" />
    <p:menuitem value="Email" icon="/images/dock/email.png" url="#" />
    <p:menuitem value="Link" icon="/images/dock/link.png" url="#" />
    <p:menuitem value="RSS" icon="/images/dock/rss.png" url="#" />
    <p:menuitem value="History" icon="/images/dock/history.png" url="#" />
</p:dock>
```

## Position

Dock can be located in two locations, *top* or *bottom* (default). For a dock positioned at top set position to top.

## Dock Effect

When mouse is over the dock items, icons are zoomed in. The configuration of this effect is done via the maxWidth and proximity attributes.

## Dynamic Menus

Menus can be created programmatically as well, see the dynamic menus part in menu component section for more information and an example.

## Skinning

Following is the list of structural style classes, {positon} can be *top* or *bottom*.

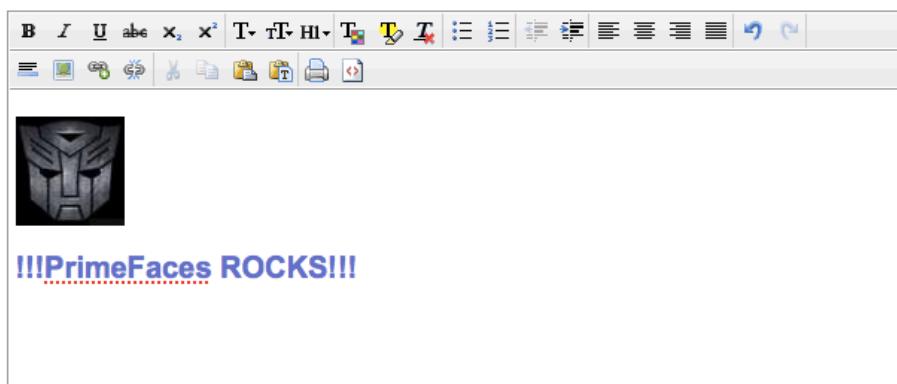
Style Class	Applies
.ui-dock-{position}	Main container.
.ui-dock-container-{position}	Menu item container.
.ui-dock-item-{position}	Each menu item.

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.30 Editor

Editor is an input component with rich text editing capabilities.



### Info

Tag	<code>editor</code>
Component Class	<code>org.primefaces.component.editor.Editor</code>
Component Type	<code>org.primefaces.component.Editor</code>
Component Family	<code>org.primefaces.component</code>
Renderer Type	<code>org.primefaces.component.EditorRenderer</code>
Renderer Class	<code>org.primefaces.component.editor.EditorRenderer</code>

### Attributes

Name	Default	Type	Description
<code>id</code>	<code>null</code>	String	Unique identifier of the component.
<code>rendered</code>	<code>TRUE</code>	Boolean	Boolean value to specify the rendering of the component.
<code>binding</code>	<code>null</code>	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
<code>value</code>	<code>null</code>	Object	Value of the component than can be either an EL expression or a literal text.
<code>converter</code>	<code>null</code>	Converter/ String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id.

Name	Default	Type	Description
immediate	FALSE	Boolean	When set true, process validations logic is executed at apply request values phase for this component.
required	FALSE	Boolean	Marks component as required.
validator	null	MethodExpr	A method expression that refers to a method validationg the input.
valueChangeListener	null	MethodExpr	A method expression that refers to a method for handling a valuchangeevent.
requiredMessage	null	String	Message to be displayed when required field validation fails.
converterMessage	null	String	Message to be displayed when conversion fails.
validatorMessage	null	String	Message to be displayed when validation fails.
widgetVar	null	String	Name of the widget to access client side api.
controls	null	String	List of controls to customize toolbar.
height	null	Integer	Height of the editor.
width	null	Integer	Width of the editor.
disabled	FALSE	Boolean	Disables editor.
lazy	FALSE	Boolean	Enables lazy initialization mode.

## Getting started with the Editor

Rich Text entered using the Editor is passed to the server using *value* expression.

```
public class Bean {
    private String text;

    //getter and setter
}
```

```
<h:form>
    <p:editor value="#{myController.text}" />

    <p:commandButton value="Save" />
</h:form>
```

## Custom Toolbar

Toolbar of editor is easy to customize using *controls* option;

```
<p:editor value="#{bean.text}" controls="bold italic underline strikethrough" />
```



Here is the full list of all available controls;

<ul style="list-style-type: none"> <li>• bold</li> <li>• italic</li> <li>• underline</li> <li>• strikethrough</li> <li>• subscript</li> <li>• superscript</li> <li>• font</li> <li>• size</li> <li>• style</li> <li>• color</li> <li>• highlight</li> <li>• bullets</li> <li>• numbering</li> <li>• alignleft</li> <li>• center</li> <li>• alignright</li> </ul>	<ul style="list-style-type: none"> <li>• justify</li> <li>• undo</li> <li>• redo</li> <li>• rule</li> <li>• image</li> <li>• link</li> <li>• unlink</li> <li>• cut</li> <li>• copy</li> <li>• paste</li> <li>• pastetext</li> <li>• print</li> <li>• source</li> <li>• outdent</li> <li>• indent</li> <li>• removeFormat</li> </ul>
--	---

## Lazy Editor

Editor cannot function properly when placed inside a container that is initially hidden (e.g. dialog, tabview). Workaround is to use a lazy editor and create the editor when the container is visible. Following example demonstrates how to place an editor inside a dialog.

```
<p:dialog onShow="editorWidget.init()">
    <p:editor value="#{bean.text}" lazy="true" widgetVar="editorWidget"/>
</p:dialog>
```

## Client Side API

Widget: *PrimeFaces.widget.Editor*

Method	Params	Return Type	Description
init()	-	void	Initializes a lazy editor, subsequent calls do not reinit the editor.
saveHTML()	-	void	Saves html text in iframe back to the textarea.
clear()	-	void	Clears the text in editor.
enable()	-	void	Enables editing.
disable()	-	void	Disables editing.
focus()	-	void	Adds cursor focus to edit area.
selectAll()	-	void	Selects all text in editor.
getSelectedHTML()	-	String	Returns selected text as HTML.
getSelectedText()	-	String	Returns selected text in plain format.

## Skinning

Following is the list of structural style classes.

Style Class	Applies
.ui-editor	Main container.
.ui-editor-toolbar	Toolbar of editor.
.ui-editor-group	Button groups.
.ui-editor-button	Each button.
.ui-editor-divider	Divider to separate buttons.
.ui-editor-disabled	Disabled editor controls.
.ui-editor-list	Dropdown lists.
.ui-editor-color	Color picker.
.ui-editor-popup	Popup overlays.
.ui-editor-prompt	Overlays to provide input.
.ui-editor-message	Overlays displaying a message.

Editor is not integrated with ThemeRoller as there is only one icon set for the controls.

## 3.31 Effect

Effect component is based on the jQuery UI effects library.

### Info

Tag	<b>effect</b>
Tag Class	<b>org.primefaces.component.effect.EffectTag</b>
Component Class	<b>org.primefaces.component.effect.Effect</b>
Component Type	<b>org.primefaces.component.Effect</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.EffectRenderer</b>
Renderer Class	<b>org.primefaces.component.effect.EffectRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
event	null	String	Dom event to attach the event that executes the animation
type	null	String	Specifies the name of the animation
for	null	String	Component that is animated
speed	1000	Integer	Speed of the animation in ms

### Getting started with Effect

Effect component needs a trigger and target which is effect's parent by default. In example below clicking outputText (trigger) will run the pulsate effect on outputText(target) itself.

```
<h:outputText value="#{bean.value}">
    <p:effect type="pulsate" event="click" />
</h:outputText>
```

## Effect Target

There may be cases where you want to display an effect on another target on the same page while keeping the parent as the trigger. Use *for* option to specify a target.

```
<h:outputLink id="lnk" value="#">  
    <h:outputText value="Show the Barca Temple" />  
    <p:effect type="appear" event="click" for="img" />  
</h:outputLink>  
  
<p:graphicImage id="img" value="/ui/barca/campnou.jpg" style="display:none"/>
```

With this setting, outputLink becomes the trigger for the effect on graphicImage. When the link is clicked, initially hidden graphicImage comes up with a fade effect.

**Note:** It's important for components that have the effect component as a child to have an assigned id because some components do not render their clientId's if you don't give them an id explicitly.

## List of Effects

Following is the list of effects supported by PrimeFaces.

- blind
- clip
- drop
- explode
- fold
- puff
- slide
- scale
- bounce
- highlight
- pulsate
- shake
- size
- transfer

## Effect Configuration

Each effect has different parameters for animation like colors, duration and more. In order to change the configuration of the animation, provide these parameters with the f:param tag.

```
<h:outputText value="#{bean.value}">
    <p:effect type="scale" event="mouseover">
        <f:param name="percent" value="90"/>
    </p:effect>
</h:outputText>
```

It's important to provide string options with single quotes.

```
<h:outputText value="#{bean.value}">
    <p:effect type="blind" event="click">
        <f:param name="direction" value="'horizontal'" />
    </p:effect>
</h:outputText>
```

For the full list of configuration parameters for each effect, please see the jquery documentation;

<http://docs.jquery.com/UI/Effects>

## Effect on Load

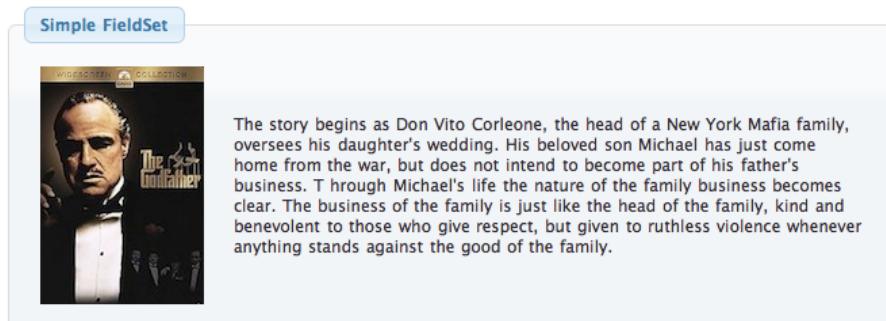
Effects can also be applied to any JSF component when page is loaded for the first time or after an ajax request is completed. Following example animates messages with pulsate effect after ajax request.

```
<p:messages id="messages">
    <p:effect type="pulsate" event="load">
        <f:param name="mode" value="'show'" />
    </p:effect>
</p:messages>

<p:commandButton value="Save" actionListener="#{bean.action}" update="messages"/>
```

## 3.32 Fieldset

Fieldset is a grouping component with a title and content.



### Info

<b>Tag</b>	<b>fieldset</b>
Component Class	<b>org.primefaces.component.fieldset.Fieldset</b>
Component Type	<b>org.primefaces.component.Fieldset</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.FieldsetRenderer</b>
Renderer Class	<b>org.primefaces.component.fieldset.FieldsetRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
widgetVar	null	String	Name of the widget to access client side api.
legend	null	String	Title text.
style	null	String	Inline style of the fieldset.
styleClass	null	String	Style class of the fieldset.
toggable	FALSE	Boolean	Makes content toggable with animation.
onToggleUpdate	null	String	Component(s) to update with ajax after toggleListener is invoked.

Name	Default	Type	Description
toggleListener	null	MethodExpr	Server side listener to invoke when content is toggled.
collapsed	FALSE	Boolean	Defines initial visibility state of content.

## Getting started with Fieldset

Fieldset is used as a container component for its children.

```
<p:fieldset legend="Simple Fieldset">
    <h:panelGrid column="2">
        <p:graphicImage value="/images/godfather/1.jpg" />
        <h:outputText value="The story begins as Don Vito Corleone ..." />
    </h:panelGrid>
</p:fieldset>
```

## Legend

Legend can be defined in two ways, with legend attribute as in example above or using legend facet. Use facet way if you need to place custom html other than simple text.

```
<p:fieldset>
    <f:facet name="legend">
    </f:facet>

    //content
</p:fieldset>
```

When both legend attribute and legend facet are present, facet is chosen.

## Toggleable Content

Clicking on fieldset legend can toggle contents, this is handy to use space efficiently in a layout. Set toggleable to true to enable this feature.

```
<p:fieldset legend="Toggleable Fieldset" toggleable="true">
    <h:panelGrid column="2">
        <p:graphicImage value="/images/godfather/2.jpg" />
        <h:outputText value="Francis Ford Coppolas' legendary ..." />
    </h:panelGrid>
</p:fieldset>
```

**-- Toggleable Fieldset**

Francis Ford Coppola's legendary continuation and sequel to his landmark 1972 film, *The Godfather*, parallels the young Vito Corleone's rise with his son Michael's spiritual fall, deepening *The Godfather's* depiction of the dark side of the American dream. In the early 1900s, the child Vito flees his Sicilian village for America after the local Mafia kills his family. Vito struggles to make a living, legally or illegally, for his wife and growing brood in Little Italy, killing the local Black Hand Fanucci after he demands his customary cut of the tyro's business. With Fanucci gone, Vito's communal stature grows.

## ToggleListener

It is possible to execute custom logic both on server and client side when content is toggled. `toggleListener` is a server side listener that is invoked with `org.primefaces.event.ToggleEvent` as a parameter, optional `onToggleUpdate` defines components to update with the ajax toggle request.

Here is an example that adds a faces message using `toggleListener` and updates growl component when fieldset is toggled.

```
<p:growl id="messages" />

<p:fieldset legend="Toggleable Fieldset" toggleable="true"
    toggleListener="#{bean.onToggle}" onToggleUpdate="messages">
    //content
</p:fieldset>
```

```
public void onToggle(ToggleEvent event) {
    Visibility visibility = event.getVisibility();
    FacesMessage msg = new FacesMessage();
    msg.setSummary("Fieldset " + event.getId() + " toggled");
    msg.setDetail("Visibility: " + visibility);

    FacesContext.getCurrentInstance().addMessage(null, msg);
}
```

## Client Side API

Widget: `PrimeFaces.widget.Fieldset`

Method	Params	Return Type	Description
<code>toggle()</code>	-	<code>void</code>	Toggles fieldset content.

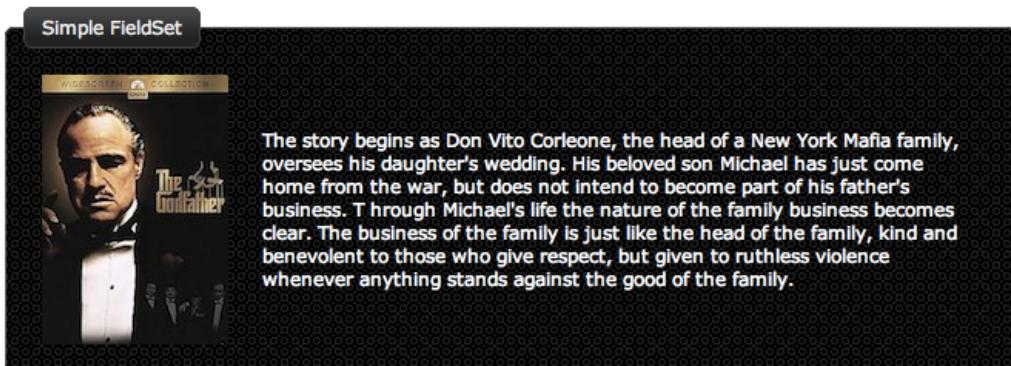
## Skinning

*style* and *styleClass* options apply to the fieldset.

Following is the list of structural style classes;

Style Class	Applies
.ui-fieldset	Main container
.ui-fieldset-toggleable	Main container when fieldset is toggleable
.ui-fieldset .ui-fieldset-legend	Legend of fieldset
.ui-fieldset-toggleable .ui-fieldset-legend	Legend of fieldset when fieldset is toggleable
.ui-fieldset .ui-fieldset-toggler	Toggle icon on fieldset

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## Tips

- A collapsed fieldset will remain collapsed after a postback since fieldset keeps its toggle state internally, you don't need to manage this using toggleListener and collapsed option.

## 3.33 FileDownload

The legacy way to present dynamic binary data to the client is to write a servlet or a filter and stream the binary data. FileDownload does all the hardwork and presents an easy binary data like files stored in database.

### Info

Tag	<b>fileDownload</b>
Tag Class	<b>org.primefaces.component.filedownload.FileDownloadTag</b>
ActionListener Class	<b>org.primefaces.component.filedownload.FileDownloadActionListener</b>

### Attributes

Name	Default	Type	Description
value	null	StreamedContent	A streamed content instance
contextDisposition	attachment	String	Specifies display mode.

### Getting started with FileDownload

A user command action is required to trigger the filedownload process. FileDownload can be attached to any command component like a commandButton or commandLink.

The value of the FileDownload must be an *org.primefaces.model.StreamedContent* instance. We suggest using the built-in *DefaultStreamedContent* implementation. First parameter of the constructor is the binary stream, second is the mimeType and the third parameter is the name of the file.

```
public class FileBean {

    private StreamedContent file;

    public FileDownloadController() {
        InputStream stream = this.getClass().getResourceAsStream("yourfile.pdf");
        file = new DefaultStreamedContent(stream, "application/pdf",
                                         "downloaded_file.pdf");
    }

    public StreamedContent getFile() {
        return this.file;
    }
}
```

This streamed content should be bound to the value of the fileDownload.

```
<h:commandButton value="Download">
    <p:fileDownload value="#{fileBean.file}" />
</h:commandButton>
```

Similarly a more graphical presentation would be to use a commandlink with an image.

```
<h:commandLink value="Download">
    <p:fileDownload value="#{fileBean.file}" />
    <h:graphicImage value="pdficon.gif" />
</h:commandLink>
```

If you'd like to use PrimeFaces commandButton and commandLink, disable ajax option as fileDownload requires a full page refresh to present the file.

```
<p:commandButton value="Download" ajax="false">
    <p:fileDownload value="#{fileBean.file}" />
</p:commandButton>
```

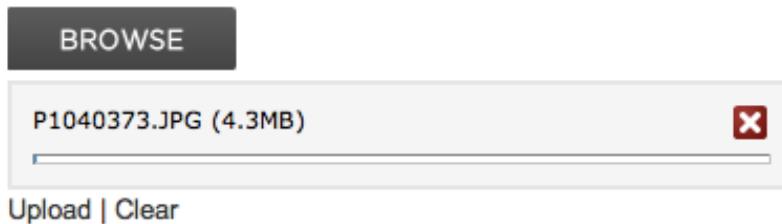
```
<p:commandLink value="Download" ajax="false">
    <p:fileDownload value="#{fileBean.file}" />
    <h:graphicImage value="pdficon.gif" />
</p:commandLink>
```

## ContentDisposition

By default, content is displayed as an *attachment* with a download dialog box, another alternative is the *inline* mode, in this case browser will try to open the file internally without a prompt. Note that content disposition is not part of the http standard although it is widely implemented.

## 3.34 FileUpload

FileUpload goes beyond the browser input type="file" functionality and features a flash-javascript solution for uploading files. File filtering, multiple uploads, partial page rendering and progress tracking are the significant features compared to legacy fileUploads.



### Info

Tag	<b>fileUpload</b>
Tag Class	<b>org.primefaces.component.fileupload.FileUploadTag</b>
Component Class	<b>org.primefaces.component.fileupload.FileUpload</b>
Component Type	<b>org.primefaces.component.FileUpload</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.FileUploadRenderer</b>
Renderer Class	<b>org.primefaces.component.fileupload.FileUploadRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
fileUploadListener	null	MethodExpression	Method expression to listen file upload events.
multiple	FALSE	boolean	Allows multi file uploads, turned off by default.
update	null	String	Client side ids of the component(s) to be updated after file upload completes.
auto	FALSE	boolean	When set to true, selecting a file starts the upload process implicitly.

Name	Default	Type	Description
label	null	String	Label of the browse button, default is 'Browse'
image	null	String	Background image of the browse button.
cancelImage	null	String	Image of the cancel button
width	null	String	Width of the browse button
height	null	String	Height of the browse button
allowTypes	null	String	Semi colon separated list of file extensions to accept.
description	null	String	Label to describe what types of files can be uploaded.
sizeLimit	null	Integer	Number of maximum bytes to allow.
wmode	null	String	wmode property of the flash object.
customUI	null	boolean	When custom UI is turned on upload and cancel links won't be rendered.
style	null	String	Style of the main container element.
styleClass	null	String	Style class of the main container element.
widgetVar	null	String	Name of the javascript widget.

## Getting started with FileUpload

First thing to do is to configure the fileupload filter which parses the multipart request. It's important to make PrimeFaces file upload filter the very first filter to consume the request.

```
<filter>
    <filter-name>PrimeFaces FileUpload Filter</filter-name>
    <filter-class>
        org.primefaces.webapp.filter.FileUploadFilter
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>PrimeFaces FileUpload Filter</filter-name>
    <servlet-name>Faces Servlet</servlet-name>
</filter-mapping>
```

## Single File Upload

By default file upload allows selecting and uploading only one file at a time, simplest file upload would be;

```
<p:fileUpload fileUploadListener="#{fileBean.handleFileUpload}" />
```



FileUploadListener is the way to access the uploaded files, when a file is uploaded defined fileUploadListener is processed with a FileUploadEvent as the parameter.

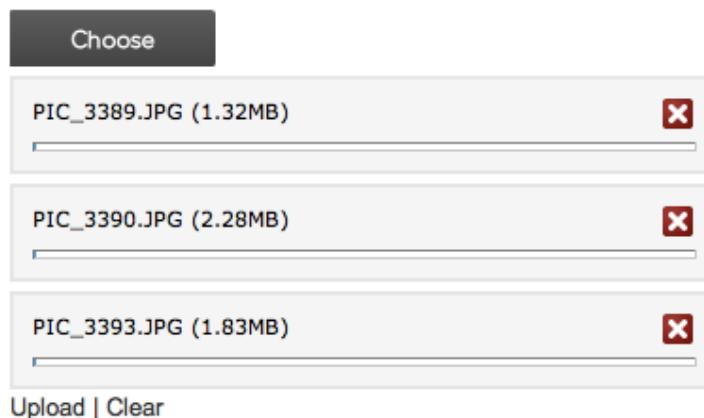
```
public class FileBean {  
  
    public void handleFileUpload(FileUploadEvent event) {  
        UploadedFile file = event.getFile();  
        //application code  
    }  
}
```

UploadedFile belongs to the PrimeFaces API and contains methods to retrieve various information about the file such as filesize, contents, file type and more. Please see the JavaDocs for more information.

## Multi FileUploads

Multiple fileuploads can be enabled using the multiple attribute. This way multiple files can be selected and uploaded together.

```
<p:fileUpload fileUploadListener="#{fileBean.handleFileUpload}" multiple="true" />
```



## Auto Upload

Default behavior requires users to trigger the upload process, you can change this way by setting auto to true. Auto uploads are triggered as soon as files are selected from the dialog.

```
<p:fileUpload fileUploadListener="#{fileBean.handleFileUpload}" auto="true" />
```

## Partial Page Update

After the fileUpload process completes you can use the PrimeFaces PPR to update any component on the page. FileUpload is equipped with the update attribute for this purpose. Following example displays a "File Uploaded" message using the growl component after file upload.

```
<p:fileUpload fileUploadListener="#{fileBean.handleFileUpload}" update="msg" />
<p:growl id="msg" />
```

```
public class FileBean {
    public void handleFileUpload(FileUploadEvent event) {
        //add facesmessage to display with growl
        //application code
    }
}
```

## File Filters

Users can be restricted to only select the file types you've configured, for example a file filter defined on \*.jpg will only allow selecting jpg files. Several different file filters can be configured for a single fileUpload component.

```
<p:fileUpload fileUploadListener="#{fileBean.handleFileUpload}"
    allowTypes="*.jpg;*.bmp;*.png;*.gif" description="Select Images"/>
```

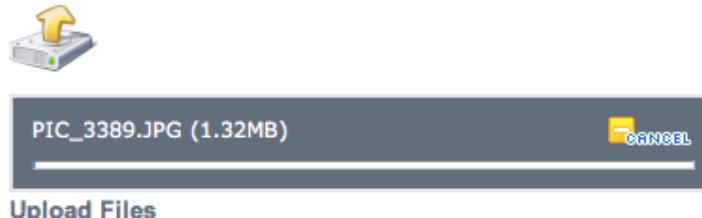
## Size Limit

Most of the time you might need to restrict the file upload size, this is as simple as setting the sizeLimit configuration. Following fileUpload limits the size to 1000 bytes for each file.

```
<p:fileUpload fileUploadListener="#{fileBean.handleFileUpload}" sizeLimit="1000" />
```

## Skinning FileUpload

FileUpload is a highly customizable component in terms of skinning. Best way to show this is start with an example. After skinning the fileUpload will look like;



```
<p:fileUpload widgetVar="uploader"
    fileUploadListener="#{fileBean.handleFileUpload}" height="48" width="48"
    image="/images/browse.png"
    cancelImage="/images/cancel.png" customUI="true"/>

<h:outputLink value="#" title="Upload" onclick="uploader.upload();">
    Upload Files
</h:outputLink>
```

The image of the browse button is customized using the image attribute and the image for cancel button is configured with cancelImage attribute. Note that when you use a custom image for the browse button set the height and width properties to be same as the image size. In addition, style and styleClass attributes apply to the main container element(span) of fileupload controls. Another important feature is the customUI. Since fileUpload is a composite component, we made the UI flexible enough to customize it for your own requirements. When customUI is set to true, default upload and cancel links are not rendered and it's up to you to handle these events if you want using the client side api. There're two simple methods upload() and clear() that you can use to plug-in your own UI.

## Filter Configuration

FileUpload filter's default settings can be configured with init parameters. Two configuration options exist, threshold size and temporary file upload location.

Parameter Name	Description
thresholdSize	Maximum file size in bytes to keep uploaded files in memory. If a file exceeds this limit, it'll be temporarily written to disk.
uploadDirectory	Disk repository path to keep temporary files that exceeds the threshold size. By default it is System.getProperty("java.io.tmpdir")

An example configuration below defined thresholdSize to be 50kb and uploads to user's temporary folder.

```
<filter>
    <filter-name>PrimeFaces FileUpload Filter</filter-name>
    <filter-class>
        org.primefaces.webapp.filter.FileUploadFilter
    </filter-class>
    <init-param>
        <param-name>thresholdSize</param-name>
        <param-value>51200</param-value>
    </init-param>
    <init-param>
        <param-name>uploadDirectory</param-name>
        <param-value>/Users/primefaces/temp</param-value>
    </init-param>
</filter>
```

## 3.35 Focus

Focus is a handy component that makes it easy to manage the element focus on a JSF page.

### Info

Tag	<b>focus</b>
Tag Class	<b>org.primefaces.component.focus.FocusTag</b>
Component Class	<b>org.primefaces.component.focus.Focus</b>
Component Type	<b>org.primefaces.component.Focus.FocusTag</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.FocusRenderer</b>
Renderer Class	<b>org.primefaces.component.focus.FocusRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
for	null	String	Specifies the exact component to set focus
context	null	String	The root component to start first input search.
minSeverity	error	String	Minimum severity level to be used when finding the first invalid component

### Getting started with Focus

In it's simplest form, focus is enabled by just placing the component on the page as;

```
<p:focus />
```

That's it, now let's add some input components to give something to focus on.

## Input Focus

You don't need to explicitly define the component to receive focus as by default focus will find the *first enabled and visible input component* on page. Input component can be any element such as input, textarea and select. Following is a simple example;

```
<h:form>
    <p:panel id="panel" header="Register">

        <p:focus />

        <p:messages />

        <h:panelGrid columns="3">
            <h:outputLabel for="firstname" value="Firstname: *" />
            <h:inputText id="firstname" value="#{pprBean.firstname}"
                         required="true" label="Firstname" />
            <p:message for="firstname" />

            <h:outputLabel for="surname" value="Surname: *" />
            <h:inputText id="surname" value="#{pprBean.surname}"
                         required="true" label="Surname"/>
            <p:message for="surname" />
        </h:panelGrid>

        <p:commandButton value="Submit" update="panel"
                          actionListener="#{pprBean.savePerson}" />
    </p:panel>
</h:form>
```

When this page initially opens, input text with id "firstname" will receive focus as it is the first input component.

## Validation Aware

Another useful feature of focus is that when validations fail, first invalid component will receive a focus. So in previous example if firstname field is valid but surname field has no input, a validation error will be raised for surname, in this case focus will be set on surname field implicitly. Note that for this feature to work on ajax requests, you need to update p:focus component as well.

## Explicit Focus

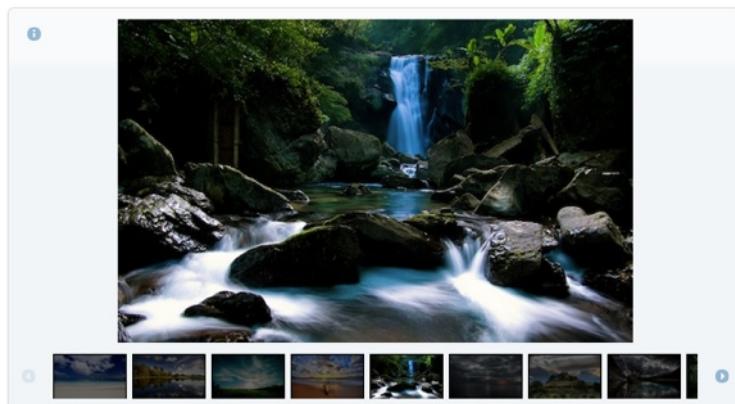
Additionally, using for attribute focus can be set explicitly on an input component which is useful when using a dialog.

```
<p:focus for="text"/>

<h:inputText id="text" value="#{bean.value}" />
```

## 3.36 Galleria

Galleria is used to display a set of images.



### Info

Tag	<b>galleria</b>
Component Class	<b>org.primefaces.component.galleria.Galleria</b>
Component Type	<b>org.primefaces.component.Galleria</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.GalleriaRenderer</b>
Renderer Class	<b>org.primefaces.component.galleria.GalleriaRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
widgetVar	null	String	Name of the widge to access client side api.
style	null	String	Inline style of the container element.
styleClass	null	String	Style class of the container element.
effect	fade	String	Name of animation to use.
effectSpeed	700	Integer	Duration of animation in milliseconds.

## Getting Started with Galleria

Images to displayed are defined as children of galleria;

```
<p:galleria effect="slide" effectDuration="1000">
    <p:graphicImage value="/images/image1.jpg" title="image1" alt="image1 desc" />
    <p:graphicImage value="/images/image2.jpg" title="image1" alt=" image2 desc" />
    <p:graphicImage value="/images/image3.jpg" title="image1" alt=" image3 desc" />
    <p:graphicImage value="/images/image4.jpg" title="image1" alt=" image4 desc" />
</p:galleria>
```

Galleria displays the details of an image using an overlay which is displayed by clicking the information icon. Title of this popup is retrieved from the image *title* attribute and description from *alt* attribute so it is suggested to provide these attributes as well.

## Dynamic Collection

Most of the time, you would need to display a dynamic set of images rather than defining each image declaratively. For this you can use ui:repeat.

```
<p:galleria>
    <ui:repeat value="#{galleriaBean.images}" var="image">
        <p:graphicImage value="#{image.path}"
                        title="#{image.title}" alt="#{image.description}" />
    </ui:repeat>
</p:galleria>
```

## Effects

There are four different options for the animation to display when switching images;

- fade (default)
- flash
- pulse
- slide

By default animation takes 700 milliseconds, use *effectDuration* option to tune this.

```
<p:galleria effect="slide" effectDuration="1000">
    <p:graphicImage value="/images/image1.jpg" title="image1" alt="image1 desc" />
    <p:graphicImage value="/images/image2.jpg" title="image1" alt=" image2 desc" />
    <p:graphicImage value="/images/image3.jpg" title="image1" alt=" image3 desc" />
    <p:graphicImage value="/images/image4.jpg" title="image1" alt=" image4 desc" />
</p:galleria>
```

## Skinning

Galleria resides in a main container element which *style* and *styleClass* options apply.

Following is the list of structural style classes;

Style Class	Applies
.ui-galleria-container	Container element for galleria.
.ui-galleria-stage	Container displaying actual images.
.ui-galleria-thumbnails-container	Container displaying thumbnail images.
.ui-galleria-thumbnails-list	Thumbnail images list
.ui-galleria-thumbnails .ui-galleria-image	Each thumbnail in list
.ui-galleria-counter	Text showing image index/total
.ui-galleria-info	Info overlay container
.ui-galleria-text	Text in info overlay.
.ui-galleria-title	Info title
.ui-galleria-description	Info description
.ui-galleria-image-thumb-nav-left	Left thumbnail navigator
.ui-galleria-image-thumb-nav-right	Right thumbnail navigator

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.37 GMap

GMap component is built on Google Maps API Version 3. Gmap is highly integrated with JSF development model and enhanced with Ajax capabilities.



### Info

Tag	<b>gmap</b>
Tag Class	<b>org.primefaces.component.gmap.GMapTag</b>
Component Class	<b>org.primefaces.component.gmap.GMap</b>
Component Type	<b>org.primefaces.component.Gmap</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.GmapRenderer</b>
Renderer Class	<b>org.primefaces.component.gmap.GmapRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.

Name	Default	Type	Description
widgetVar	null	String	Name of the client side widget.
model	null	MapModel	An org.primefaces.model.MapModel instance.
style	null	String	Inline style of the map container.
styleClass	null	String	Style class of the map container.
type	null	String	Type of the map.
center	null	String	Center point of the map.
zoom	8	Integer	Defines the initial zoom level.
onOverlaySelectUpdate	null	String	Component(s) to update with ajax when an overlay is selected.
onOverlaySelectStart	null	String	Javascript callback to execute before ajax request to select an overlay begins.
onOverlaySelectComplete	null	String	Javascript callback to execute after ajax request to select an overlay completes.
overlaySelectListener	null	MethodExpr	Server side listener to invoke when an overlay is selected with ajax.
stateChangeListener	null	MethodExpr	Server side listener to invoke when stated of the map is changed.
onStateChangeUpdate	null	String	Component(s) to update with ajax when state of the map is changed.
pointSelectListener	null	MethodExpr	Server side listener to invoke when a point on map is selected.
onPointSelectUpdate	null	String	Component(s) to update with ajax when a point on map is selected.
markerDragListener	null	MethodExpr	Server side listener to invoke when a marker on map is dragged.
onMarkerDragUpdate	null	String	Component(s) to update with ajax when a marker on map is dragged.
streetView	FALSE	Boolean	Controls street view support.
disableDefaultUI	FALSE	Boolean	Disables default UI controls
navigationControl	TRUE	Boolean	Defines visibility of navigation control.
mapTypeControl	TRUE	Boolean	Defines visibility of map type control.
draggable	TRUE	Boolean	Defines draggability of map.
disabledDoubleClickZoom	FALSE	Boolean	Disables zooming on mouse double click.
onPointClick	null	String	Javascript callback to execute when a point on map is clicked.

## Getting started with GMap

First thing to do is placing V3 of the Google Maps API that the GMap based on. Ideal location is the head section of your page.

```
<script src="http://maps.google.com/maps/api/js?sensor=true|false"
    type="text/javascript"></script>
```

As Google Maps api states, mandatory sensor parameter is used to specify if your application requires a sensor like GPS locator. Also you don't need an api key anymore with the V3 api. Four options are required to place a gmap on a page, these are center, zoom, type and style.

```
<p:gmap center="41.381542, 2.122893" zoom="15" type="hybrid"
    style="width:600px;height:400px" />
```

*center*: Center of the map in lat, lng format

*zoom*: Zoom level of the map

*type*: Type of map, valid values are, "hybrid", "satellite", "hybrid" and "terrain".

*style*: Dimensions of the map.

## MapModel

GMap is backed by an *org.primefaces.model.map.MapModel* instance, PrimeFaces provides *org.primefaces.model.map.DefaultMapModel* as the default implementation. API Docs of all GMap related model classes are available at the end of GMap section and also at javadocs of PrimeFaces.

## Markers

A marker is represented by *org.primefaces.model.map.Marker* class and can be displayed on a map using a MapModel.

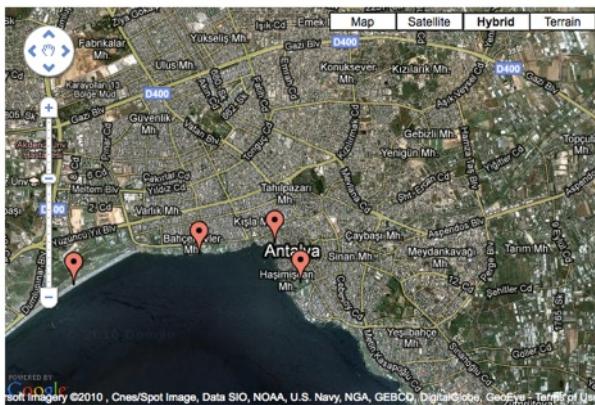
```
<p:gmap center="41.381542, 2.122893" zoom="15" type="hybrid"
    style="width:600px;height:400px" model="#{mapBean.model}" />
```

```
public class MapBean {

    private MapModel model = new DefaultMapModel();

    public MapBean() {
        model.addOverlay(new Marker(new LatLng(36.879466, 30.667648), "M1"));
        //more overlays
    }

    public MapModel getModel() { return this.model; }
}
```



## Polyline

A polyline is represented by `org.primefaces.model.map.Polyline` and can be displayed on a map using a `MapModel`.

```
<p:gmap center="41.381542, 2.122893" zoom="15" type="hybrid"
        style="width:600px;height:400px" model="#{mapBean.model}"/>
```

```
public class MapBean {

    private MapModel model;

    public MapBean() {
        model = new DefaultMapModel();

        Polyline polyline = new Polyline();
        polyline.getPaths().add(new LatLng(36.879466, 30.667648));
        polyline.getPaths().add(new LatLng(36.883707, 30.689216));
        polyline.getPaths().add(new LatLng(36.879703, 30.706707));
        polyline.getPaths().add(new LatLng(36.885233, 37.702323));

        model.addOverlay(polyline);
    }

    public MapModel getModel() { return this.model; }
}
```



## Polygons

A polygon is represented by `org.primefaces.model.map.Polygon` and can be displayed on a map using a MapModel.

```
<p:gmap center="41.381542, 2.122893" zoom="15" type="hybrid"
        style="width:600px;height:400px" model="#{mapBean.model}" />
```

```
public class MapBean {

    private MapModel model;

    public MapBean() {
        model = new DefaultMapModel();

        Polygon polygon = new Polygon();
        polyline.getPaths().add(new LatLng(36.879466, 30.667648));
        polyline.getPaths().add(new LatLng(36.883707, 30.689216));
        polyline.getPaths().add(new LatLng(36.879703, 30.706707));

        model.addOverlay(polygon);
    }

    public MapModel getModel() { return this.model; }
}
```



## Overlay Selection

Overlays such as markers, polylines and polygons can respond to selection by invoking a server side `overlaySelectListener` with ajax, passing an `overlaySelectEvent` that contains a reference to the selected overlay. Optionally other components can be updated with ajax using `onOverlaySelectUpdate` attribute. `onOverlaySelectStart` and `onOverlaySelectComplete` are optional client side callbacks.

Following example displays a FacesMessage about the selected marker with growl component.

```
<h:form>
    <p:growl id="growl" />

    <p:gmap center="41.381542, 2.122893" zoom="15" type="hybrid"
        style="width:600px;height:400px" model="#{mapBean.model}"
        overlaySelectListener="#{mapBean.onMarkerSelect}"
        onOverlaySelectUpdate="growl"/>

</h:form>
```

```
public class MapBean {

    private MapModel model;

    public MapBean() {
        model = new DefaultMapModel();
        //add markers
    }

    public MapModel getModel() {
        return model
    }

    public void onMarkerSelect(OverlaySelectEvent event) {
        Marker marker = (Marker) event.getOverlay();
        //add facesmessage
    }
}
```

## InfoWindow

A common use case is displaying an info window when a marker is selected. *gmapInfoWindow* is used to implement this special use case. Following example, displays an info window that contains an image of the selected marker data.

```
<h:form>

    <p:gmap center="41.381542, 2.122893" zoom="15" type="hybrid"
        style="width:600px;height:400px" model="#{mapBean.model}"
        overlaySelectListener="#{mapBean.onMarkerSelect}">
        <p:gmapInfoWindow>
            <p:graphicImage value="/images/#{mapBean.marker.data.image}" />
            <h:outputText value="#{mapBean.marker.data.title}" />
        </p:gmapInfoWindow>
    </p:gmap>

</h:form>
```

```

public class MapBean {

    private MapModel model;

    private Marker marker;

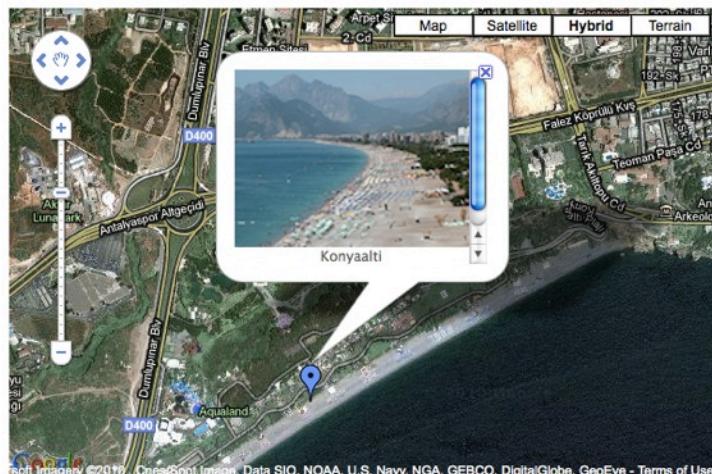
    public MapBean() {
        model = new DefaultMapModel();
        //add markers
    }

    public MapModel getModel() { return model; }

    public Marker getMarker() { return marker; }

    public void onMarkerSelect(OverlaySelectEvent event) {
        this.marker = (Marker) event.getOverlay();
    }
}

```



## Draggable Markers

When a draggable marker is dragged and dropped, a server side `markerDragListener` can be invoked, passing a `MarkerDragEvent` that contains a reference to the dragged marker whose position is updated already. Optional `onMarkerDragUpdate` options enables updating other component(s) on page after marker is dropped to it's new location.

```

<h:form>

    <p:growl id="growl" />
    <p:gmap center="41.381542, 2.122893" zoom="15" type="hybrid"
           style="width:600px;height:400px" model="#{mapBean.model}"
           markerDragListener="#{mapBean.onMarkerDrag}"
           onMarkerDragUpdate="growl"/>

</h:form>

```

```

public class MapBean {

    private MapModel model;

    public MapBean() {
        model = new DefaultMapModel();
        //create model with draggable markers
    }

    public MapModel getModel() { return model; }

    public void onMarkerDrag(MarkerDragEvent event) {
        Marker marker = (Marker) event.getMarker();
        //add facesmessage
    }
}

```

## Map Events

GMap itself can respond to events like drag and zoom change. When map state changes a server side *stateChangeListener* is invoked by passing a *StateChangeEvent* that contains information about new map state. Optional *onStateChangeUpdate* option enables updating other components on page after state change listener is invoked with ajax.

```

<h:form>
    <p:growl id="growl" />

    <p:gmap center="41.381542, 2.122893" zoom="15" type="hybrid"
        style="width:600px;height:400px" model="#{mapBean.model}"
        stateChangeListener="#{mapBean.onStateChange}"
        onStateChangeUpdate="growl"/>
</h:form>

```

```

public class MapBean {

    private MapModel model;

    public MapBean() {
        model = new DefaultMapModel();
        //create model with draggable markers
    }

    public MapModel getModel() { return model; }

    public void onStateChange(StateChangeEvent event) {
        int zoom = event.getZoomLevel();
        LatLngBounds bounds = event.getBounds();

        //add facesmessage
    }
}

```

## Point Selection

When a point with no overlay is selected, a server side *pointSelectListener* can be invoked passing a *PointSelectEvent* that contains information about the selected point. Optional *onPointUpdate* attribute allows updating other components on page after *pointSelectListener* is invoked with ajax.

```
<h:form>
    <p:growl id="growl" />
    <p:gmap center="41.381542, 2.122893" zoom="15" type="hybrid"
        style="width:600px;height:400px" model="#{mapBean.model}"
        pointSelectListener="#{mapBean.onPointSelect}"
        onPointSelectUpdate="growl"/>
</h:form>
```

```
public class MapBean {

    private MapModel model;

    public MapBean() {
        model = new DefaultMapModel();
        //create model
    }

    public MapModel getModel() { return model; }

    public void onPointSelect(PointSelectEvent event) {
        LatLng location = event.getLatLng();

        //add facesmessage
    }
}
```

## Street View

StreetView is enabled simply by setting *streetView* option to true.

```
<p:gmap center="41.381542, 2.122893" zoom="15" type="hybrid"
    style="width:600px;height:400px" streetView="true" />
```



## Map Controls

Controls on map can be customized via attributes like *navigationControl* and *mapTypeControl*. Alternatively setting *disableDefaultUI* to true will remove all controls at once.

```
<p:gmap center="41.381542, 2.122893" zoom="15" type="terrain"
        style="width:600px;height:400px"
        mapTypeControl="false" navigationControl="false" />
```



## Map inside Hidden Container

When map is placed inside an initially hidden container like a dialog or tabview, like other components that do dimension calculation, it cannot work properly. Workaround is to calculate the map dimensions using *checkResize()* method when the container becomes visible. Following example demonstrates how to properly display gmap in a modal dialog;

```
<p:commandButton type="button" value="Show Map" onclick="dlg.show()" />

<p:dialog widgetVar="dlg" width="625" height="450" modal="true"
    onShow="mymap.checkResize()>
        <p:gmap center="41.381542, 2.122893" zoom="15" type="HYBRID"
            style="width:600px;height:400px" widgetVar="mymap"/>
    </p:dialog>
```

*onShow* callback of dialog is a perfect place to execute *checkResize* on map.

## Native Google Maps API

In case you need to access native google maps api with javascript, use provided *getMap()* method.

```
var gmap = yourWidgetVar.getMap();
//gmap is a google.maps.Map instance
```

Full map api is provided at;

<http://code.google.com/apis/maps/documentation/javascript/reference.html>

## GMap API

*org.primefaces.model.map.MapModel* (*org.primefaces.model.map.DefaultMapModel* is the default implementation)

Method	Description
addOverlay(Overlay overlay)	Adds an overlay to map
List<Marker> getMarkers()	Returns the list of markers
List<Polyline> getPolylines()	Returns the list of polylines
List<Polygon> getPolygons()	Returns the list of polygons
Overlay findOverlay(String id)	Finds an overlay by it's unique id

*org.primefaces.model.map.Overlay*

Property	Default	Type	Description
id	null	String	Id of the overlay, generated and used internally
data	null	Object	Data represented in marker

*org.primefaces.model.map.Marker* extends *org.primefaces.model.map.Overlay*

Property	Default	Type	Description
title	null	String	Text to display on rollover
latlng	null	LatLng	Location of the marker
icon	null	String	Icon of the foreground
shadow	null	String	Shadow image of the marker
cursor	pointer	String	Cursor to display on rollover
draggable	FALSE	Boolean	Defines if marker can be dragged
clickable	TRUE	Boolean	Defines if marker can be dragged
flat	FALSE	Boolean	If enabled, shadow image is not displayed
visible	TRUE	Boolean	Defines visibility of the marker

*org.primefaces.model.map.Polyline* extends *org.primefaces.model.map.Overlay*

Property	Default	Type	Description
paths	null	List	List of coordinates
strokeColor	null	String	Color of a line
strokeOpacity	1	Double	Opacity of a line
strokeWeight	1	Integer	Width of a line

*org.primefaces.model.map.Polygon* extends *org.primefaces.model.map.Overlay*

Property	Default	Type	Description
paths	null	List	List of coordinates
strokeColor	null	String	Color of a line
strokeOpacity	1	Double	Opacity of a line
strokeWeight	1	Integer	Width of a line
fillColor	null	String	Background color of the polygon
fillOpacity	1	Double	Opacity of the polygon

*org.primefaces.model.map.LatLng*

Property	Default	Type	Description
lat	null	double	Latitude of the coordinate
lng	null	double	Longitude of the coordinate

*org.primefaces.model.map.LatLngBounds*

Property	Default	Type	Description
center	null	LatLng	Center coordinate of the boundary
northEast	null	LatLng	NorthEast coordinate of the boundary
southWest	null	LatLng	SouthWest coordinate of the boundary

## GMap Event API

All classes in event api extends from *javax.faces.event.FacesEvent*.

*org.primefaces.event.map.MarkerDragEvent*

Property	Default	Type	Description
marker	null	Marker	Dragged marker instance

*org.primefaces.event.map.OverlaySelectEvent*

Property	Default	Type	Description
overlay	null	Overlay	Selected overlay instance

*org.primefaces.event.map.PointSelectEvent*

Property	Default	Type	Description
latLng	null	LatLng	Coordinates of the selected point

*org.primefaces.event.map.StateChangeEvent*

Property	Default	Type	Description
bounds	null	LatLngBounds	Boundaries of the map
zoomLevel	0	Integer	Zoom level of the map

## 3.38 GMapInfoWindow

GMapInfoWindow is used with GMap component to open a window on map when an overlay is selected.



### Info

Tag	<b>gmapInfoWindow</b>
Tag Class	<b>org.primefaces.component.gmap.GMapInfoWindowTag</b>
Component Class	<b>org.primefaces.component.gmap.GMapInfoWindow</b>
Component Type	<b>org.primefaces.component.GMapInfoWindow</b>
Component Family	<b>org.primefaces.component</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
maxWidth	null	Integer	Maximum width of the info window

### Getting started with GMapInfoWindow

See GMap section for more information about how gmapInfoWindow is used.

## 3.39 GraphicImage

PrimeFaces GraphicImage extends standard JSF graphic image component with the ability of displaying binary data like an inputstream. Main use cases of GraphicImage is to make displaying images stored in database or on-the-fly images easier. Legacy way to do this is to come up with a Servlet that does the streaming, GraphicImage does all the hard work without the need of a Servlet.

### Info

Tag	<b>graphicImage</b>
Tag Class	<b>org.primefaces.component.graphicimage.GraphicImageTag</b>
Component Class	<b>org.primefaces.component.graphicimage.GraphicImage</b>
Component Type	<b>org.primefaces.component.GraphicImage</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.GraphicImageRenderer</b>
Renderer Class	<b>org.primefaces.component.graphicimage.GraphicImageRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Binary data to stream or context relative path.
alt	null	String	Alternate text for the image
url	null	String	Alias to value attribute
width	null	String	Width of the image
height	null	String	Height of the image
title	null	String	Title of the image
dir	null	String	Direction of the text displayed
lang	null	String	Language code
ismap	FALSE	Boolean	Specifies to use a server-side image map
usemap	null	String	Name of the client side map

Name	Default	Type	Description
style	null	String	Style of the image
styleClass	null	String	Style class of the image
onclick	null	String	onclick dom event handler
ondblclick	null	String	ondblclick dom event handler
onkeydown	null	String	onkeydown dom event handler
onkeypress	null	String	onkeypress dom event handler
onkeyup	null	String	onkeyup dom event handler
onmousedown	null	String	onmousedown dom event handler
onmousemove	null	String	onmousemove dom event handler
onmouseout	null	String	onmouseout dom event handler
onmouseover	null	String	onmouseover dom event handler
onmouseup	null	String	onmouseup dom event handler
cache	TRUE	String	Enables/Disables browser from caching the image

## Getting started with GraphicImage

GraphicImage requires a *org.primefaces.model.StreamedContent* content as it's value. StreamedContent is an interface and PrimeFaces provides a ready implementation called *DefaultStreamedContent*. Following examples loads an image from the classpath.

```
<p:graphicImage value="#{imageBean.image}" />
```

```
public class ImageBean {

    private StreamedContent image;

    public DynamicImageController() {
        InputStream stream = this.getClass().getResourceAsStream("barcalogo.jpg");
        image = new DefaultStreamedContent(stream, "image/jpeg");
    }

    public StreamedContent getImage() {
        return this.image;
    }
}
```

DefaultStreamedContent gets an inputstream as the first parameter and mime type as the second. Please see the javadocs if you require more information.

In a real life application, you can create the inputstream after reading the image from the database. For example `java.sql.ResultSet` API has the `getBinaryStream()` method to read blob files stored in database.

## Displaying Charts with JFreeChart

`StreamedContent` is a powerful API that can display images created on-the-fly as well. Here's an example that generates a chart with JFreeChart and displays it with `p:graphicImage`.

```
<p:graphicImage value="#{chartBean.chart}" />
```

```
public class ChartBean {

    private StreamedContent chart;

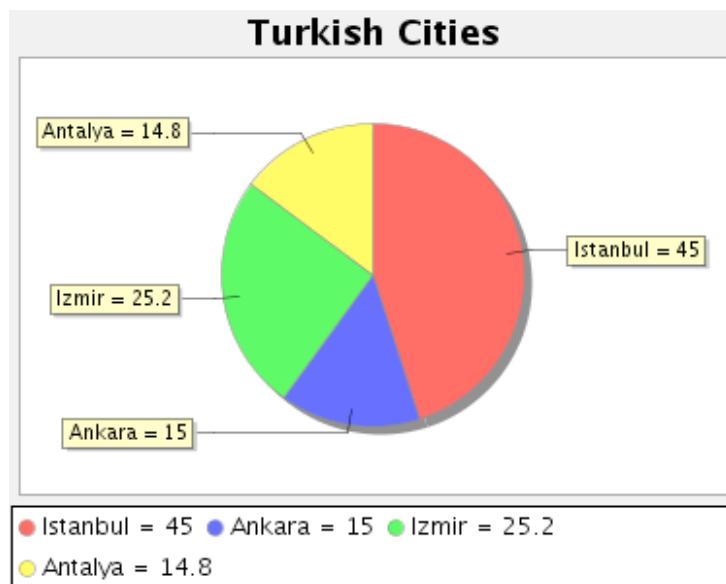
    public BackingBean() {
        try {
            JFreeChart jfreechart = ChartFactory.createPieChart(
                "Turkish Cities", createDataset(), true, true, false);
            File chartFile = new File("dynamichart");
            ChartUtilities.saveChartAsPNG(chartFile, jfreechart, 375, 300);
            chart = new DefaultStreamedContent(
                new FileInputStream(chartFile), "image/png");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private PieDataset createDataset() {
        DefaultPieDataset dataset = new DefaultPieDataset();
        dataset.setValue("Istanbul", new Double(45.0));
        dataset.setValue("Ankara", new Double(15.0));
        dataset.setValue("Izmir", new Double(25.2));
        dataset.setValue("Antalya", new Double(14.8));

        return dataset;
    }

    public StreamedContent getChart() {
        return this.chart;
    }
}
```

Basically `p:graphicImage` makes any JSF chart component using JFreechart obsolete and lets you to avoid wrappers(e.g. JSF ChartCreator project which we've created in the past) to take full advantage of JFreechart API.



## Displaying a Barcode

Similar to the chart example, a barcode can be generated as well. This sample uses barbecue project for the barcode API.

```
<p:graphicImage value="#{backingBean.barcode}" />
```

```
public class BarcodeBean {

    private StreamedContent barcode;

    public BackingBean() {
        try {
            File barcodeFile = new File("dynamicbarcode");
            BarcodeImageHandler.saveJPEG(
                BarcodeFactory.createCode128("PRIMEFACES"),
                barcodeFile);
            barcode = new DefaultStreamedContent(
                new FileInputStream(barcodeFile), "image/jpeg");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public BarcodeBean getBarcode() {
        return this.barcode;
    }
}
```



## Passing Parameters

Behind the scenes, dynamic images are generated by a different request whose format is defined initially by the `graphicImage`. Suppose you want to generate different images depending on a request parameter. Problem is the request parameter can only be available at initial load of page containing the `graphicImage`, you'd lose the value of the parameter for the actual request that generates the image. To solve this, you can pass request parameters to the `graphicImage` via `f:param` tags, as a result the actual request rendering the image can have access to these values.

## Displaying Regular Images

As `GraphicImage` extends standard `graphicImage` component, it can also display regular non dynamic images.

```
<p:graphicImage value="barcalogo.jpg" />
```

## 3.40 GraphicText

GraphicText can convert any text to an image format in runtime.

### Info

Tag	<b>graphicText</b>
Tag Class	<b>org.primefaces.component.graphictext.GraphicTextTag</b>
Component Class	<b>org.primefaces.component.graphictext.GraphicText</b>
Component Type	<b>org.primefaces.component.GraphicText</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.GraphicTextRenderer</b>
Renderer Class	<b>org.primefaces.component.graphictext.GraphicTextRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Text value to render as an image
fontName	Verdana	String	Name of the font.
fontStyle	plain	String	Style of the font, valid values are "bold", "italic" or "plain".
fontSize	12	Integer	Size of the font.
alt	null	String	Alternate text for the image
url	null	String	Alias to value attribute
title	null	String	Title of the image
style	null	String	Style of the image
styleClass	null	String	Style class of the image
onclick	null	String	onclick dom event handler
ondblclick	null	String	ondblclick dom event handler

Name	Default	Type	Description
onkeydown	null	String	onkeydown dom event handler
onkeypress	null	String	onkeypress dom event handler
onkeyup	null	String	onkeyup dom event handler
onmousedown	null	String	onmousedown dom event handler
onmousemove	null	String	onmousemove dom event handler
onmouseout	null	String	onmouseout dom event handler
onmouseover	null	String	onmouseover dom event handler
onmouseup	null	String	onmouseup dom event handler

## Getting started with GraphicText

GraphicText only requires the text value to display.

```
<p:graphicText value="PrimeFaces" />
```

## Font Settings

Font of the text in generated image is configured via font\* attributes.

```
<p:graphicText value="PrimeFaces" fontName="Arial" fontSize="14"
                fontStyle="bold"/>
```

## 3.41 Growl

Growl is based on the Mac's growl notification widget and used to display FacesMessages similar to h:messages.



### Info

Tag	<b>growl</b>
Component Class	<b>org.primefaces.component.growl.Growl</b>
Component Type	<b>org.primefaces.component.Growl</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.GrowlRenderer</b>
Renderer Class	<b>org.primefaces.component.growl.GrowlRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
sticky	FALSE	Boolean	Specifies if the message should stay instead of hidden automatically.
showSummary	TRUE	Boolean	Specifies if the summary of message should be displayed.
showDetail	FALSE	Boolean	Specifies if the detail of message should be displayed.
globalOnly	FALSE	Boolean	When true, only facesmessages without clientids are displayed.
life	6000	Integer	Duration in milliseconds to display non-sticky messages.

Name	Default	Type	Description
warnIcon	null	String	Image of the warning messages.
infoIcon	null	String	Image of the info messages.
errorIcon	null	String	Image of the error messages.
fatalIcon	null	String	Image of the fatal messages.

## Getting Started with Growl

Growl is a replacement of h:messages and usage is very similar indeed. Simply place growl anywhere on your page, since messages are displayed as an overlay, the location of growl in JSF page does not matter.

```
<p:growl />
```

## Lifetime of messages

By default each message will be displayed for 6000 ms and then hidden. A message can be made sticky meaning it'll never be hidden automatically.

```
<p:growl sticky="true" />
```

If growl is not working in sticky mode, it's also possible to tune the duration of displaying messages. Following growl will display the messages for 5 seconds and then fade-out.

```
<p:growl life="5000" />
```

## Growl with Ajax Updates

If you need to display messages with growl after an ajax request you just need to update it.

```
<p:growl id="messages"/>
<p:commandButton value="Submit" update="messages" />
```

## Positioning

Growl is positioned at top right corner by default, position can be controlled with a CSS selector called *ui-growl*.

```
.ui-growl {
    left:20px;
}
```

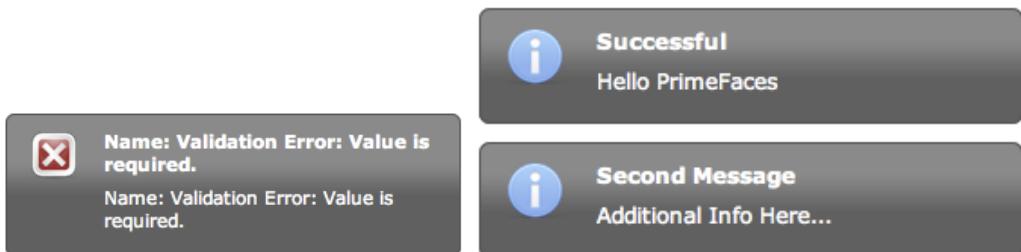
With this setting growl will be located at top left corner.

## Skinning

Following is the list of structural style classes;

Style Class	Applies
.ui-growl	Main container element of growl
.ui-growl-item-container	Container of messages
.ui-growl-item	Container of a message
.ui-growl-image	Severity icon
.ui-growl-message	Text message container
.ui-growl-title	Summary of the message
.ui-growl-message p	Detail of the message

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.42 HotKey

HotKey is a generic key binding component that can bind any formation of keys to javascript event handlers or ajax calls.

### Info

Tag	<b>hotkey</b>
Component Class	<b>org.primefaces.component.hotkey.HotKey</b>
Component Type	<b>org.primefaces.component.HotKey</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.HotKeyRenderer</b>
Renderer Class	<b>org.primefaces.component.hotkey.HotKeyRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
bind	null	String	The Key binding.
handler	null	String	Javascript event handler to be executed when the key binding is pressed.
action	null	MethodExpr	A method expression that'd be processed in the partial request caused by uiajax.
actionListener	null	MethodExpr	An actionlistener that'd be processed in the partial request caused by uiajax.
immediate	FALSE	Boolean	Boolean value that determines the phaseId, when true actions are processed at apply_request_values, when false at invoke_application phase.
async	FALSE	Boolean	When set to true, ajax requests are not queued.
process	null	String	Component id(s) to process partially instead of whole view.
update	null	String	Client side id of the component(s) to be updated after async partial submit request.

Name	Default	Type	Description
onstart	null	String	Javascript handler to execute before ajax request is begins.
oncomplete	null	String	Javascript handler to execute when ajax request is completed.
onsuccess	null	String	Javascript handler to execute when ajax request succeeds.
onerror	null	String	Javascript handler to execute when ajax request fails.
global	TRUE	Boolean	Global ajax requests are listened by ajaxStatus component, setting global to false will not trigger ajaxStatus.

## Getting Started with HotKey

HotKey is used in two ways, either on client side with the event handler or with ajax support. Simplest example would be;

```
<p:hotkey bind="a" handler="alert('Pressed a');" />
```

When this hotkey is on page, pressing the a key will alert the ‘Pressed key a’ text.

## Key combinations

Most of the time you'd need key combinations rather than a single key.

```
<p:hotkey bind="ctrl+s" handler="alert('Pressed ctrl+s');" />
```

```
<p:hotkey bind="ctrl+shift+s" handler="alert('Pressed ctrl+shift+s');" />
```

## Integration

Here's an example demonstrating how to integrate hotkeys with a client side api. Using left and right keys will switch the images displayed via the p:imageSwitch component.

```
<p:hotkey bind="left" handler="switcher.previous();" />
<p:hotkey bind="right" handler="switcher.next();" />

<p:imageSwitch widgetVar="switcher">
    //content
</p:imageSwitch>
```

## Ajax Support

Ajax is a built-in feature of hotKeys meaning you can do ajax calls with key combinations. Following form can be submitted with the *ctrl+shift+s* combination.

```
<h:form>

    <p:hotkey bind="ctrl+shift+s" update="display" />

    <h:panelGrid columns="2">
        <h:outputLabel for="name" value="Name:> />
        <h:inputText id="name" value="#{bean.name}" />
    </h:panelGrid>

    <h:outputText id="display" value="Hello: #{bean.firstname}" />

</h:form>
```

Note that hotkey must be nested inside a form to use the ajax support. We're also planning to add built-in hotkey support for p:commandButton and p:commandLink since hotkeys are a common use case for command components.

## 3.43 IdleMonitor

IdleMonitor watches user actions on a page and notify several callbacks in case they go idle or active again.

**Source**

```

01. <p:idleMonitor timeout="10000" />
02.
03. <p:dialog header="What's happening?" 
04.   " widgetVar="idleDialog" modal="true" fixedCenter="true" close="false"
05.   width="400px">
06.     <h:outputText value="Dude, are you there?" />
07.   </p:dialog>

```

### Info

Tag	<b>idleMonitor</b>
Component Class	<b>org.primefaces.component.idlemonitor.IdleMonitor</b>
Component Type	<b>org.primefaces.component.IdleMonitor</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.IdleMonitorRenderer</b>
Renderer Class	<b>org.primefaces.component.idlemonitor.IdleMonitor</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
timeout	300000	Integer	Time to wait in milliseconds until deciding if the user is idle. Default is 5 minutes.
onidle	null	String	Client side callback to execute when user goes idle.
onactive	null	String	Client side callback to execute when user goes active.

Name	Default	Type	Description
idleListener	null	MethodExpr	Server side event to be executed in case user goes idle.
update	null	String	Component(s) to update after processing a server side IdleEvent.
widgetVar	null	String	Name of the client side widget.

## Getting Started with IdleMonitor

To begin with, you can listen to events that are called when a user goes idle or becomes active again. Example below displays a warning dialog onidle and hides it back when user moves the mouse or uses the keyboard.

```
<p:idleMonitor onidle="idleDialog.show();" onactive="idleDialog.hide();"/>

<p:dialog header="What's happening?" widgetVar="idleDialog" modal="true"
    fixedCenter="true" close="false" width="400px" visible="true">
    <h:outputText value="Dude, are you there?" />
</p:dialog>
```

## Controlling Timeout

By default, idleMonitor waits for 5 minutes (300000 ms) until triggering the onidle event. You can customize this duration with the timeout attribute.

### IdleListener

Most of the time you may need to be notified on server side as well about IdleEvents so that necessary actions like invalidating the session or logging can be done. For this purpose use the idleListeners that are notified with ajax. A conventional idleEvent is passed as parameter to the idleListener.

```
<p:idleMonitor idleListener="#{idleMonitorController.handleIdle}"/>
```

handleIdle is a simple method that's defined in idleMonitorController bean.

```
public void handleIdle(IdleEvent event) {
    //Invalidate user
}
```

## AJAX Update

IdleMonitor uses PrimeFaces PPR to update the dom with the server response after an idleListener is notified. Example below adds a message and updates an outputText.

```
<h:form>
    <p:idleMonitor idleListener="#{idleMonitorController.handleIdle}"
        update="msgs"/>

    <p:messages id="msgs" />
</h:form>
```

```
public class IdleMonitorController {

    public void idleListener(IdleEvent event) {
        //Add facesmessage
    }
}
```

## 3.44 ImageCompare

ImageCompare provides a rich user interface to compare two images.



### Info

Tag	<b>imageCompare</b>
Component Class	<b>org.primefaces.component.imagecompare.ImageCompare</b>
Component Type	<b>org.primefaces.component.ImageCompare</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.ImageCompareRenderer</b>
Renderer Class	<b>org.primefaces.component.imagecompare.ImageCompareRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean

Name	Default	Type	Description
leftImage	null	String	Source of the image placed on the left side
rightImage	null	String	Source of the image placed on the right side
width	null	String	Width of the images
height	null	String	Height of the images
style	null	String	Style of the image container element
styleClass	null	String	Style class of the image container element

## Getting started with ImageCompare

ImageCompare is created with two images with same height and width.

```
<p:imageCompare leftImage="xbox.png" rightImage="ps3.png"
                 width="438" height="246"/>
```

It is required to always set width and height of the images.

## Skinning

Both images are placed inside a div container element, *style* and *styleClass* attributes apply to this element.

## 3.45 ImageCropper

ImageCropper allows cropping a certain region of an image. A new image is created containing the cropped area and assigned to a CroppedImage instanced on the server side.



### Info

Tag	<b>imageCropper</b>
Component Class	<b>org.primefaces.component.imagecropper.ImageCropper</b>
Component Type	<b>org.primefaces.component.ImageCropper</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.ImageCropperRenderer</b>
Renderer Class	<b>org.primefaces.component.imagecropper.ImageCropperRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Value of the component than can be either an EL expression of a literal text
converter	null	Converter/ String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id

Name	Default	Type	Description
immediate	FALSE	Boolean	When set true, process validations logic is executed at apply request values phase for this component.
required	FALSE	Boolean	Marks component as required
validator	null	MethodExpr	A method binding expression that refers to a method validationg the input
valueChangeListener	null	ValueChange Listener	A method binding expression that refers to a method for handling a valuchangeevent
requiredMessage	null	String	Message to be displayed when required field validation fails.
converterMessage	null	String	Message to be displayed when conversion fails.
validatorMessage	null	String	Message to be displayed when validation fields.
image	null	String	Context relative path to the image.
widgetVar	null	String	Name of the client side widget.
alt	null	String	Alternate text of the image.
aspectRatio	null	Double	Aspect ratio of the cropper area.
minSize	null	String	Minimum size of the cropper area.
maxSize	null	String	Maximum size of the cropper area.
backgroundColor	null	String	Background color of the container.
backgroundOpacity	0.6	Double	Background opacity of the container
initialCoords	null	String	Initial coordinates of the cropper area.

## Getting started with the ImageCropper

Image to be cropped is provided via the *image* attribute. ImageCropper is an input component and the cropped area of the original image is used to create a new image, this new image can be accessed on the server side jsf backing bean by setting the value attribute of the image cropper. Assuming the image is at %WEBAPP\_ROOT%/campnou.jpg

```
<p:imageCropper value="#{cropper.croppedImage}" image="/campnou.jpg" />
```

```
public class Cropper {
    private CroppedImage croppedImage;

    //getter and setter
}
```

*org.primefaces.model.CroppedImage* belongs a PrimeFaces api and contains handy information about the crop process. Following table describes CroppedImage properties.

Property	Type	Description
originalFileName	String	Name of the original file that's cropped
bytes	byte[]	Contents of the cropped area as a byte array
left	int	Left coordinate
right	int	Right coordinate
width	int	Width of the cropped image
height	int	Height of the cropped image

## External Images

ImageCropper has the ability to crop external images as well.

```
<p:imageCropper value="#{cropper.croppedImage}"  
    image="http://primefaces.prime.com.tr/en/images/schema.png">  
</p:imageCropper>
```

## Context Relative Path

For local images, ImageCropper always requires the image path to be context relative. So to accomplish this simply just add slash ("path/to/image.png") and imagecropper will recognize it at %WEBAPP\_ROOT%/path/to/image.png. Action url relative local images are not supported.

## Initial Coordinates

By default, user action is necessary to initiate the cropper area on an image, you can specify an initial area to display on page load using *initialCoords* option in *x,y,w,h* format.

```
<p:imageCropper value="#{cropper.croppedImage}" image="/campnou.jpg"  
    initialCoords="225,75,300,125"/>
```

## Boundaries

*minSize* and *maxSize* attributes are control to limit the size of the area to crop.

```
<p:imageCropper value="#{cropper.croppedImage}" image="/campnou.jpg"
    minSize="50,100" maxSize="150,200"/>
```

## Saving Images

```
<p:imageCropper value="#{cropper.croppedImage}" image="/campnou.jpg" />
<p:commandButton value="Crop" actionListener="#{myBean.crop}" />
```

```
public class Cropper {

    private CroppedImage croppedImage;

    //getter and setter

    public String crop() {
        ServletContext servletContext = (ServletContext)
FacesContext.getCurrentInstance().getExternalContext().getContext();
        String newFileName = servletContext.getRealPath("") + File.separator +
"ui" + File.separator + "barca" + File.separator+ croppedImage.getOriginalFileName()
+ "cropped.jpg";

        FileImageOutputStream imageOutput;
        try {
            imageOutput = new FileImageOutputStream(new File(newFileName));
            imageOutput.write(croppedImage.getBytes(), 0,
croppedImage.getBytes().length);
            imageOutput.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

## 3.46 ImageSwitch

ImageSwitch component is used to enable switching between a set of images with nice effects. ImageSwitch also provides a simple client side api for flexibility.



### Info

Tag	<b>imageSwitch</b>
Component Class	<b>org.primefaces.component.imageswitch.ImageSwitch</b>
Component Type	<b>org.primefaces.component.ImageSwitch</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.ImageSwitchRenderer</b>
Renderer Class	<b>org.primefaces.component.imageswitch.ImageSwitchRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
effect	null	String	Name of the effect for transition.
speed	500	Integer	Speed of the effect in milliseconds.
slideshowSpeed	3000	Integer	Slideshow speed in milliseconds.
slideshowAuto	TRUE	Boolean	Starts slideshow automatically on page load.
style	null	String	Style of the main container.
styleClass	null	String	Style class of the main container.

## Getting started with ImageSwitch

ImageSwitch component needs a set of images to display. Provide the image collection as a set of children components.

```
<p:imageSwitch effect="FlyIn" widgetVar="imageswitch">
    <p:graphicImage value="/images/nature1.jpg" />
    <p:graphicImage value="/images/nature2.jpg" />
    <p:graphicImage value="/images/nature3.jpg" />
    <p:graphicImage value="/images/nature4.jpg" />
</p:imageSwitch>
```

Most of the time, images would be dynamic, ui:repeat is supported to implement this case.

```
<p:imageSwitch widgetVar="imageswitch">
    <ui:repeat value="#{bean.images}" var="image">
        <p:graphicImage value="#{image}" />
    </ui:repeat>
</p:imageSwitch>
```

## Slideshow or Manual

ImageSwitch is in slideShow mode by default, if you'd like manual transitions disable slideshow and use client side api to create controls.

```
<p:imageSwitch effect="FlyIn" widgetVar="imageswitch">
    //images
</p:imageSwitch>

<span onclick="imageswitch.previous();">Previous</span>
<span onclick="imageswitch.next();">Next</span>
```

## Client Side API

Method	Description
void previous()	Switches to previous image.
void next()	Switches to next image.
void startSlideshow()	Starts slideshow mode.
void stopSlideshow()	Stops slideshow mode.
void pauseSlideshow();	Pauses slideshow mode.
void toggleSlideShow()	Toggles slideshow mode.

## Effect Speed

The speed is considered in terms of milliseconds and specified via the speed attribute.

```
<p:imageSwitch effect="FlipOut" speed="150" widgetVar="imageswitch" >
    //set of images
</p:imageSwitch>
```

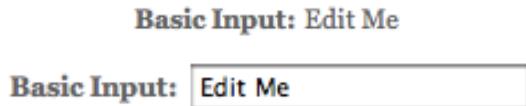
## List of Effects

ImageSwitch supports a wide range of transition effects. Following is the full list, note that values are case sensitive.

- blindX
- blindY
- blindZ
- cover
- curtainX
- curtainY
- fade
- fadeZoom
- growX
- growY
- none
- scrollUp
- scrollDown
- scrollLeft
- scrollRight
- scrollVert
- shuffle
- slideX
- slideY
- toss
- turnUp
- turnDown
- turnLeft
- turnRight
- uncover
- wipe
- zoom

## 3.47 Inplace

Inplace provides easy inplace editing and inline content display. Inplace consists of two members, display element is the initial clickable label and inline element is the hidden content that'll be displayed when display element is toggled.



### Info

Tag	<b>inplace</b>
Component Class	<b>org.primefaces.component.inplace.Inplace</b>
Component Type	<b>org.primefaces.component.Inplace</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.InplaceRenderer</b>
Renderer Class	<b>org.primefaces.component.inplace.InplaceRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
label	null	String	Label to be shown in display mode.
emptyLabel	null	String	Label to be shown in display mode when value is empty.
effect	fade	String	Effect to be used when toggling.
effectSpeed	normal	String	Speed of the effect.
disabled	FALSE	Boolean	Prevents hidden content to be shown.
style	null	String	Inline style of the main container element.
styleClass	null	String	Style class of the main container element.
editor	FALSE	Boolean	Specifies the editor mode.
saveLabel	Save	String	Tooltip text of save button in editor mode.

Name	Default	Type	Description
cancelLabel	Cancel	String	Tooltip text of cancel button in editor mode.
onEditUpdate	null	String	Component(s) to update after ajax editing.
event	click	String	Name of the client side event to display inline content.
saveListener	null	MethodExpr	Server side callback to execute during ajax editing.
widgetVar	null	String	Name of the client side widget.

## Getting Started with Inplace

The inline component needs to be a child of inplace.

```
<p:inplace>
    <h:inputText value="Edit me" />
</p:inplace>
```

## Custom Labels

By default inplace displays it's first child's value as the label, you can customize it via the label attribute.

```
<h:outputText value="Select One:" />

<p:inplace label="Cities">
    <h:selectOneMenu>
        <f:selectItem itemLabel="Istanbul" itemValue="Istanbul" />
        <f:selectItem itemLabel="Ankara" itemValue="Ankara" />
    </h:selectOneMenu>
</p:inplace>
```

Select One: Cities

Select One: **Istanbul** ▾

## Effects

Default effect is *fade* and other possible effect is *slide*, also effect speed can be tuned with values *slow*, *normal* and *fast*.

```
<p:inplace label="Show Image" effect="slide" effectSpeed="fast">
    <p:graphicImage value="/images/nature1.jpg" />
</p:inplace>
```

## Ajax Editing

Inplace has built-in support for ajax editing which is activated by *editor* option.

```
public class InplaceBean {  
  
    private String text = "PrimeFaces";  
  
    public String getText() { return this.text; }  
    public void setText(String text) { this.text = text; }  
}
```

```
<p:inplace editor="true">  
    <h:inputText value="#{inplaceBean.text}" />  
</p:inplace>
```



Save button will update the model with the new value and cancel button will revert changes. You might get notified when save button is clicked for cases like persisting changes, in this case use `saveListener`. Optionally other components on page can be update with `onEditUpdate` option, following example adds a `facesmessage` and updates `growl` component to display on save.

```
public class InplaceBean {  
  
    private String text;  
  
    public void handleSave() {  
        //add faces message with update text value  
    }  
  
    public String getText() { return this.text; }  
    public void setText(String text) { this.text = text; }  
}
```

```
<p:inplace editor="true" saveListener="#{inplaceBean.handleSave}"  
    onEditUpdate="msgs">  
    <h:inputText value="#{inplaceBean.text}" />  
</p:inplace>  
  
<p:growl id="msgs" />
```

## Client Side API

Widget: *PrimeFaces.widget.Inplace*

Method	Params	Return Type	Description
show()	-	void	Shows content and hides display element.
hide()	-	void	Shows display element and hides content.
toggle()	-	void	Toggles visibility of between content and display element.
save()	-	void	Triggers an ajax request to process inplace input.
cancel()	-	void	Triggers an ajax request to revert inplace input.

## Skinning

Inplace resides in a main container element which *style* and *styleClass* options apply.

Following is the list of structural style classes;

Style Class	Applies
.ui-inplace	Main container element.
.ui-inplace-disabled	Main container element when disabled.
.ui-inplace-display	Display element.
.ui-inplace-content	Inline content.
.ui-inplace-editor	Editor controls container.
.ui-inplace-save	Save button.
.ui-inplace-cancel	Cancel button.

As skinning style classes are global, see the main Skinning section for more information.

## 3.48 InputMask

InputMask forces an input to fit in a defined mask template.

Date:	<input type="text" value="11/12/2010"/>
Phone:	<input type="text" value="(523) 453-4253"/>
Phone with Ext:	<input type="text" value="(234) 532-4524 x35254"/>
taxId:	<input type="text" value="52-3434234"/>
SSN:	<input type="text" value="234-52-3452"/>
Product Key:	<input type="text" value="____-____-_____"/>

### Info

Tag	<b>inputMask</b>
Component Class	<b>org.primefaces.component.inputmask.InputMask</b>
Component Type	<b>org.primefaces.component.InputMask</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.InputMaskRenderer</b>
Renderer Class	<b>org.primefaces.component.inputmask.InputMaskRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
mask	null	Integer	Mask template
placeHolder	null	String	PlaceHolder in mask template.
value	null	Object	Value of the component than can be either an EL expression of a literal text

Name	Default	Type	Description
converter	null	Converter/ String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id
immediate	FALSE	Boolean	When set true, process validations logic is executed at apply request values phase for this component.
required	FALSE	Boolean	Marks component as required
validator	null	MethodExpr	A method binding expression that refers to a method validationg the input
valueChangeListener	null	MethodExpr	A method binding expression that refers to a method for handling a valuchangeevent
requiredMessage	null	String	Message to be displayed when required field validation fails.
converterMessage	null	String	Message to be displayed when conversion fails.
validatorMessage	null	String	Message to be displayed when validation fields.
widgetVar	null	String	Name of the client side widget.
accesskey	null	String	Access key that when pressed transfers focus to the input element.
alt	null	String	Alternate textual description of the input field.
autocomplete	null	String	Controls browser autocomplete behavior.
dir	null	String	Direction indication for text that does not inherit directionality. Valid values are LTR and RTL.
disabled	FALSE	Boolean	Disables input field
label	null	String	A localized user presentable name.
lang	null	String	Code describing the language used in the generated markup for this component.
maxlength	null	Integer	Maximum number of characters that may be entered in this field.
onblur	null	String	Client side callback to execute when input element loses focus.
onchange	null	String	Client side callback to execute when input element loses focus and its value has been modified since gaining focus.
onclick	null	String	Client side callback to execute when input element is clicked.
ondblclick	null	String	Client side callback to execute when input element is double clicked.

Name	Default	Type	Description
onfocus	null	String	Client side callback to execute when input element receives focus.
onkeydown	null	String	Client side callback to execute when a key is pressed down over input element.
onkeypress	null	String	Client side callback to execute when a key is pressed and released over input element.
onkeyup	null	String	Client side callback to execute when a key is released over input element.
onmousedown	null	String	Client side callback to execute when a pointer button is pressed down over input element
onmousemove	null	String	Client side callback to execute when a pointer button is moved within input element.
onmouseout	null	String	Client side callback to execute when a pointer button is moved away from input element.
onmouseover	null	String	Client side callback to execute when a pointer button is moved onto input element.
onmouseup	null	String	Client side callback to execute when a pointer button is released over input element.
onselect	null	String	Client side callback to execute when text within input element is selected by user.
readonly	FALSE	Boolean	Flag indicating that this component will prevent changes by the user.
size	null	Integer	Number of characters used to determine the width of the input element.
style	null	String	Inline style of the input element.
styleClass	null	String	Style class of the input element.
tabindex	null	Integer	Position of the input element in the tabbing order.
title	null	String	Advisory tooltip information.

## Getting Started with InputMask

InputMask below enforces input to be in 99/99/9999 date format.

```
<p:inputMask value="#{bean.field}" mask="99/99/9999" />
```

## Mask Samples

Here are more samples based on different masks;

```
<h:outputText value="Phone: " />
<p:inputMask value="#{maskController.phone}" mask="(999) 999-9999"/>

<h:outputText value="Phone with Ext: " />
<p:inputMask value="#{maskController.phoneExt}" mask="(999) 999-9999? x99999"/>

<h:outputText value="SSN: " />
<p:inputMask value="#{maskController.ssn}" mask="999-99-9999"/>

<h:outputText value="Product Key: " />
<p:inputMask value="#{maskController.productKey}" mask="a*-999-a999"/>
```

## Skinning

*style* and *styleClass* options apply to the displayed input element. As skinning style classes are global, see the main Skinning section for more information. Here's an example based on a different theme;

Date: 

## 3.49 InputText

InputText is an extension to standard inputText with skinning capabilities.



### Info

Tag	<b>inputText</b>
Component Class	<b>org.primefaces.component.inputtext.InputText</b>
Component Type	<b>org.primefaces.component.InputText</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.InputTextRenderer</b>
Renderer Class	<b>org.primefaces.component.inputtext.InputTextRender</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Value of the component than can be either an EL expression of a literal text
converter	null	Converter/String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id
immediate	FALSE	Boolean	When set true, process validations logic is executed at apply request values phase for this component.
required	FALSE	Boolean	Marks component as required
validator	null	Method Expr	A method binding expression that refers to a method validationg the input
valueChangeListener	null	Method Expr	A method binding expression that refers to a method for handling a valuchangeevent
requiredMessage	null	String	Message to be displayed when required field validation fails.

Name	Default	Type	Description
converterMessage	null	String	Message to be displayed when conversion fails.
validatorMessage	null	String	Message to be displayed when validation fields.
widgetVar	null	String	Name of the client side widget.
accesskey	null	String	Access key that when pressed transfers focus to the input element.
alt	null	String	Alternate textual description of the input field.
autocomplete	null	String	Controls browser autocomplete behavior.
dir	null	String	Direction indication for text that does not inherit directionality. Valid values are LTR and RTL.
disabled	FALSE	Boolean	Disables input field
label	null	String	A localized user presentable name.
lang	null	String	Code describing the language used in the generated markup for this component.
maxlength	null	Integer	Maximum number of characters that may be entered in this field.
onblur	null	String	Client side callback to execute when input element loses focus.
onchange	null	String	Client side callback to execute when input element loses focus and its value has been modified since gaining focus.
onclick	null	String	Client side callback to execute when input element is clicked.
ondblclick	null	String	Client side callback to execute when input element is double clicked.
onfocus	null	String	Client side callback to execute when input element receives focus.
onkeydown	null	String	Client side callback to execute when a key is pressed down over input element.
onkeypress	null	String	Client side callback to execute when a key is pressed and released over input element.
onkeyup	null	String	Client side callback to execute when a key is released over input element.
onmousedown	null	String	Client side callback to execute when a pointer button is pressed down over input element
onmousemove	null	String	Client side callback to execute when a pointer button is moved within input element.
onmouseout	null	String	Client side callback to execute when a pointer button is moved away from input element.

Name	Default	Type	Description
onmouseover	null	String	Client side callback to execute when a pointer button is moved onto input element.
onmouseup	null	String	Client side callback to execute when a pointer button is released over input element.
onselect	null	String	Client side callback to execute when text within input element is selected by user.
readonly	FALSE	Boolean	Flag indicating that this component will prevent changes by the user.
size	null	Integer	Number of characters used to determine the width of the input element.
style	null	String	Inline style of the input element.
styleClass	null	String	Style class of the input element.
tabindex	null	Integer	Position of the input element in the tabbing order.
title	null	String	Advisory tooltip information.

## Getting Started with InputText

InputText usage is same as standard inputText;

```
<p:inputText value="#{bean.propertyName}" />
```

```
public class Bean {
    private String propertyName;
    //getter and setter
}
```

## Skinning

*style* and *styleClass* options apply to the input element. As skinning style classes are global, see the main Skinning section for more information. Here's an example based on a different theme.

## 3.50 InputTextarea

InputTextarea is an extension to standard inputTextara with skinning capabilities and auto growing.



### Info

Tag	<b>inputTextarea</b>
Component Class	<b>org.primefaces.component.inputtextarea.InputTextarea</b>
Component Type	<b>org.primefaces.component.InputTextarea</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.InputTextareaRenderer</b>
Renderer Class	<b>org.primefaces.component.inputtextarea.InputTextareaRender</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Value of the component than can be either an EL expression or a literal text
converter	null	Converter/String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id
immediate	FALSE	Boolean	When set true, process validations logic is executed at apply request values phase for this component.
required	FALSE	Boolean	Marks component as required
validator	null	Method Expr	A method binding expression that refers to a method validationg the input

Name	Default	Type	Description
valueChangeListener	null	Method Expr	A method binding expression that refers to a method for handling a valuechangeevent
requiredMessage	null	String	Message to be displayed when required field validation fails.
converterMessage	null	String	Message to be displayed when conversion fails.
validatorMessage	null	String	Message to be displayed when validation fields.
widgetVar	null	String	Name of the client side widget.
autoResize	TRUE	Boolean	Specifies auto growing when being typed.
maxHeight	500	Integer	Maximum height to grow automatically.
effectDuration	200	Integer	Speed of the grow animation in milliseconds.
accesskey	null	String	Access key that when pressed transfers focus to the input element.
alt	null	String	Alternate textual description of the input field.
autocomplete	null	String	Controls browser autocomplete behavior.
dir	null	String	Direction indication for text that does not inherit directionality. Valid values are LTR and RTL.
disabled	FALSE	Boolean	Disables input field
label	null	String	A localized user presentable name.
lang	null	String	Code describing the language used in the generated markup for this component.
maxlength	null	Integer	Maximum number of characters that may be entered in this field.
onblur	null	String	Client side callback to execute when input element loses focus.
onchange	null	String	Client side callback to execute when input element loses focus and its value has been modified since gaining focus.
onclick	null	String	Client side callback to execute when input element is clicked.
ondblclick	null	String	Client side callback to execute when input element is double clicked.
onfocus	null	String	Client side callback to execute when input element receives focus.
onkeydown	null	String	Client side callback to execute when a key is pressed down over input element.
onkeypress	null	String	Client side callback to execute when a key is pressed and released over input element.

Name	Default	Type	Description
onkeyup	null	String	Client side callback to execute when a key is released over input element.
onmousedown	null	String	Client side callback to execute when a pointer button is pressed down over input element
onmousemove	null	String	Client side callback to execute when a pointer button is moved within input element.
onmouseout	null	String	Client side callback to execute when a pointer button is moved away from input element.
onmouseover	null	String	Client side callback to execute when a pointer button is moved onto input element.
onmouseup	null	String	Client side callback to execute when a pointer button is released over input element.
onselect	null	String	Client side callback to execute when text within input element is selected by user.
readonly	FALSE	Boolean	Flag indicating that this component will prevent changes by the user.
size	null	Integer	Number of characters used to determine the width of the input element.
style	null	String	Inline style of the input element.
styleClass	null	String	Style class of the input element.
tabindex	null	Integer	Position of the input element in the tabbing order.
title	null	String	Advisory tooltip information.

## Getting Started with InputTextarea

InputTextarea usage is same as standard inputTextarea;

```
<p:inputTextarea value="#{bean.propertyName}" />
```

```
public class Bean {
    private String propertyName;
    //getter and setter
}
```

## AutoResize

When textarea is being typed, if content height exceeds the allocated space, textarea can grow automatically. Use autoResize option to turn on/off this feature.

## Skinning

*style* and *styleClass* options apply to the textarea element. As skinning style classes are global, see the main Skinning section for more information. Here's an example based on a different theme.



## 3.51 Keyboard

Keyboard is an input component that uses a virtual keyboard to provide the input. Notable features are the customizable layouts and skinning capabilities.



### Info

Tag	<b>keyboard</b>
Component Class	<b>org.primefaces.component.keyboard.Keyboard</b>
Component Type	<b>org.primefaces.component.Keyboard</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.KeyboardRenderer</b>
Renderer Class	<b>org.primefaces.component.keyboard.KeyboardRenderer</b>

### Attributes

Name	Default	Type	Description
id	Assigned by JSF	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Value of the component than can be either an EL expression or a literal text
converter	null	Converter/ String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id
immediate	FALSE	Boolean	When set true, process validations logic is executed at apply request values phase for this component.

Name	Default	Type	Description
required	FALSE	Boolean	Marks component as required
validator	null	MethodExpr	A method binding expression that refers to a method validating the input
valueChangeListener	null	MethodExpr	A method binding expression that refers to a method for handling a valuechangeevent
requiredMessage	null	String	Message to be displayed when required field validation fails.
converterMessage	null	String	Message to be displayed when conversion fails.
validatorMessage	null	String	Message to be displayed when validation fields.
password	FALSE	Boolean	Makes the input a password field.
showMode	focus	String	Specifies the showMode, 'focus', 'button', 'both'
buttonImage	null	String	Image for the button.
buttonImageOnly	FALSE	boolean	When set to true only image of the button would be displayed.
effect	fadeIn	String	Effect of the display animation.
effectDuration	null	String	Length of the display animation.
layout	qwerty	String	Built-in layout of the keyboard.
layoutTemplate	null	String	Template of the custom layout.
keypadOnly	focus	Boolean	Specifies displaying a keypad instead of a keyboard.
promptLabel	null	String	Label of the prompt text.
closeLabel	null	String	Label of the close key.
clearLabel	null	String	Label of the clear key.
backspaceLabel	null	String	Label of the backspace key.
accesskey	null	String	Access key that when pressed transfers focus to the input element.
alt	null	String	Alternate textual description of the input field.
autocomplete	null	String	Controls browser autocomplete behavior.
dir	null	String	Direction indication for text that does not inherit directionality. Valid values are LTR and RTL.
disabled	FALSE	Boolean	Disables input field
label	null	String	A localized user presentable name.
lang	null	String	Code describing the language used in the generated markup for this component.

Name	Default	Type	Description
maxlength	null	Integer	Maximum number of characters that may be entered in this field.
onblur	null	String	Client side callback to execute when input element loses focus.
onchange	null	String	Client side callback to execute when input element loses focus and its value has been modified since gaining focus.
onclick	null	String	Client side callback to execute when input element is clicked.
ondblclick	null	String	Client side callback to execute when input element is double clicked.
onfocus	null	String	Client side callback to execute when input element receives focus.
onkeydown	null	String	Client side callback to execute when a key is pressed down over input element.
onkeypress	null	String	Client side callback to execute when a key is pressed and released over input element.
onkeyup	null	String	Client side callback to execute when a key is released over input element.
onmousedown	null	String	Client side callback to execute when a pointer button is pressed down over input element
onmousemove	null	String	Client side callback to execute when a pointer button is moved within input element.
onmouseout	null	String	Client side callback to execute when a pointer button is moved away from input element.
onmouseover	null	String	Client side callback to execute when a pointer button is moved onto input element.
onmouseup	null	String	Client side callback to execute when a pointer button is released over input element.
onselect	null	String	Client side callback to execute when text within input element is selected by user.
readonly	FALSE	Boolean	Flag indicating that this component will prevent changes by the user.
size	null	Integer	Number of characters used to determine the width of the input element.
style	null	String	Inline style of the input element.
styleClass	null	String	Style class of the input element.
tabindex	null	Integer	Position of the input element in the tabbing order.
title	null	String	Advisory tooltip information.

Name	Default	Type	Description
widgetVar	null	String	Name of the client side widget.

## Getting Started with Keyboard

Keyboard is used just like a simple inputText, by default when the input gets the focus a keyboard is displayed.

```
<p:keyboard value="#{bean.value}" />
```

## Built-in Layouts

There're a couple of built-in keyboard layouts these are 'qwerty', 'qwertyBasic' and 'alphabetic'. For example keyboard below has the alphabetic layout.

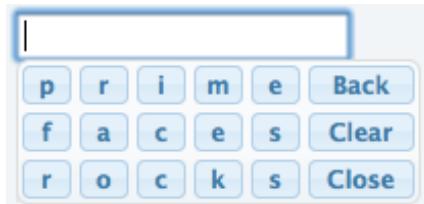
```
<p:keyboard value="#{bean.value}" layout="alphabetic"/>
```



## Custom Layouts

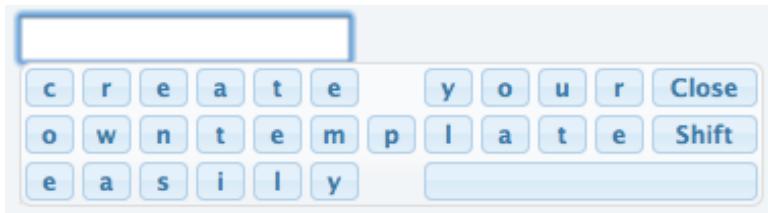
Keyboard has a very flexible layout mechanism allowing you to come up with your own layout.

```
<p:keyboard value="#{bean.value}"
    layout="custom"
    layoutTemplate="prime-back,faces-clear,rocks-close"/>
```



Another example:

```
<p:keyboard value="#{bean.value}"
    layout="custom"
    layoutTemplate="create-space-your-close,owntemplate-shift,easily-space-
spacebar"/>
```



A layout template basically consists of built-in keys and your own keys. Following is the list of all built-in keys.

- back
- clear
- close
- shift
- spacebar
- space
- halfspace

All other text in a layout is realized as separate keys so "prime" would create 5 keys as "p" "r" "i" "m" "e". Use dash to separate each member in layout and use commas to create a new row.

## Keypad

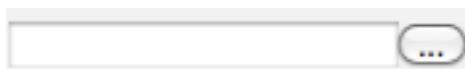
By default keyboard displays whole keys, if you only need the numbers use the keypad mode.

```
<p:keyboard value="#{bean.value}" keypadOnly="true"/>
```

## ShowMode

There're a couple of different ways to display the keyboard, by default keyboard is shown once input field receives the focus. This is customized using the showMode feature which accept values 'focus', 'button', 'both'. Keyboard below displays a button next to the input field, when the button is clicked the keyboard is shown.

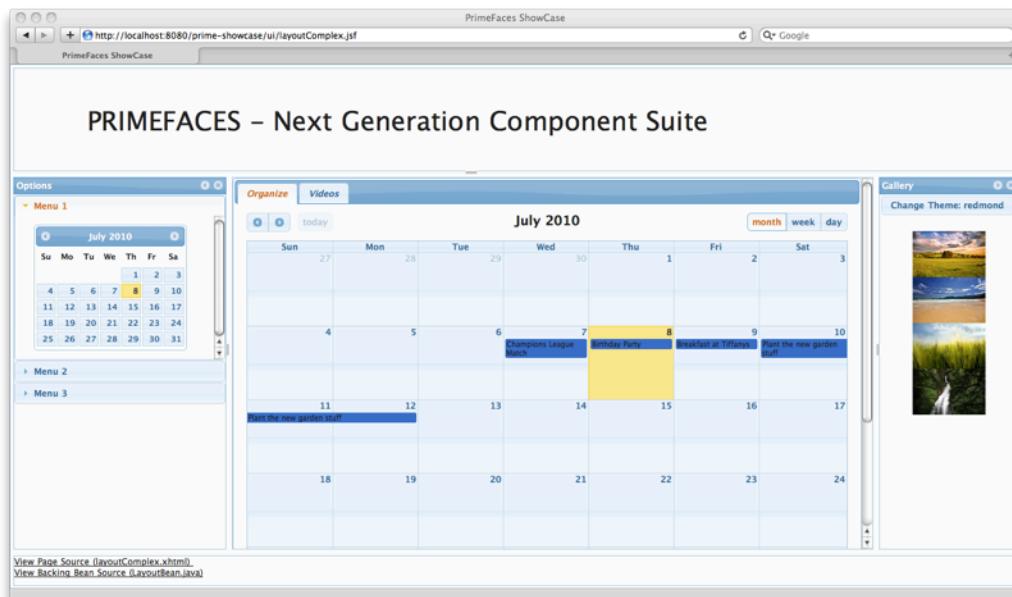
```
<p:keyboard value="#{bean.value}" showMode="button"/>
```



Button can also be customized using the *buttonImage* and *buttonImageOnly* attributes.

## 3.52 Layout

Layout component features a highly customizable borderLayout model making it very easy to create complex layouts even if you're not familiar with web design.



### Info

Tag	<b>layout</b>
Component Class	<b>org.primefaces.component.layout.Layout</b>
Component Type	<b>org.primefaces.component.Layout</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.LayoutRenderer</b>
Renderer Class	<b>org.primefaces.component.layout.LayoutRenderer</b>

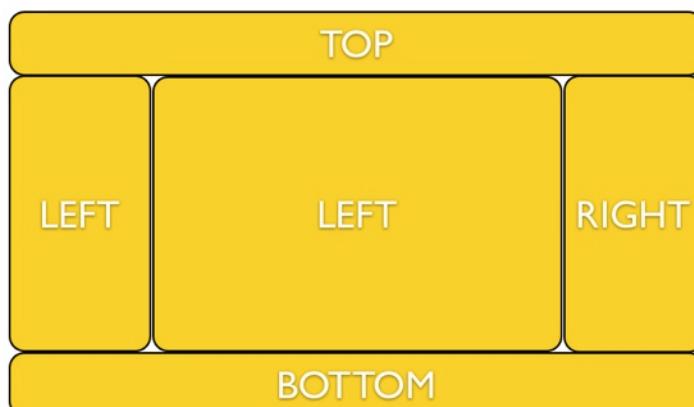
### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
widgetVar	null	String	Name of the client side widget.

Name	Default	Type	Description
fullPage	FALSE	Boolean	Specifies whether layout should span all page or not.
style	null	String	Style to apply to container element, this is only applicable to element based layouts.
styleClass	null	String	Style class to apply to container element, this is only applicable to element based layouts.
closeTitle	null	String	Title label for the close button of closable units.
collapseTitle	null	String	Title label for the collapse button of collapsible units.
expandTitle	null	String	Title label for the expand button of closable units.
closeListener	null	MethodExpr	A server side listener to process a CloseEvent
onCloseUpdate	null	String	Components to partially update with ajax after closeListener is processed and unit is closed.
toggleListener	null	MethodExpr	A server side listener to process a ToggleEvent
onToggleUpdate	null	String	Components to partially update with ajax after toggleListener is processed and unit is toggled.
resizeListener	null	MethodExpr	A server side listener to process a ResizeEvent
onResizeUpdate	null	String	Components to partially update with ajax after resizeListener is processed and unit is resized.
onToggleComplete	null	String	Client side callback for completed toggle
onCloseComplete	null	String	Client side callback for completed close
onResizeComplete	null	String	Client side callback for completed toggle

## Getting started with Layout

Layout is based on a borderLayout model that consists of 5 different layout units which are top, left, center, right and bottom. This model is visualized in the schema below;

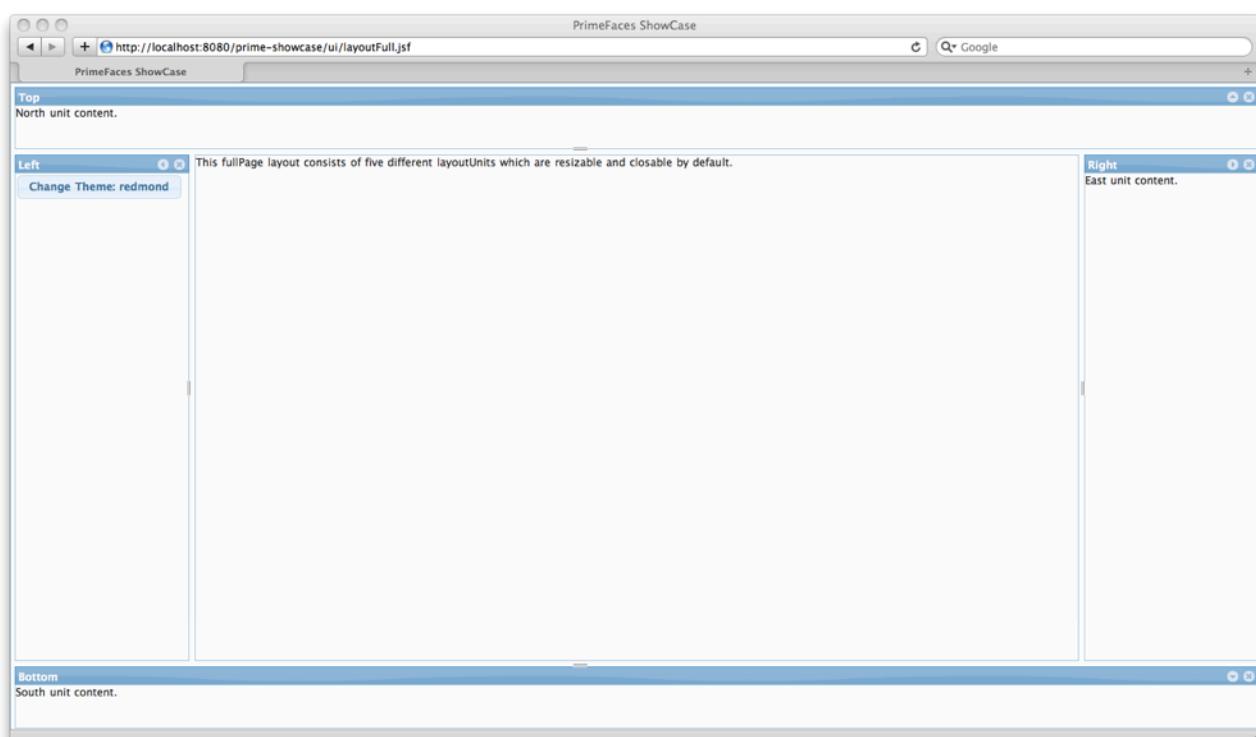


## Full Page Layout

Layout has two modes, you can either use it for a full page layout or for a specific region in your page. This setting is controlled with the `fullPage` attribute which is false by default.

The regions in a layout are defined by `layoutUnits`, following is a simple full page layout with all possible units. Note that you can place any content in each layout unit.

```
<p:layout fullPage="true">
    <p:layoutUnit position="top" header="TOP" height="50">
        <h:outputText value="Top content." />
    </p:layoutUnit>
    <p:layoutUnit position="bottom" header="BOTTOM" height="100">
        <h:outputText value="Bottom content." />
    </p:layoutUnit>
    <p:layoutUnit position="left" header="LEFT" width="300">
        <h:outputText value="Left content" />
    </p:layoutUnit>
    <p:layoutUnit position="right" header="RIGHT" width="200">
        <h:outputText value="Right Content" />
    </p:layoutUnit>
    <p:layoutUnit position="center" header="CENTER">
        <h:outputText value="Center Content" />
    </p:layoutUnit>
</p:layout>
```



## Forms in Full Page Layout

When working with forms and full page layout, avoid using a form that contains layoutunits as generated dom will not be the same. So following is **invalid**.

```
<p:layout fullPage="true">
    <h:form>
        <p:layoutUnit position="left" width="100">
            <h:outputText value="Left Pane" />
        </p:layoutUnit>
        <p:layoutUnit position="center">
            <h:outputText value="Right Pane" />
        </p:layoutUnit>
    </h:form>
</p:layout>
```

A layout unit must have it's own form instead, also avoid trying to update layoutunits because of same reason.

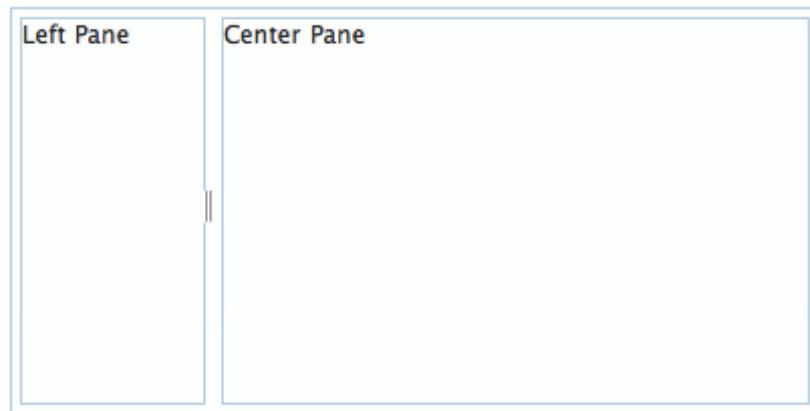
## Dimensions

Except center layoutUnit, other layout units **must** have dimensions defined. For top and bottom units use height attribute whereas for left and right units width attribute applies.

## Element based layout

Another use case of layout is the element based layout. This is the default case actually so just ignore fullPage attribute or set it to false. Layout example below demonstrates creating a split panel implementation.

```
<p:layout style="width:400px;height:200px">
    <p:layoutUnit position="left" width="100">
        <h:outputText value="Left Pane" />
    </p:layoutUnit>
    <p:layoutUnit position="center">
        <h:outputText value="Right Pane" />
    </p:layoutUnit>
</p:layout>
```



## Responding to Events

Layout can respond to toggle, close and resize events, by binding ajax listeners to these events you can have your custom logic processed easily. This is very useful if you'd like to keep the state of the layout by persisting users' preferences.

**Note:** At least one form needs to be present on page to use ajax listeners.

### ToggleEvent, ToggleListener and onToggleUpdate

Ajax toggle listener is invoked with a toggle event whenever a toggleable layout unit is collapsed or expanded. Optionally other components on page can be updated with ajax using onToggleUpdate attribute.

```
<p:layout toggleListener="#{layoutBean.handleToggle}"
          onToggleUpdate="messages">

    //Content

</p:layout>

<p:growl id="messages" />
```

```
public class LayoutBean {

    public void handleClose(ToggleEvent event) {
        LayoutUnit toggledUnit = event.getComponent();
        Visibility status = event.getVisibility();

        //...
    }
}
```

*org.primefaces.event.ToggleEvent*

Method	Description
getComponent()	Toggled layout unit instance
getVisibility()	org.primefaces.model.Visibility instance, this is an enum with two values; VISIBLE or HIDDEN.

### CloseEvent, CloseListener and onCloseUpdate

Ajax close listener is invoked with a close event whenever a closable layout unit is closed. Optionally other components on page can be updated with ajax using onCloseUpdate attribute.

```
<p:layout closeListener="#{layoutBean.handleClose}"
    onCloseListener="messages">
    //Content
</p:layout>
<p:growl id="messages" />
```

```
public void handleClose(CloseEvent event) {
    LayoutUnit closedUnit = event.getComponent();
    ...
}
```

*org.primefaces.event.CloseEvent*

Method	Description
getComponent()	Closed layout unit instance

### ResizeEvent, ResizeListener and onResizeUpdate

Ajax resize listener is invoked with a resize event whenever a resizable layout unit is resized. Optionally other components on page can be updated with ajax using onResizeUpdate attribute.

```
<p:layout resizeListener="#{layoutBean.handleResize}"
    onResizeUpdate="messages">
    //Content
</p:layout>
<p:growl id="messages" />
```

```
public void handleResize(CloseEvent event) {
    LayoutUnit resizedUnit = event.getComponent();
    //...
}
```

*org.primefaces.event.ResizeEvent*

Method	Description
getComponent()	Resized layout unit instance
getWidth()	New width in pixels
getHeight()	New height in pixels

## Stateful Layout

Making layout stateful would be easy, once you create your data to store the user preference, you can update this data using ajax event listeners provided by layout. For example if a layout unit is collapsed, you can save and persist this information. By binding this persisted information to the collapsed attribute of the layout unit layout will be rendered as the user left it last time. Similarly visible, width and height attributes can be preconfigures using same approach.

## Skinning

Following is the list of structural style classes;

Style Class	Applies
.ui-layout	Main wrapper container element
.ui-layout-doc	Layout container
.ui-layout-unit	Each layout unit container
.ui-layout-unit-{position}	Position based layout unit
.ui-layout-wrap	Wrapper of a layoutunit
.ui-layout-hd	Layout unit header
.ui-layout-bd	Layout unit body

As skinning style classes are global, see the main Skinning section for more information.

## 3.53 LayoutUnit

LayoutUnit represents a region in the border layout model of the Layout component.

### Info

Tag	<b>layoutUnit</b>
Component Class	<b>org.primefaces.component.layout.LayoutUnit</b>
Component Type	<b>org.primefaces.component.LayoutUnit</b>
Component Family	<b>org.primefaces.component</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
position	null	String	Position of the unit.
width	null	Integer	Width of the unit in pixels, applies to left and right units.
height	null	Integer	Height of the unit in pixels, applies to top and center units.
resizable	FALSE	Boolean	Makes the unit resizable.
closable	FALSE	Boolean	Makes the unit closable.
collapsible	FALSE	Boolean	Makes the unit collapsible.
scrollable	FALSE	Boolean	Makes the unit scrollable.
header	null	String	Text of header.
footer	null	String	Text of footer.
minWidth	null	Integer	Minimum dimension of width in resizing
maxWidth	null	Integer	Maximum dimension of width in resizing
minHeight	null	Integer	Minimum dimension of height in resizing
maxHeight	null	Integer	Maximum dimension of height in resizing
gutter	4px	String	Gutter size of layout unit.

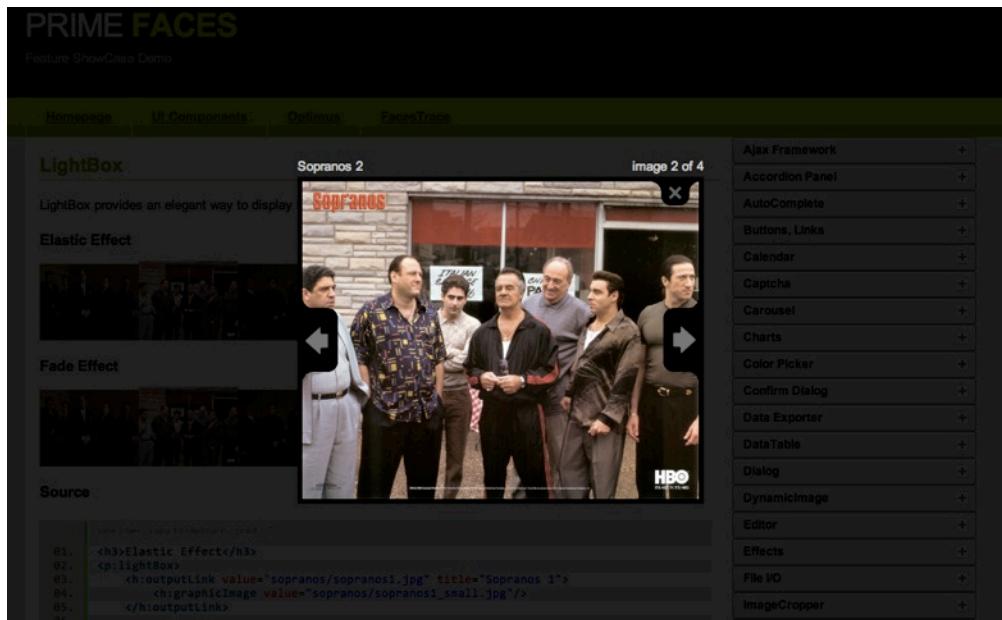
Name	Default	Type	Description
zindex	null	Integer	zindex property to control overlapping with other components
visible	TRUE	Boolean	Specifies default visibility
collapsed	FALSE	Boolean	Specifies toggle status of unit
proxyResize	TRUE	Boolean	Specifies resize preview mode
collapseSize	null	Integer	Size of the unit when collapsed

## Getting started with LayoutUnit

See layout component documentation for more information regarding the usage of layoutUnits.

## 3.54 LightBox

Lightbox features a powerful overlay that can display images, multimedia content, other JSF components and external urls.



### Info

Tag	<b>lightBox</b>
Component Class	<b>org.primefaces.component.lightbox.LightBox</b>
Component Type	<b>org.primefaces.component.LightBox</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.LightBoxRenderer</b>
Renderer Class	<b>org.primefaces.component.lightbox.LightBoxRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
style	null	String	Style of the container element not the overlay element.

Name	Default	Type	Description
styleClass	null	String	Style class of the container element not the overlay element.
widgetVar	null	String	Javascript variable name of the client side widget
transition	elastic	String	Name of the transition effect. Valid values are 'elastic','fade' and 'none'.
speed	350	Integer	Speed of the transition effect in milliseconds.
width	null	String	Width of the overlay.
height	null	String	Height of the overlay.
iframe	FALSE	Boolean	Specifies an iframe to display an external url in overlay.
opacity	0.85	Double	Level of overlay opacity between 0 and 1.
visible	FALSE	Boolean	Displays lightbox without requiring any user interaction by default.
slideshow	FALSE	Boolean	Displays lightbox without requiring any user interaction by default.
slideshowSpeed	2500	Integer	Speed for slideshow in milliseconds.
slideshowStartText	null	String	Label of slideshow start text.
slideshowStopText	null	String	Label of slideshow stop text.
slideshowAuto	TRUE	Boolean	Starts slideshow automatically.
currentTemplate	null	String	Text template for current image display like "1 of 3". Default is "{current} of {total}".
overlayClose	TRUE	Boolean	When true clicking outside of overlay will close lightbox.
group	TRUE	Boolean	Defines grouping, by default children belong to same group and switching is enabled.

## Images

The images displayed in the lightBox need to be nested as child outputLink components. Following lightBox is displayed when any of the links are clicked.

```
<p:lightbox>
    <h:outputLink value="sopranos/sopranos1.jpg" title="Sopranos 1">
        <h:graphicImage value="sopranos/sopranos1_small.jpg"/>
    </h:outputLink>

    <h:outputLink value="sopranos/sopranos2.jpg" title="Sopranos 2">
        <h:graphicImage value="sopranos/sopranos2_small.jpg" />
    </h:outputLink>

    <h:outputLink value="sopranos/sopranos3.jpg" title="Sopranos 3">
        <h:graphicImage value="sopranos/sopranos3_small.jpg"/>
    </h:outputLink>

    <h:outputLink value="sopranos/sopranos4.jpg" title="Sopranos 4">
        <h:graphicImage value="sopranos/sopranos4_small.jpg"/>
    </h:outputLink>
</p:lightbox>
```

Output of this lightbox is;

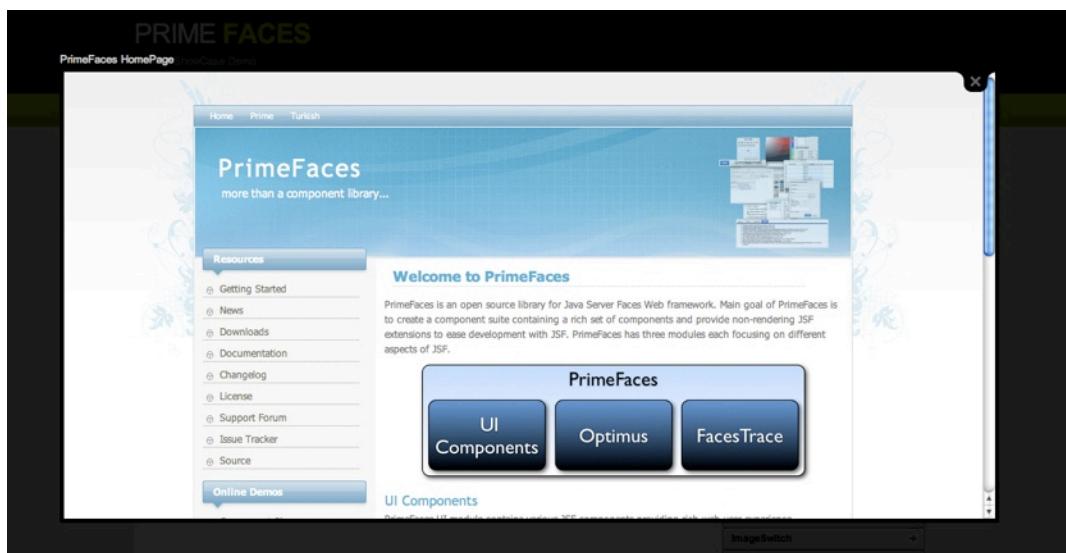


## IFrame Mode

LightBox also has the ability to display iframes inside the page overlay, following lightbox displays the PrimeFaces homepage when the link inside is clicked.

```
<p:lightbox iframe="true" width="80%" height="80%">
    <h:outputLink value="http://www.primefaces.org"
        title="PrimeFaces
    HomePage">
        <h:outputText value="PrimeFaces HomePage"/>
    </h:outputLink>
</p:lightbox>
```

Clicking the outputLink will display PrimeFaces homepage within an iframe.

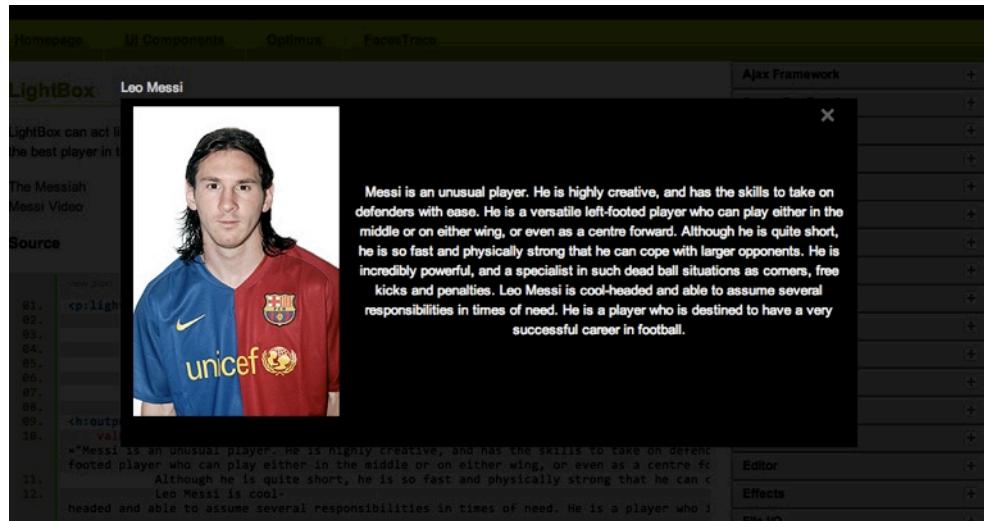


## Inline Mode

Inline mode acts like a modal panel, you can display other JSF content on the page using the lightbox overlay. Simply place your overlay content in the "inline" facet. Clicking the link in the example below will display the panelGrid contents in overlay.

```
<p:lightBox width="50%" height="50%">
    <h:outputLink value="#" title="Leo Messi" >
        <h:outputText value="The Messiah"/>
    </h:outputLink>

    <f:facet name="inline">
        <h:panelGrid columns="2">
            <h:graphicImage value="barca/messi.jpg" />
        <h:outputText style="color:#FFFFFF"
            value="Messi is an unusual player....." />
        </h:panelGrid>
    </f:facet>
</p:lightBox>
```



## SlideShow

If you want to use lightbox images as a slideshow, turn slideshow setting to true.

```
<p:lightbox slideshow="true" slideshowSpeed="2000"
    slideshowStartText="Start" slideshowStopText="Stop">
    <h:outputLink value="sopranos/sopranos1.jpg" title="Sopranos 1">
        <h:graphicImage value="sopranos/sopranos1_small.jpg"/>
    </h:outputLink>

    <h:outputLink value="sopranos/sopranos2.jpg" title="Sopranos 2">
        <h:graphicImage value="sopranos/sopranos2_small.jpg" />
    </h:outputLink>
</p:lightbox>
```

## Tips

- If lightbox is not working, it may be due to lack of DOCTYPE declaration.

## 3.55 Media

Media component is used for embedding multimedia content such as videos and music to JSF views. Media renders `<object />` or `<embed />` html tags depending on the user client.

### Info

Tag	<b>media</b>
Component Class	<b>org.primefaces.component.media.Media</b>
Component Type	<b>org.primefaces.component.Media</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.MediaRenderer</b>
Renderer Class	<b>org.primefaces.component.media.MediaRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
value	null	String	Media source to play.
player	null	String	Type of the player, possible values are "quicktime","windows","flash","real".
width	null	String	Width of the player.
height	null	String	Height of the player.
style	null	String	Style of the player.
styleClass	null	String	StyleClass of the player.

### Getting started with Media

In it's simplest form media component requires a source to play, this is defined using the `value` attribute.

```
<p:media value="/media/ria_with_primefaces.mov" />
```

## Player Types

By default, players are identified using the value extension so for instance mov files will be played by quicktime player. You can customize which player to use with the player attribute.

```
<p:media value="http://www.youtube.com/v/ABCDEFGH" player="flash"/>
```

Following is the supported players and file types.

Player	Types
windows	asx, asf, avi, wma, wmv
quicktime	aif, aiff, aac, au, bmp, gsm, mov, mid, midi, mpg, mpeg, mp4, m4a, psd, qt, qtif, qif, qti, snd, tif, tiff, wav, 3g2, 3pg
flash	flv, mp3, swf
real	ra, ram, rm, rpm, rv, smi, smil

## Parameters

Different proprietary players might have different configuration parameters, these can be specified using f:param tags.

```
<p:media value="/media/ria_with_primefaces.mov">
    <f:param name="param1" value="value1" />
    <f:param name="param2" value="value2" />
</p:media>
```

## StreamedContent Support

Media component can also play binary media content, example for this use case is storing media files in database using binary format. In order to implement this, bind a StreamedContent.

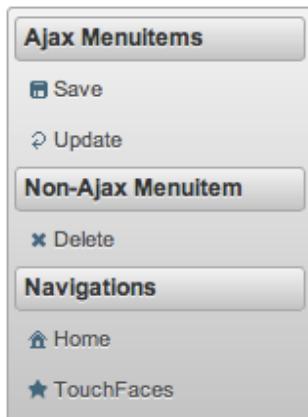
```
<p:media value="#{mediaBean.media}" width="250" height="225" player="quicktime"/>
```

```
public class MediaBean {
    private StreamedContent media;

    public MediaController() {
        InputStream stream = //Create binary stream from database
        media = new DefaultStreamedContent(stream, "video/quicktime");
    }
    public StreamedContent getMedia() { return media; }
}
```

## 3.56 Menu

Menu is a navigation component with various customized modes like multi tiers, ipod style sliding and overlays.



### Info

Tag	<b>menu</b>
Component Class	<b>org.primefaces.component.menu.Menu</b>
Component Type	<b>org.primefaces.component.Menu</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.MenuRenderer</b>
Renderer Class	<b>org.primefaces.component.menu.MenuRenderer</b>

### Attributes

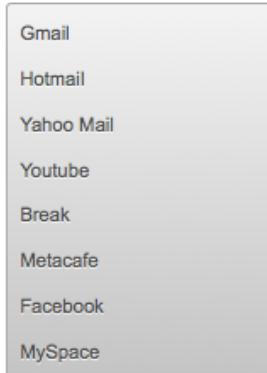
Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
widgetVar	null	String	Name of the client side widget.
model	null	MenuModel	A menu model instance to create menu programmatically.
trigger	null	String	Id of component whose click event will show the dynamic positioned menu.
my	null	String	Corner of menu to align with trigger element.

Name	Default	Type	Description
at	null	String	Corner of trigger to align with menu element.
position	static	String	Defines positioning type of menu, either static or dynamic.
tiered	FALSE	Boolean	(Deprecated: use type as tiered instead) Enabled nested tiered menus.
type	plain	String	Type of menu, valid values are <i>plain</i> , <i>tiered</i> and <i>sliding</i> .
effect	fade	String	Effect to use when showing overlay menus, valid values are <i>fade</i> and <i>slide</i> .
effectDuration	400	Integer	Duration of animation in milliseconds.
style	null	String	Inline style of the main container element.
styleClass	null	String	Style class of the main container element.
zindex	1	Integer	z-index property to control overlapping with other elements.
backLabel	Back	String	Text for back link, only applies to sliding menus.
maxHeight	200	Integer	Maximum height for menu, only applies to sliding menu.

## Getting started with the Menu

A menu is composed of submenus and menuitems.

```
<p:menu>
    <p:menuitem value="Gmail" url="http://www.google.com" />
    <p:menuitem value="Hotmail" url="http://www.hotmail.com" />
    <p:menuitem value="Yahoo Mail" url="http://mail.yahoo.com" />
    <p:menuitem value="Youtube" url="http://www.youtube.com" />
    <p:menuitem value="Break" url="http://www.break.com" />
    <p:menuitem value="Metacafe" url="http://www.metacafe.com" />
    <p:menuitem value="Facebook" url="http://www.facebook.com" />
    <p:menuitem value="MySpace" url="http://www.myspace.com" />
</p:menu>
```

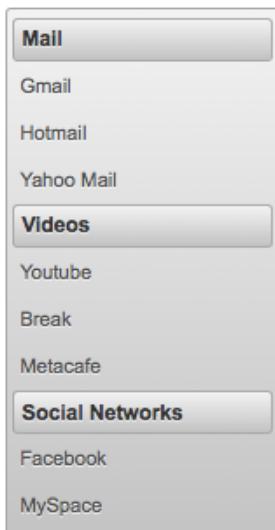


Submenus are used to group menuitems;

```
<p:menu>
    <p:submenu label="Mail">
        <p:menuitem value="Gmail" url="http://www.google.com" />
        <p:menuitem value="Hotmail" url="http://www.hotmail.com" />
        <p:menuitem value="Yahoo Mail" url="http://mail.yahoo.com" />
    </p:submenu>

    <p:submenu label="Videos">
        <p:menuitem value="Youtube" url="http://www.youtube.com" />
        <p:menuitem value="Break" url="http://www.break.com" />
        <p:menuitem value="Metacafe" url="http://www.metacafe.com" />
    </p:submenu>

    <p:submenu label="Social Networks">
        <p:menuitem value="Facebook" url="http://www.facebook.com" />
        <p:menuitem value="MySpace" url="http://www.myspace.com" />
    </p:submenu>
</p:menu>
```



## Overlay Menu

Menu can be positioned on a page in two ways; "static" and "dynamic". By default it's static meaning the menu is in normal page flow. In contrast dynamic menus is not on the normal flow of the page allowing them to overlay other elements.

A dynamic menu is created by setting position option to dynamic and defining a trigger to show the menu. Location of menu on page will be relative to the trigger and defined by my and at options that take combination of four values;

- left
- right
- bottom
- top

For example, clicking the button below will display the menu whose top left corner is aligned with bottom left corner of button.

```
<p:menu position="dynamic" trigger="btn" my="left top" at="bottom left">
    ...submenus and menuitems
</p:menu>

<p:commandButton id="btn" value="Show Menu" type="button"/>
```

## Menu Types

Menu has three different types, *plain*, *tiered* and *sliding*.

```
<p:menu type="plain|tiered|sliding">
    ...submenus and menuitems
</p:menu>
```

Plain	Tiered	Sliding (iPod)

## Ajax and Non-Ajax Actions

As menu uses menuitems, it is easy to invoke actions with or without ajax as well as navigation. See menuitem documentation for more information about the capabilities.

```
<p:menu>
    <p:submenu label="Options">
        <p:menuitem value="Save" actionListener="#{bean.save}" update="comp"/>
        <p:menuitem value="Update" actionListener="#{bean.update}" ajax="false"/>
        <p:menuitem value="Navigate" url="http://www.primefaces.org"/>
    </p:submenu>
</p:menu>
```

## Effects

Menu has a built-in animation to use when displaying&hiding itself and it's submenus. This animation is customizable using attributes like *effect* and *effectDuration*. Available animations are *fade* or *slide*, effectDuration is defined in milliseconds defaulting to 400.

## Dynamic Menus

Menus can be created programmatically as well, this is more flexible compared to the declarative approach. Menu metadata is defined using an *org.primefaces.model.MenuModel* instance, PrimeFaces provides the built-in *org.primefaces.model.DefaultMenuModel* implementation. For further customization you can also create and bind your own MenuModel implementation.

```
<p:menu model="#{menuBean.model}" />
```

```
public class MenuBean {

    private MenuModel model;

    public MenuBean() {
        model = new DefaultMenuModel();

        //First submenu
        Submenu submenu = new Submenu();
        submenu.setLabel("Dynamic Submenu 1");

        MenuItem item = new MenuItem();
        item.setValue("Dynamic MenuItem 1.1");
        item.setUrl("#");
        submenu.getChildren().add(item);

        model.addSubmenu(submenu);

        //Second submenu
        submenu = new Submenu();
        submenu.setLabel("Dynamic Submenu 2");

        item = new MenuItem();
        item.setValue("Dynamic MenuItem 2.1");
        item.setUrl("#");
        submenu.getChildren().add(item);

        item = new MenuItem();
        item.setValue("Dynamic MenuItem 2.2");
        item.setUrl("#");
        submenu.getChildren().add(item);

        model.addSubmenu(submenu);
    }

    public MenuModel getModel() { return model; }
}
```

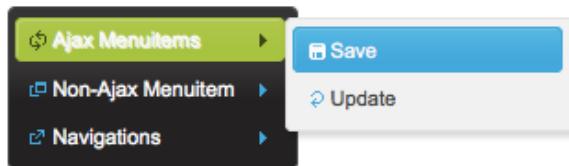
## Skinning

Menu resides in a main container element which *style* and *styleClass* attributes apply.

Following is the list of structural style classes;

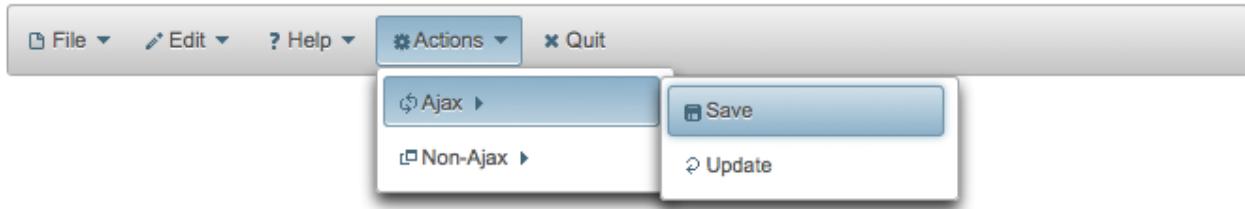
Style Class	Applies
.wijmo-wijmenu	Container element of menu
.wijmo-wijmenu-list	List container
.wijmo-wijmenu-item	Each menu item
.wijmo-wijmenu-link	Anchor element in a link item
.wijmo-wijmenu-text	Text element in an item
.wij-menu-ipod	Container of ipod like sliding menu
.wij-menu-breadcrumb	Container of ipod like navigation menu controls

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.57 Menubar

Menubar is a horizontal navigation component.



### Info

Tag	<b>menubar</b>
Component Class	<b>org.primefaces.component.menubar.MenuBar</b>
Component Type	<b>org.primefaces.component.MenuBar</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.MenuBarRenderer</b>
Renderer Class	<b>org.primefaces.component.menubar.MenuBarRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
effect	fade	String	Sets the effect for the menu display.
effectDuration	400	Integer	Sets the effect duration in milliseconds.
autoSubMenuDisplay	FALSE	Boolean	When set to true, submenus are displayed on mouseover of a menuitem.
widgetVar	null	String	Name of the client side widget
model	null	MenuModel	MenuModel instance to create menus programmatically
style	null	String	Inline style of menubar
styleClass	null	String	Style class of menubar

## Getting started with Menubar

Submenus and menuitems as child components are required to compose the menubar.

```
<p:menubar>
    <p:submenu label="Mail">
        <p:menuitem value="Gmail" url="http://www.google.com" />
        <p:menuitem value="Hotmail" url="http://www.hotmail.com" />
        <p:menuitem value="Yahoo Mail" url="http://mail.yahoo.com" />
    </p:submenu>
    <p:submenu label="Videos">
        <p:menuitem value="Youtube" url="http://www.youtube.com" />
        <p:menuitem value="Break" url="http://www.break.com" />
    </p:submenu>
</p:menubar>
```

## Nested Menus

To create a menubar with a higher depth, nest submenus in parent submenus.

```
<p:menubar>
    <p:submenu label="File">
        <p:submenu label="New">
            <p:menuitem value="Project" url="#" />
            <p:menuitem value="Other" url="#" />
        </p:submenu>
        <p:menuitem value="Open" url="#" /></p:menuitem>
        <p:menuitem value="Quit" url="#" /></p:menuitem>
    </p:submenu>
    <p:submenu label="Edit">
        <p:menuitem value="Undo" url="#" /></p:menuitem>
        <p:menuitem value="Redo" url="#" /></p:menuitem>
    </p:submenu>
    <p:submenu label="Help">
        <p:menuitem label="Contents" url="#" />
        <p:submenu label="Search">
            <p:submenu label="Text">
                <p:menuitem value="Workspace" url="#" />
            </p:submenu>
            <p:menuitem value="File" url="#" />
        </p:submenu>
    </p:submenu>
</p:menubar>
```

## Effects

Menu has a built-in animation to use when displaying&hiding itself and it's submenus. This animation is customizable using attributes like *effect* and *effectDuration*. Available animations are *fade* or *slide* and effectDuration is defined in milliseconds defaulting to 400.

## Root MenuItem

Menubar supports menuitem as root menu options as well; Following example allows a root menubar item to execute an action to logout the user.

```
<p:menubar>
    //submenus
    <p:menuitem label="Logout" action="#{bean.logout}" />
</p:menubar>
```

## Ajax and Non-Ajax Actions

As menu uses menuitems, it is easy to invoke actions with or without ajax as well as navigation. See menuitem documentation for more information about the capabilities.

```
<p:menubar>
    <p:submenu label="Options">
        <p:menuitem value="Save" actionListener="#{bean.save}" update="comp"/>
        <p:menuitem value="Update" actionListener="#{bean.update}" ajax="false"/>
        <p:menuitem value="Navigate" url="http://www.primefaces.org"/>
    </p:submenu>
</p:menubar>
```

## Dynamic Menus

Menus can be created programmatically as well, see the dynamic menus part in menu component section for more information and an example.

## Skinning

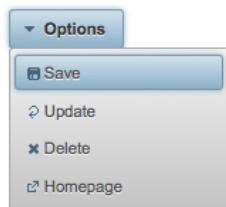
Menubar resides in a main container which *style* and *styleClass* attributes apply. Following is the list of structural style classes;

Style Class	Applies
.wijmo-wijmenu	Container element of menu
.wijmo-wijmenu-horizontal	Container element of menu
.wijmo-wijmenu-list	List container
.wijmo-wijmenu-item	Each menu item
.wijmo-wijmenu-link	Anchor element in a link item
.wijmo-wijmenu-text	Text element in an item

As skinning style classes are global, see the main Skinning section for more information.

## 3.58 MenuButton

MenuButton displays different commands in a popup menu.



### Info

Tag	<b>menuButton</b>
Component Class	<b>org.primefaces.component.menubutton.MenuButton</b>
Component Type	<b>org.primefaces.component.MenuButton</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.MenuButtonRenderer</b>
Renderer Class	<b>org.primefaces.component.menubutton.MenuButtonRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
value	null	String	Label of the button
style	null	String	Style of the main container element
styleClass	null	String	Style class of the main container element
widgetVar	null	String	Name of the client side widget
model	null	MenuModel	MenuModel instance to create menus programmatically
disabled	FALSE	Boolean	Disables or enables the button.
zindex	1	Integer	z-index property to control overlapping with other elements.

Name	Default	Type	Description
effect	fade	String	Sets the effect for the menu display.
effectDuration	400	Integer	Sets the effect duration in milliseconds.

## Getting started with the MenuButton

MenuButton consists of one or more menuitems. Following menubutton example has three menuitems, first one is used triggers an action with ajax, second one does the similar but without ajax and third one is used for redirect purposes.

```
<p:menuButton value="Options">
    <p:menuItem value="Save" actionListener="#{bean.save}" update="comp" />
    <p:menuItem value="Update" actionListener="#{bean.update}" ajax="false" />
    <p:menuItem value="Go Home" url="/home.jsf" />
</p:menuButton>
```

## Effects

Menu has a built-in animation to use when displaying&hiding itself and it's submenus. This animation is customizable using attributes like *effect* and *effectDuration*. Available animations are *fade* or *slide*, effectDuration is defined in milliseconds defaulting to 400.

## Dynamic Menus

Menus can be created programmatically as well, see the dynamic menus part in menu component section for more information and an example.

## Skinning

MenuButton resides in a main container which *style* and *styleClass* attributes apply. As skinning style classes are global, see the main Skinning section for more information. Following is the list of structural style classes;

Style Class	Applies
.ui-button	Button element
.ui-button-text	Label of button
.wijmo-wijmenu	Container element of menu
.wijmo-wijmenu-list	List container
.wijmo-wijmenu-item	Each menu item
.wijmo-wijmenu-link	Anchor element in a link item
.wijmo-wijmenu-text	Text element in an item

## 3.59 MenuItem

MenuItem is used by various menu components of PrimeFaces.

### Info

Tag	<b>menuItem</b>
Tag Class	<b>org.primefaces.component.menuitem.MenuItemTag</b>
Component Class	<b>org.primefaces.component.menuitem.MenuItem</b>
Component Type	<b>org.primefaces.component.MenuItem</b>
Component Family	<b>org.primefaces.component</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
value	null	String	Label of the menuitem
actionListener	null	javax.el.MethodExpression	Action listener to be invoked when menuitem is clicked.
action	null	javax.el.MethodExpression	Action to be invoked when menuitem is clicked.
immediate	FALSE	Boolean	When true, action of this menuitem is processed after apply request phase.
label	null	String	Label of the menuitem (Deprecated, use label instead)
url	null	String	Url to be navigated when menuitem is clicked
target	null	String	Target type of url navigation
helpText	null	String	Text to display additional information
style	null	String	Style of the menuitem label
styleClass	null	String	StyleClass of the menuitem label
onclick	null	String	Javascript event handler for click event
async	FALSE	Boolean	When set to true, ajax requests are not queued.

Name	Default	Type	Description
process	null	String	Component id(s) to process partially instead of whole view.
update	null	String	Client side id of the component(s) to be updated after async partial submit request.
onstart	null	String	Javascript handler to execute before ajax request begins.
oncomplete	null	String	Javascript handler to execute when ajax request is completed.
onsuccess	null	String	Javascript handler to execute when ajax request succeeds.
onerror	null	String	Javascript handler to execute when ajax request fails.
global	TRUE	Boolean	Global ajax requests are listened by ajaxStatus component, setting global to false will not trigger ajaxStatus.
ajax	TRUE	Boolean	Specifies submit mode.
icon	null	String	Path of the menuitem image.

## Getting started with MenuItem

MenuItem is a generic component used by the following PrimeFaces components.

- Menu
- MenuBar
- Breadcrumb
- Dock
- Stack
- MenuButton

Note that some attributes of menuItem might not be supported by these menu components. Refer to the specific component documentation for more information.

## Navigation vs Action

MenuItem has two use cases, directly navigating to a url with GET and doing a POST do execute an action which you can still do navigation with JSF navigation rules. This is decided by url attribute, if it is present menuItem does a GET request, if not parent form is posted.

## Icons

There are two ways to specify an icon of a menuItem, you can either use bundled icons within PrimeFaces or provide your own via css.

### ThemeRoller Icons

```
<p:menuItem icon="ui-icon ui-icon-disk" ... />
```

### Custom Icons

```
<p:menuItem icon="barca" ... />
```

```
.barca {  
    background: url(barca_logo.png) no-repeat;  
    height:16px;  
    width:16px;  
}
```

## 3.60 Message

Message is a pre-skinned extended version of the standard JSF message component.



### Info

Tag	<b>message</b>
Component Class	<b>org.primefaces.component.message.Message</b>
Component Type	<b>org.primefaces.component.Message</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.MessageRenderer</b>
Renderer Class	<b>org.primefaces.component.message.MessageRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
showSummary	FALSE	Boolean	Specifies if the summary of the FacesMessage should be displayed.
showDetail	TRUE	Boolean	Specifies if the detail of the FacesMessage should be displayed.
for	null	String	Id of the component whose messages to display.
redisplay	TRUE	Boolean	Defines if already rendered messages should be displayed
display	both	String	Defines the display mode.

### Getting started with Message

Message usage is exactly same as standard message.

```
<h:inputText id="txt" value="#{bean.text}" />
<p:message for="txt" />
```

## Display Mode

Message component has three different display modes;

- text : Only message text is displayed.
- icon : Only message severity is displayed and message text is visible as a tooltip.
- both (default) : Both icon and text are displayed.

## Skinning Message

Full list of CSS selectors of message is as follows;

Style Class	Applies
ui-message-{severity}	Container element of the message
ui-message-{severity}-summary	Summary text
ui-message-{severity}-info	Detail text

{severity} can be ‘info’, ‘error’, ‘warn’ and error.

## 3.61 Messages

Messages is a pre-skinned extended version of the standard JSF messages component.



**Sample info message** PrimeFaces rocks!



**Sample warn message** Watch out for PrimeFaces!



**Sample error message** PrimeFaces makes no mistakes



**Sample fatal message** Fatal Error in System

### Info

Tag	<b>messages</b>
Component Class	<b>org.primefaces.component.messages.Messages</b>
Component Type	<b>org.primefaces.component.Messages</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.MessagesRenderer</b>
Renderer Class	<b>org.primefaces.component.messages.MessagesRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
showSummary	FALSE	Boolean	Specifies if the summary of the FacesMessages should be displayed.
showDetail	TRUE	Boolean	Specifies if the detail of the FacesMessages should be displayed.
globalOnly	FALSE	String	When true, only facesmessages with no clientIds are displayed.

Name	Default	Type	Description
redisplay	TRUE	Boolean	Defines if already rendered messages should be displayed

## Getting started with Messages

Message usage is exactly same as standard messages.

```
<p:messages />
```

## Skinning Message

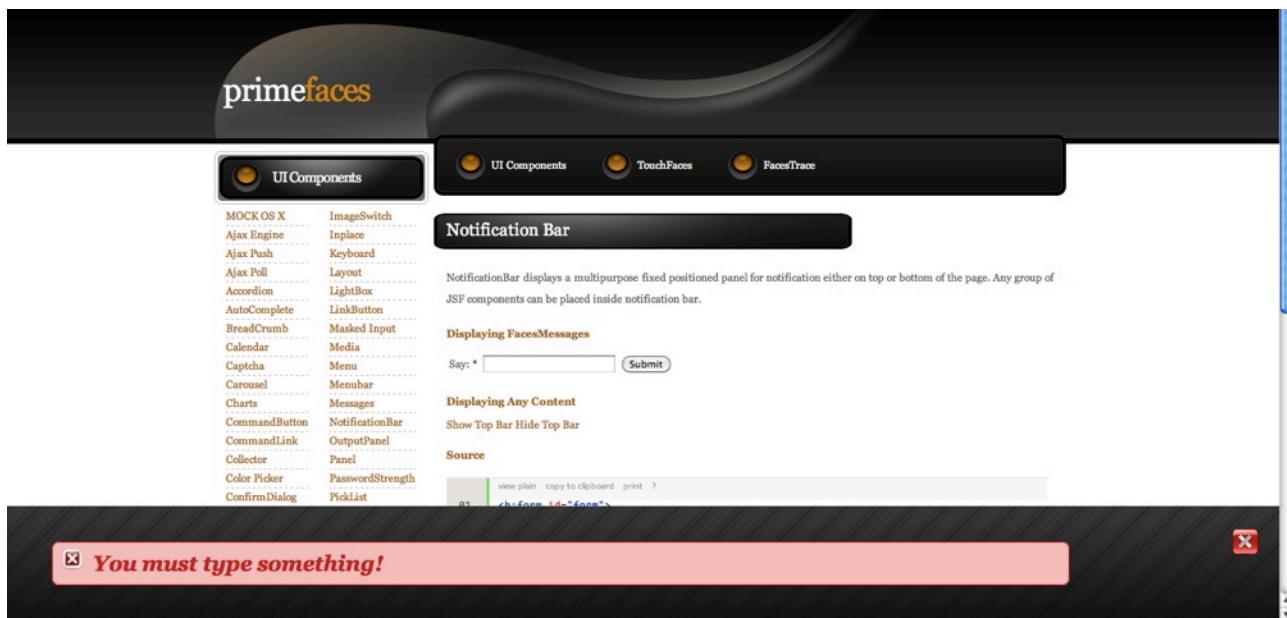
Full list of CSS selectors of message is as follows;

Style Class	Applies
ui-messages-{severity}	Container element of the message
ui-messages-{severity}-summary	Summary text
ui-messages-{severity}-detail	Detail text
ui-messages-{severity}-icon	Icon of the message.

{severity} can be ‘info’, ‘error’, ‘warn’ and error.

## 3.62 NotificationBar

NotificationBar displays a multipurpose fixed positioned panel for notification. Any group of JSF content can be placed inside notificationbar.



### Info

<b>Tag</b>	<b>notificationBar</b>
Component Class	<b>org.primefaces.component.notificationbar.NotificationBar</b>
Component Type	<b>org.primefaces.component.NotificationBar</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.NotificationBarRenderer</b>
Renderer Class	<b>org.primefaces.component.notificationbar.NotificationBarRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
style	null	String	Style of the container element

Name	Default	Type	Description
styleClass	null	String	StyleClass of the container element
position	top	String	Position of the bar, "top" or "bottom".
effect	fade	String	Name of the effect, "fade", "slide" or "none".
effectSpeed	normal	String	Speed of the effect, "slow", "normal" or "fast".

## Getting started with NotificationBar

As notificationBar is a panel component, any JSF and non-JSF content can be placed inside.

```
<p:notificationBar widgetVar="topBar">
    //Content
</p:notificationBar>
```

## Showing and Hiding

To show and hide the content, notificationBar provides an easy to use client side api that can be accessed through the widgetVar. *show()* displays the bar and *hide()* hides it.

```
<p:notificationBar widgetVar="topBar">
    //Content
</p:notificationBar>

<h:outputLink value="#" onclick="topBar.show()">Show</h:outputLink>
<h:outputLink value="#" onclick="topBar.hide()">Hide</h:outputLink>
```

Note that notificationBar has a default built-in close icon to hide the content.

## Effects

Default effect to be used when displaying and hiding the bar is "fade", another possible effect is "slide".

```
<p:notificationBar widgetVar="topBar" effect="slide">
    //Content
</p:notificationBar>
```

If you'd like to turn off animation, set effect name to "none". In addition duration of the animation is controlled via effectSpeed attribute that can take "normal", "slow" or "fast" as its value.

## Position

Default position of bar is "top", other possibility is placing the bar at the bottom of the page. Note that bar positioning is fixed so even page is scrolled, bar will not scroll.

```
<p:notificationBar widgetVar="topBar" position="bottom">
    //Content
</p:notificationBar>
```

## Skinning

style and styleClass attributes apply to the main container element. Additionally there are two pre-defined css selectors used to customize the look and feel.

Selector	Applies
.ui-notificationbar	Main container element
.ui-notificationbar-close	Close icon element

## 3.63 OutputPanel

OutputPanel is a display only element that's useful in various cases such as adding placeholders to a page.

### Info

Tag	<b>outputPanel</b>
Component Class	<b>org.primefaces.component.outputpanel.OutputPanel</b>
Component Type	<b>org.primefaces.component.OutputPanel</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.OutputPanelRenderer</b>
Renderer Class	<b>org.primefaces.component.output.OutputPanelRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
style	null	String	Style of the html container element
styleClass	null	String	StyleClass of the html container element
layout	inline	String	Layout of the panel, valid values are <i>inline(span)</i> or <i>block(div)</i> .

### AjaxRendered

Due to the nature of ajax, it is much simpler to update an existing element on page rather than inserting a new element to the dom. When a JSF component is not rendered, no markup is rendered so for components with conditional rendering regular PPR mechanism may not work since the markup to update on page does not exist. OutputPanel is useful in this case.

Suppose the rendered condition on bean is false when page is loaded initially and search method on bean sets the condition to be true meaning datatable will be rendered after a page submit. The problem is although partial output is generated, the markup on page cannot be updated since it doesn't exist.

```
<p:dataTable id="tbl" rendered="#{bean.condition}" ...>
    //columns
</p:dataTable>

<p:commandButton update="tbl" actionListener="#{bean.search}" />
```

Solution is to use the outputPanel as a placeHolder.

```
<p:outputPanel id="out">
    <p:dataTable id="tbl" rendered="#{bean.condition}" ...>
        //columns
    </p:dataTable>
</p:outputPanel>

<p:commandButton update="out" actionListener="#{bean.list}" />
```

Note that you won't need an outputPanel if commandButton has no update attribute specified, in this case parent form will be updated partially implicitly making an outputPanel use obsolete.

## Layout

OutputPanel has two layout modes;

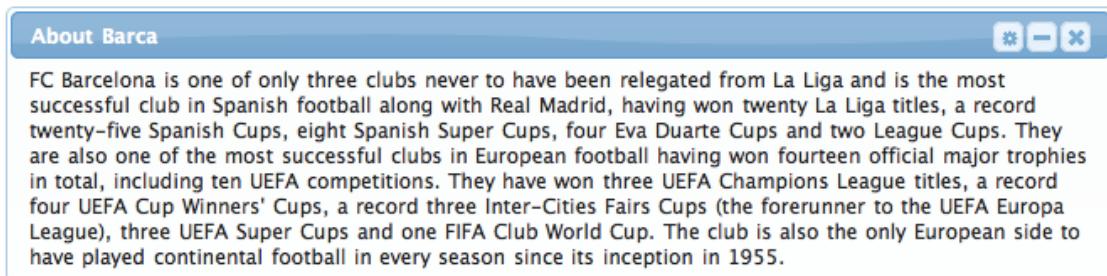
- inline (default): Renders a span
- block: Renders a div

## Skinning OutputPanel

*style* and *styleClass* attributes are used to skin the outputPanel.

## 3.64 Panel

Panel is a grouping component for other components, notable features are toggling, closing, built-in popup menu and ajax event listeners.



### Info

Tag	<b>panel</b>
Component Class	<b>org.primefaces.component.panel.Panel</b>
Component Type	<b>org.primefaces.component.Panel</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.PanelRenderer</b>
Renderer Class	<b>org.primefaces.component.panel.PanelRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
header	null	String	Header text
footer	null	String	Footer text
toggable	FALSE	Boolean	Makes panel toggable.
toggleSpeed	1000	Integer	Speed of toggling in milliseconds
onToggleUpdate	null	String	Component(s) to update after ajax toggleListener is invoked and panel is toggled

Name	Default	Type	Description
toggleListener	null	MethodExpr	Server side listener to invoke with ajax when a panel is toggled.
collapsed	FALSE	Boolean	Renders a toggleable panel as collapsed.
style	null	String	Style of the panel
styleClass	null	String	Style class of the panel
closable	FALSE	Boolean	Make panel closable.
closeListener	null	MethodExpr	Server side listener to invoke with ajax when a panel is closed.
onCloseUpdate	null	String	Component(s) to update after ajax closeListener is invoked and panel is closed.
onCloseStart	null	String	Javascript event handler to invoke before closing starts.
onCloseComplete	null	String	Javascript event handler to invoke after closing completes.
closeSpeed	1000	Integer	Speed of closing effect in milliseconds
visible	TRUE	Boolean	Renders panel as hidden.
widgetVar	null	String	Name of the client side widget

## Getting started with Panel

In it's simplest form, panel encapsulates other components.

```
<p:panel>
    //Child components here...
</p:panel>
```

## Header and Footer

Header and Footer texts can be provided by *header* and *footer* attributes.

```
<p:panel header="Header Text" footer="Footer Text">
    //Child components here...
</p:panel>
```

Instead of text, you can place custom content with facets as well.

```
<p:panel>
    <f:facet name="header">
        <h:outputText value="Header Text" />
    </f:facet>

    <f:facet name="footer">
        <h:outputText value="Footer Text" />
    </f:facet>

    //Child components here...
</p:panel>
```

When both header attribute and header facet is defined, facet is chosen, same applies to footer.

## Toggling

Panel contents can be toggled with a slide effect using the toggleable feature. Toggling is turned off by default and toggleable needs to be set to true to enable it. By default toggling takes 1000 milliseconds, this can be tuned by the *toggleSpeed* attribute.

```
<p:panel header="Header Text" toggleable="true">
    //Child components here...
</p:panel>
```

If you'd like to get notified on server side when a panel is toggled, you can do so by using ajax toggleListener. Optionally onToggleUpdate is used to update other components with ajax after toggling is completed. Following example adds a FacesMessage and displays it when panel is toggled.

```
<p:panel toggleListener="#{panelBean.handleToggle}" onToggleUpdate="msg">
    //Child components here...
</p:panel>

<p:messages id="msg" />
```

```
public void handleToggle(ToggleEvent event) {
    Visibility visibility = event.getVisibility();

    //Add facesmessage
}
```

*org.primefaces.event.ToggleEvent* provides visibility information using *org.primefaces.model.Visibility* enum that has the values HIDDEN or VISIBLE.

## Closing

Similar to toggling, a panel can also be closed as well. This is enabled by setting closable to true.

```
<p:panel closable="true">
    //Child components here...
</p:panel>
```

If you'd like to bind client side event handlers to the close event, provide the names of javascript functions using onCloseStart and onCloseComplete attributes. On the other hand, for server side use ajax closeListener and optional onCloseUpdate options.

```
<p:panel closeListener="#{panelBean.handleClose}" onCloseUpdate="msg">
    //Child components here...
</p:panel>

<p:messages id="msg" />
```

```
public void handleClose(CloseEvent event) {
    //Add facesmessage
}
```

CloseEvent is an *org.primefaces.event.CloseEvent* instance.

## Popup Menu

Panel has built-in support to display a fully customizable popup menu, an icon to display the menu is placed at top-right corner. This feature is enabled by defining a menu component and defining it as the options facet.

```
<p:panel closable="true">
    //Child components here...

    <f:facet name="options">
        <p:menu>
            //Menuitems
        </p:menu>
    </f:facet>
</p:panel>
```

## Skinning Panel

Panel resides in a main container which *style* and *styleClass* attributes apply.

Following is the list of structural style classes;

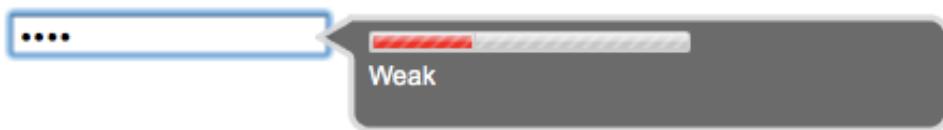
Style Class	Applies
.ui-panel	Main container element of panel
.ui-panel-titlebar	Header container
.ui-panel-title	Header text
.ui-panel-titlebar-icon	Option icon in header
.ui-panel-content	Panel content
.ui-panel-footer	Panel footer

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.65 Password

Password component is an extended version of standard inputSecret component with theme integration and strength indicator.



### Info

Tag	<b>password</b>
Component Class	<b>org.primefaces.component.password.Password</b>
Component Type	<b>org.primefaces.component.Password</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.PasswordRenderer</b>
Renderer Class	<b>org.primefaces.component.password.PasswordRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Value of the component than can be either an EL expression or a literal text
converter	null	Converter/ String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id
immediate	FALSE	Boolean	When set true, process validations logic is executed at apply request values phase for this component.
required	FALSE	boolean	Marks component as required
validator	null	MethodBinding	A method binding expression that refers to a method validationg the input

Name	Default	Type	Description
valueChangeListener	null	MethodExpr	A method binding expression that refers to a method for handling a valuechangeevent
requiredMessage	null	String	Message to be displayed when required field validation fails.
converterMessage	null	String	Message to be displayed when conversion fails.
validatorMessage	null	String	Message to be displayed when validation fields.
feedback	FALSE	Boolean	Enables strength indicator.
minLength	8	Integer	Minimum length of a strong password
inline	FALSE	boolean	Displays feedback inline rather than using a popup.
promptLabel	Please enter a password	String	Label of prompt.
level	1	Integer	Level of security.
weakLabel	Weak	String	Label of weak password.
goodLabel	Good	String	Label of good password.
strongLabel	String	String	Label of strong password.
onshow	null	String	Javascript event handler to be executed when password strength indicator is shown.
onhide	null	String	Javascript event handler to be executed when password strength indicator is hidden.
widgetVar	null	String	Name of the client side widget.
accesskey	null	String	Access key that when pressed transfers focus to the input element.
alt	null	String	Alternate textual description of the input field.
autocomplete	null	String	Controls browser autocomplete behavior.
dir	null	String	Direction indication for text that does not inherit directionality. Valid values are LTR and RTL.
disabled	FALSE	Boolean	Disables input field
label	null	String	A localized user presentable name.
lang	null	String	Code describing the language used in the generated markup for this component.
maxlength	null	Integer	Maximum number of characters that may be entered in this field.
onblur	null	String	Client side callback to execute when input element loses focus.

Name	Default	Type	Description
onchange	null	String	Client side callback to execute when input element loses focus and its value has been modified since gaining focus.
onclick	null	String	Client side callback to execute when input element is clicked.
ondblclick	null	String	Client side callback to execute when input element is double clicked.
onfocus	null	String	Client side callback to execute when input element receives focus.
onkeydown	null	String	Client side callback to execute when a key is pressed down over input element.
onkeypress	null	String	Client side callback to execute when a key is pressed and released over input element.
onkeyup	null	String	Client side callback to execute when a key is released over input element.
onmousedown	null	String	Client side callback to execute when a pointer button is pressed down over input element
onmousemove	null	String	Client side callback to execute when a pointer button is moved within input element.
onmouseout	null	String	Client side callback to execute when a pointer button is moved away from input element.
onmouseover	null	String	Client side callback to execute when a pointer button is moved onto input element.
onmouseup	null	String	Client side callback to execute when a pointer button is released over input element.
onselect	null	String	Client side callback to execute when text within input element is selected by user.
readonly	FALSE	Boolean	Flag indicating that this component will prevent changes by the user.
size	null	Integer	Number of characters used to determine the width of the input element.
style	null	String	Inline style of the input element.
styleClass	null	String	Style class of the input element.
tabindex	null	Integer	Position of the input element in the tabbing order.
title	null	String	Advisory tooltip information.

## Getting Started with Password

Password is an input component and used just like a standard input text. Most important attribute is *feedback*, when enabled (default) a password strength indicator is displayed, disabling feedback option will make password component behave like standard inputSecret.

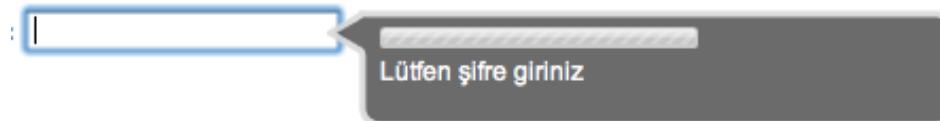
```
<p:password value="#{bean.password}" feedback="true|false" />
```

```
public class Bean {  
    private String password;  
  
    public String getPassword() { return password; }  
    public void setPassword(String password) { this.password = password; }  
}
```

## I18N

Although all labels are in English by default, you can provide custom labels as well. Following password gives feedback in Turkish.

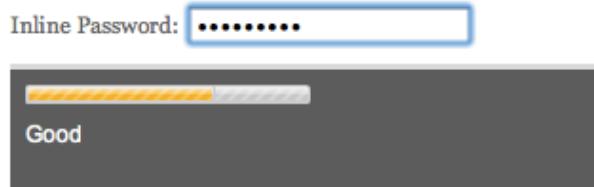
```
<p:password value="#{bean.password}" promptLabel="Lütfen şifre giriniz"  
weakLabel="Zayıf" goodLabel="Orta seviye" strongLabel="Güçlü" />
```



## Inline Strength Indicator

By default strength indicator is shown in an overlay, if you prefer an inline indicator just enable inline mode.

```
<p:password value="#{mybean.password}" inline="true"/>
```



## Custom Animations

Using onshow and onhide callbacks, you can create your own animation as well.

```
<p:password value="#{mybean.password}" inline="true"
    onshow="fadein" onhide="fadeout"/>
```

This examples uses jQuery api for fadeIn and fadeOut effects. Each callback takes two parameters; input and container. input is the actual input element of password and container is the strength indicator element.

```
<script type="text/javascript">
    function fadein(input, container) {
        container.fadeIn("slow");
    }

    function fadeout(input, container) {
        container.fadeOut("slow");
    }
</script>
```

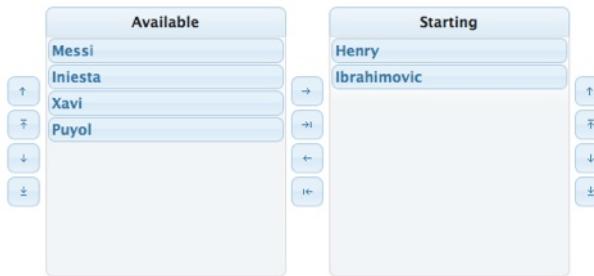
## Skinning Password

Skinning selectors for password is as follows;

Name	Applies
.jpassword	Container element of strength indicator.
.jpassword-meter	Visual bar of strength indicator.
.jpassword-info	Feedback text of strength indicator.

## 3.66 PickList

PickList is used for transferring data between two different collections.



### Info

Tag	<b>pickList</b>
Component Class	<b>org.primefaces.component.picklist.Panel</b>
Component Type	<b>org.primefaces.component.PickList</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.PickListRenderer</b>
Renderer Class	<b>org.primefaces.component.picklist.PickListRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Value of the component than can be either an EL expression or a literal text
converter	null	Converter/String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id
immediate	FALSE	Boolean	When set true, process validations logic is executed at apply request values phase for this component.
required	FALSE	Boolean	Marks component as required

Name	Default	Type	Description
validator	null	Method Expr	A method binding expression that refers to a method validating the input
valueChangeListener	null	Method Expr	A method binding expression that refers to a method for handling a valuechangeevent
requiredMessage	null	String	Message to be displayed when required field validation fails.
converterMessage	null	String	Message to be displayed when conversion fails.
validatorMessage	null	String	Message to be displayed when validation fields.
var	null	String	Name of the iterator.
itemLabel	null	String	Label of an item.
itemValue	null	Object	Value of an item.
style	null	String	Style of the main container.
styleClass	null	String	Style class of the main container.
widgetVar	null	String	Name of the client side widget.
disabled	FALSE	Boolean	Disables the component.
effect	null	String	Name of the animation to display.
effectSpeed	null	String	Speed of the animation.
iconOnly	FALSE	Boolean	When enabled picklist button controls only render icons and texts are displayed as tooltips.
addLabel	Add	String	Text of add button.
addAllLabel	Add All	String	Text of add all button.
removeLabel	Remove	String	Text of remove button.
removeAllLabel	Remove All	String	Text of remove all button.
moveUpLabel	Move Up	String	Text of move up button.
moveTopLabel	Move Top	String	Text of move top button.
moveDownLabel	Move Down	String	Text of move down button.
moveBottomLabel	Move Bottom	String	Text of move bottom button.
showSourceControls	FALSE	String	Specifies visibility of reorder buttons of source list.
showTargetControls	FALSE	String	Specifies visibility of reorder buttons of target list.
onTransfer	null	String	Client side callback to execute when an item is transferred from one list to another.

## Getting started with PickList

You need to create custom model called `org.primefaces.model.picklist.DualListModel` to use PickList. As the name suggests it consists of two lists, one is the source list and the other is the target. As the first example we'll create a DualListModel that contains basic Strings.

```
public class PickListBean {

    private DualListModel<String> cities;

    public PickListBean() {
        List<String> source = new ArrayList<String>();
        List<String> target = new ArrayList<String>();

        citiesSource.add("Istanbul");
        citiesSource.add("Ankara");
        citiesSource.add("Izmir");
        citiesSource.add("Antalya");
        citiesSource.add("Bursa");

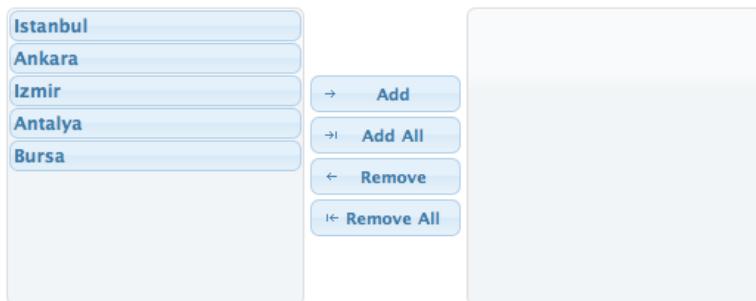
        cities = new DualListModel<String>(citiesSource, citiesTarget);
    }

    public DualListModel<String> getCities() {
        return cities;
    }

    public void setCities(DualListModel<String> cities) {
        this.cities = cities;
    }
}
```

And bind the cities dual list to the picklist;

```
<p:pickList value="#{pickListBean.cities}" var="city"
            itemLabel="#{city}" itemValue="#{city}">
```



When the enclosed form is submitted, the dual list reference is populated with the new values and you can access these values with `DualListModel.getSource()` and `DualListModel.getTarget()` api.

## POJOs

Most of the time you would deal with complex pojos rather than simple types like String. This use case is no different except the addition of a converter. Following pickList displays a list of players(name, age ...).

```
public class PickListBean {

    private DualListModel<Player> players;

    public PickListBean() {
        //Players
        List<Player> source = new ArrayList<Player>();
        List<Player> target = new ArrayList<Player>();

        source.add(new Player("Messi", 10));
        source.add(new Player("Ibrahimovic", 9));
        source.add(new Player("Henry", 14));
        source.add(new Player("Iniesta", 8));
        source.add(new Player("Xavi", 6));
        source.add(new Player("Puyol", 5));

        players = new DualListModel<Player>(source, target);
    }

    public DualListModel<Player> getPlayers() {
        return players;
    }
    public void setPlayers(DualListModel<Player> players) {
        this.players = players;
    }
}
```

```
<p:pickList value="#{pickListBean.players}" var="player"
    itemLabel="#{player.name}" itemValue="#{player}" converter="player">
```

PlayerConverter in this case should implement *javax.faces.convert.Converter* contract and implement *getAsString*, *getAsObject* methods.

Note that a converter is always necessary for primitive types like long, integer, boolean as well.

## Reordering

PickList support reordering of source and target lists, these are enabled by *showSourceControls* and *showTargetControls* options.

## Icon Only

Both transfer and reordering controls are buttons with an icon and text, to save some space enable *iconOnly* option that displays only icons on buttons and labels as tooltips.

## Effects

An animation is displayed when transferring when item to another or reordering a list, default effect is fade and following options are available to be applied using *effect* attribute;

- blind
- bounce
- clip
- drop
- explode
- fold
- highlight
- puff
- pulsate
- scale
- shake
- size
- slide

*effectSpeed* attribute is used to customize the animation speed, valid values are *slow*, *normal* and *fast*.

## onTransfer

If you'd like to execute custom javascript when an item is transferred bind your javascript function to *onTransfer* attribute.

```
<p:pickList value="#{pickListBean.cities}" var="city"
    itemLabel="#{city}" itemValue="#{city}" onTransfer="handleTransfer(e)">
```

```
<script type="text/javascript">
    function handleTransfer(e) {
        //item = e.item
        //fromList = e.from
        //toList = e.toList
    }
</script>
```

## Captions

Caption texts for lists are defined with facets named *sourceCaption* and *targetCaption*;

```
<p:pickList value="#{pickListBean.cities}" var="city"
    itemLabel="#{city}" itemValue="#{city}" onTransfer="handleTransfer(e)">
    <f:facet name="sourceCaption">Available</facet>
    <f:facet name="targetCaption">Selected</facet>
</p:pickList>
```

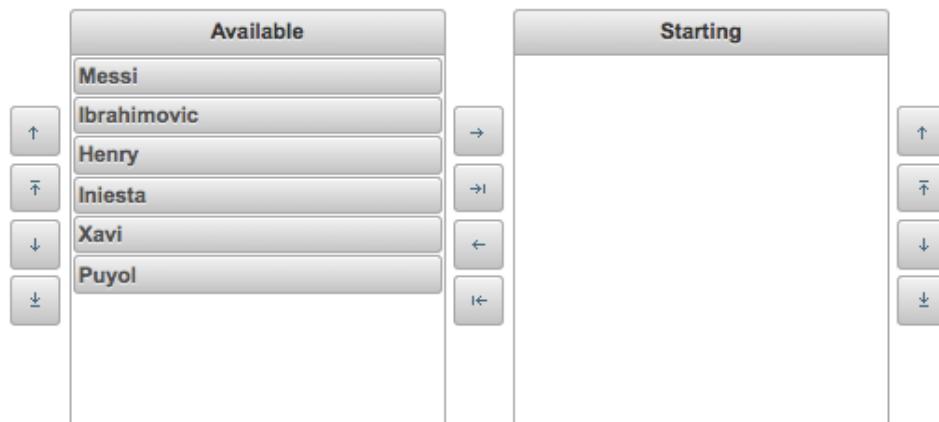
## Skinning

Panel resides in a main container which *style* and *styleClass* attributes apply.

Following is the list of structural style classes;

Style Class	Applies
.ui-picklist	Main container element(table) of picklist
.ui-picklist-list	Lists of a picklist
.ui-picklist-list-source	Source list
.ui-picklist-list-target	Target list
.ui-picklist-source-controls	Container element of source list reordering controls
.ui-picklist-target-controls	Container element of target list reordering controls
.ui-picklist-button	Buttons of a picklist
.ui-picklist-button-move-up	Move up button
.ui-picklist-button-move-top	Move top button
.ui-picklist-button-move-down	Move down button
.ui-picklist-button-move-bottom	Move bottom button
.ui-picklist-button-add	Add button
.ui-picklist-button-add-all	Add all button
.ui-picklist-button-remove-all	Remove all button
.ui-picklist-button-add	Add button

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.67 Poll

Poll is an ajax component that has the ability to send periodical ajax requests and execute listeners on JSF backing beans.

### Info

Tag	<b>poll</b>
Component Class	<b>org.primefaces.component.poll.Poll</b>
Component Type	<b>org.primefaces.component.Poll</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.PollRenderer</b>
Renderer Class	<b>org.primefaces.component.poll.PollRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
interval	2	Integer	Interval in seconds to do periodic ajax requests.
listener	null	MethodExpr	A method expression to invoke by polling.
action	null	MethodExpr	(Deprecated, use listener instead) A method expression to invoke by polling.
actionListener	null	MethodExpr	(Deprecated, use listener instead) A method expression to invoke by polling.
immediate	FALSE	Boolean	Boolean value that determines the phaseId, when true actions are processed at apply_request_values, when false at invoke_application phase.
widgetVar	null	String	Name of the client side widget.
async	FALSE	Boolean	When set to true, ajax requests are not queued.
process	null	String	Component id(s) to process partially instead of whole view.
update	null	String	Component(s) to be updated with ajax.

Name	Default	Type	Description
onstart	null	String	Javascript handler to execute before ajax request is begins.
oncomplete	null	String	Javascript handler to execute when ajax request is completed.
onsuccess	null	String	Javascript handler to execute when ajax request succeeds.
onerror	null	String	Javascript handler to execute when ajax request fails.
global	TRUE	Boolean	Global ajax requests are listened by ajaxStatus component, setting global to false will not trigger ajaxStatus.
autoStart	TRUE	Boolean	In autoStart mode, polling starts automatically on page load, to start polling on demand set to false.

## Getting started with Poll

Poll below invokes increment method on CounterBean every 2 seconds and txt\_count is updated with the new value of the count variable. Note that poll must be nested inside a form.

```
<h:outputText id="txt_count" value="#{counterBean.count}" />
<p:poll listener="#{counterBean.increment}" update="txt_count" />
```

```
public class CounterBean {
    private int count;
    public void increment() {
        count++;
    }
    public int getCount() {
        return this.count;
    }
    public void setCount(int count) {
        this.count = count;
    }
}
```

## Tuning timing

By default the periodic interval is 2 seconds, this is changed with the interval attribute. Following poll works every 5 seconds.

```
<h:outputText id="txt_count" value="#{counterBean.count}" />  
<p:poll listener="#{counterBean.increment}" update="txt_count" interval="5" />
```

## Start and Stop

Poll can be started manually, handy widgetVar attribute is once again comes for help.

```
<h:form>  
    <h:outputText id="txt_count" value="#{counterBean.count}" />  
    <p:poll interval="5" actionListener="#{counterBean.increment}"  
           update="txt_count" widgetVar="myPoll" autoStart="false" />  
    <a href="#" onclick="myPoll.start();">Start</a>  
    <a href="#" onclick="myPoll.stop();">Stop</a>  
</h:form>
```

## 3.68 Printer

Printer allows sending a specific JSF component to the printer, not the whole page.

### Info

Tag	<b>printer</b>
Component Class	<b>org.primefaces.component.printer.Printer</b>
Component Type	<b>org.primefaces.component.Printer</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.PrinterRenderer</b>
Renderer Class	<b>org.primefaces.component.printer.PrinterRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
target	null	String	Id of the component to print.

### Getting started with the Printer

Printer is attached to any command component like a button or a link. Examples below demonstrates how to print a simple output text or a particular image on page;

```
<h:commandButton id="btn" value="Print">
    <p:printer target="output" />
</h:commandButton>
<h:outputText id="output" value="PrimeFaces Rocks!" />

<h:outputLink id="lnk" value="#">
    <p:printer target="image" />
    <h:outputText value="Print Image" />
</h:outputLink>
<p:graphicImage id="image" value="/images/nature1.jpg" />
```

## 3.69 ProgressBar

ProgressBar is a process status indicator that can either work purely on client side or interact with server side using ajax.



### Info

Tag	<b>proppressBar</b>
Component Class	<b>org.primefaces.component.progressbar.ProgressBar</b>
Component Type	<b>org.primefaces.component.Progressbar</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.ProgressBarRenderer</b>
Renderer Class	<b>org.primefaces.component.progressbar.ProgressBarRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
widgetVar	null	String	Name of the client side widget
value	0	Integer	Value of the progress bar
disabled	FALSE	Boolean	Disables or enables the progressbar
ajax	FALSE	Boolean	Specifies the mode of progressBar, in ajax mode progress value is retrieved from a backing bean.
interval	3000	Integer	Interval in seconds to do periodic requests in ajax mode.
style	null	String	Inline style of the main container element.
styleClass	null	String	Style class of the main container element.
oncomplete	null	String	Client side callback to execute when progress ends.

Name	Default	Type	Description
onCompleteUpdate	null	String	Specifies component(s) to update with ajax when progress is completed
completeListener	null	MethodExpr	A server side listener to be invoked when a progress is completed.
onCancelUpdate	null	String	Specifies component(s) to update with ajax when progress is cancelled
cancelListener	null	MethodExpr	A server side listener to be invoked when a progress is cancelled.

## Getting started with the ProgressBar

ProgressBar has two modes, "client"(default) or "ajax". Following is a pure client side progressBar.

```
<p:progressBar widgetVar="pb" />

<p:commandButton value="Start" type="button" onclick="start()" />
<p:commandButton value="Cancel" type="button" onclick="cancel()" />

<script type="text/javascript">
    function start() {
        this.progressInterval = setInterval(function(){
            pb.setValue(pbClient.getValue() + 10);
        }, 2000);
    }

    function cancel() {
        clearInterval(this.progressInterval);
        pb.setValue(0);
    }
</script>
```

## Ajax Progress

Ajax mode is enabled by setting ajax attribute to true, in this case the value defined on a managed bean is retrieved periodically and used to update the progress.

```
<p:progressBar ajax="true" value="#{progressBean.progress}" />
```

```
public class ProgressBean {

    private int progress;

    //getter and setter
}
```

## Interval

ProgressBar is based on polling and 3000 milliseconds is the default interval for ajax progress bar meaning every 3 seconds progress value will be recalculated. In order to set a different value, use the interval attribute.

```
<p:progressBar interval="5000" />
```

## Ajax Event Listeners

If you'd like to execute custom logic on server side when progress is completed or cancelled, define a *completeListener* or *cancelListener* respectively that refers to a java method.

Optionally *oncompleteUpdate* and *onCancelUpdate* options can be defined to update a part of the page. Following example adds a faces message and updates the growl component to display it when progress is completed or cancelled.

```
public class ProgressBean {  
  
    private int progress;  
  
    public void handleComplete() {  
        //Add a faces message  
    }  
  
    public void handleCancel() {  
        //Add a faces message  
    }  
  
    public int getProgress() {  
    }  
  
    public void setProgress(int progress) {  
        this.progress = progress;  
    }  
}
```

```
<p:progressBar widgetVar="pb" value="#{progressBean.progress}"  
    completeListener="#{progressBean.handleComplete}"  
    onCompleteUpdate="messages"  
    cancellistener="#{progressBean.handleCancel}"  
    onCancelUpdate="messages"  
    ajax="true"/>  
  
<p:growl id="messages" />
```

## Client Side API

Widget: *PrimeFaces.widget.ProgressBar*

Method	Params	Return Type	Description
getValue()	-	Number	Returns current value
setValue(value)	value: Value to display	void	Sets current value
start()	-	void	Starts ajax progress bar
cancel()	-	void	Stops ajax progress bar

## Skinning

ProgressBar resides in a main container which *style* and *styleClass* attributes apply.

Following is the list of structural style classes;

Style Class	Applies
.ui-progressbar	Main container element of progressbar
.ui-progressbar-value	Value of the progressbar

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.70 Push

Push component is an agent that creates a channel between the server and the client.

### Info

Tag	<b>push</b>
Component Class	<b>org.primefaces.component.push.Push</b>
Component Type	<b>org.primefaces.component.Push</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.PushRenderer</b>
Renderer Class	<b>org.primefaces.component.push.PushRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
channel	null	Object	Unique channel name of the connection between subscriber and the server.
onpublish	null	Object	Javascript event handler that is process when the server publishes data.

### Getting started with the Push

See chapter 6, "Ajax Push/Comet" for detailed information.

## 3.71 Rating

Rating component features a star based rating system. Rating can be used as a plain input component or with ajax RateListeners.



### Info

Tag	<b>rating</b>
Component Class	<b>org.primefaces.component.rating.Rating</b>
Component Type	<b>org.primefaces.component.Rating</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.RatingRenderer</b>
Renderer Class	<b>org.primefaces.component.rating.RatingRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Value of the component than can be either an EL expression of a literal text
converter	null	Converter/ String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id
immediate	FALSE	Boolean	Boolean value that specifies the lifecycle phase the valueChangeEvents should be processed, when true the events will be fired at "apply request values", if immediate is set to false, valueChange Events are fired in "process validations" phase
required	FALSE	Boolean	Marks component as required

Name	Default	Type	Description
validator	null	MethodExpr	A method binding expression that refers to a method validating the input
valueChangeListener	null	MethodExpr	A method binding expression that refers to a method for handling a valuechangeevent
requiredMessage	null	String	Message to be displayed when required field validation fails.
converterMessage	null	String	Message to be displayed when conversion fails.
validatorMessage	null	String	Message to be displayed when validation fields.
widgetVar	null	String	Name of the client side widget.
stars	5	Integer	Number of stars to display
rateListener	null	MethodExpr	A server side listener to process a RateEvent
update	null	String	Component(s) to update with ajax.
disabled	FALSE	Boolean	Disabled user interaction
onRate	null	String	Client side callback to execute when rate happens.

## Getting Started with Rating

Rating is an input component that takes a double variable as it's value.

```
public class RatingBean {
    private double rating;
    private double getRating() { return this.rating; }
    private void setRating(double rating) { this.rating = rating; }
}
```

```
<p:rating value="#{ratingBean.rating}" />
```

When the enclosing form is submitted value of the rating will be assigned to the rating variable.

## Number of Stars

Default number of stars is 5, if you need less or more stars use the stars attribute. Following rating consists of 10 stars.

```
<p:rating value="#{ratingController.rating}" stars="10"/>
```



## Display Value Only

In cases where you only want to use the rating component to display the rating value and disallow user interaction, set *disabled* to true.

## Ajax RateListeners

In order to respond to rate events instantly rather than waiting for the user to submit the form, use the *rateListener* feature which sends an *org.primefaces.event.RateEvent* via an ajax request. On server side you can listen these RateEvent by defining RateListeners as MethodExpressions.

Rating below responds to a rate event instantly and updates the message component whose value is provided by the defined rateListener.

```
<p:rating rateListener="#{ratingController.handleRate}" update="msgs"/>
<p:messages id="msgs" />
```

```
public class RatingBean {
    public void handleRate(RateEvent rateEvent) {
        int rating = (int) rateEvent.getRating();
        //Add facesmessage
    }
}
```

## Client Side API

Widget: *PrimeFaces.widget.Rating*

Method	Params	Return Type	Description
getValue()	-	Number	Returns the current value
setValue(value)	value: Value to set	void	Updates rating value with provided one.
disable()	-	void	Disables component.
enable()	-	void	Enables component.

## Skinning

Following is the list of css classes for star rating;

Style Class	Applies
.star-rating-control	Main container element of progressbar
.rating-cancel	Value of the progressbar
.star-rating	Default star
.star-rating-on	Active star
.star-rating-hover	Hover star

## 3.72 RemoteCommand

RemoteCommand provides a way to execute JSF backing bean methods directly from javascript.

### Info

Tag	<b>remoteCommand</b>
Component Class	<b>org.primefaces.component.remotecommand.RemoteCommand</b>
Component Type	<b>org.primefaces.component.RemoteCommand</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.RemoteCommandRenderer</b>
Renderer Class	<b>org.primefaces.component.remotecommand.RemoteCommandRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
action	null	MethodExpr	A method expression that'd be processed in the partial request caused by uiajax.
actionListener	null	MethodExpr	An actionlistener that'd be processed in the partial request caused by uiajax.
immediate	FALSE	Boolean	Boolean value that determines the phaseId, when true actions are processed at apply_request_values, when false at invoke_application phase.
name	null	String	Name of the command
async	FALSE	Boolean	When set to true, ajax requests are not queued.
process	null	String	Component(s) to process partially instead of whole view.
update	null	String	Component(s) to update with ajax.
onstart	null	String	Javascript handler to execute before ajax request begins.
oncomplete	null	String	Javascript handler to execute when ajax request is completed.

Name	Default	Type	Description
onsuccess	null	String	Javascript handler to execute when ajax request succeeds.
onerror	null	String	Javascript handler to execute when ajax request fails.
global	TRUE	Boolean	Global ajax requests are listened by ajaxStatus component, setting global to false will not trigger ajaxStatus.

## Getting started with RemoteCommand

RemoteCommand is used by invoking the command from your javascript code.

```
<p:remoteCommand name="increment" actionListener="#{counter.increment}"
    out="count" />

<h:outputText id="count" value="#{counter.count}" />
```

```
<script type="text/javascript">
    function customfunction() {
        //your custom code

        increment();           //makes a remote call
    }
</script>
```

That's it whenever you execute your custom javascript function(eg customfunction()), a remote call will be made, actionListener is processed and output text is updated.

Note that remoteCommand must be nested inside a form.

## 3.73 Resizable

PrimeFaces features a resizable component that has the ability to make a JSF component resizable. Resizable can be used on various components like resize an input fields, panels, menus, images and more.

### Info

Tag	<b>resizable</b>
Component Class	<b>org.primefaces.component.resizable.Resizable</b>
Component Type	<b>org.primefaces.component.Resizable</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.ResizableRenderer</b>
Renderer Class	<b>org.primefaces.component.resizable.ResizableRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
widgetVar	null	String	Name of the client side widget.
for	null	String	Identifier of the target component to make resizable.
aspectRatio	FALSE	Boolean	Defines if aspectRatio should be kept or not.
proxy	FALSE	Boolean	Displays proxy element instead of actual element.
handles	null	String	Specifies the resize handles.
ghost	FALSE	Boolean	In ghost mode, resize helper is displayed as the original element with less opacity.
animate	FALSE	Boolean	Enables animation.
effect	swing	String	Effect to use in animation.
effectDuration	normal	String	Effect duration of animation.
maxWidth	null	Integer	Maximum width boundary in pixels.

Name	Default	Type	Description
maxHeight	null	Integer	Maximum height boundary in pixels.
minWidth	10	Integer	Minimum width boundary in pixels.
minHeight	10	Integer	Maximum height boundary in pixels.
containment	FALSE	Boolean	Sets resizable boundaries as the parents size.
grid	1	Integer	Snaps resizing to grid structure.
onResizeUpdate	null	String	Component(s) to update after ajax resizing.
resizeListener	null	MethodExpr	Server side method to execute after resize is completed.
onStart	null	String	Client side callback to execute when resizing begins.
onResize	null	String	Client side callback to execute during resizing.
onStop	null	String	Client side callback to execute after resizing end.

## Getting started with Resizable

Resizable is used by setting *for* option as the identifier of the target.

```
<p:graphicImage id="img" value="campnou.jpg" />
<p:resizable for="img" />
```

Another example is the input fields, if users need more space for a textarea, make it resizable by;

```
<h:inputTextarea id="area" value="Resize me if you need more space" />
<p:resizable for="area" />
```

## Boundaries

To prevent overlapping with other elements on page, boundaries need to be specified. There're 4 attributes for this *minWidth*, *maxWidth*, *minHeight* and *maxHeight*. The valid values for these attributes are numbers in terms of pixels.

```
<h:inputTextarea id="area" value="Resize me if you need more space" />
<p:resizable for="area" minWidth="20" minHeight="40" maxWidth="50" maxHeight="100"/>
```

## Handles

Resize handles to display are customize using *handles* attribute with a combination of *n*, *e*, *s*, *w*, *ne*, *se*, *sw* and *nw* as the value. Default value is "e, s, se".

```
<h:inputTextarea id="area" value="Resize me if you need more space" />
<p:resizable for="area" handles="e,w,n,se,sw,ne,nw"/>
```

## Visual Feedback

Resize helper is the element used to provide visual feedback during resizing. By default actual element itself is the helper and two options are available to customize the way feedback is provided. Enabling *ghost* option displays the element itself with a lower opacity, in addition enabling *proxy* option adds a css class called *.ui-resizable-proxy* which you can override to customize.

```
<h:inputTextarea id="area" value="Resize me if you need more space" />
<p:resizable for="area" proxy="true" />
```

```
.ui-resizable-proxy {
    border: 2px dotted #00F;
}
```

## Effects

Resizing can be animated using *animate* option and setting an *effect* name. Animation speed is customized using *effectDuration* option "slow", "normal" and "fast" as valid values.

```
<h:inputTextarea id="area" value="Resize me if you need more space" />
<p:resizable for="area" animate="true" effect="swing" effectDuration="normal" />
```

Following is the list of available effect names:

<ul style="list-style-type: none"> <li>• swing</li> <li>• easeInQuad</li> <li>• easeOutQuad</li> <li>• easeInOutQuad</li> <li>• easeInCubic</li> <li>• easeOutCubic</li> <li>• easeInOutCubic</li> </ul>	<ul style="list-style-type: none"> <li>• easeInQuart</li> <li>• easeOutQuart</li> <li>• easeInOutQuart</li> <li>• easeInQuint</li> <li>• easeOutQuint</li> <li>• easeInOutQuint</li> <li>• easeInSine</li> </ul>	<ul style="list-style-type: none"> <li>• easeOutSine</li> <li>• easeInExpo</li> <li>• easeOutExpo</li> <li>• easeInOutExpo</li> <li>• easeInCirc</li> <li>• easeOutCirc</li> <li>• easeInOutCirc</li> </ul>	<ul style="list-style-type: none"> <li>• easeInElastic</li> <li>• easeOutElastic</li> <li>• easeInOutElastic</li> <li>• easeInBack</li> <li>• easeOutBack</li> <li>• easeInOutBack</li> </ul>	<ul style="list-style-type: none"> <li>• easeInBounce</li> <li>• easeOutBounce</li> <li>• easeInOutBounce</li> </ul>
--	--	---	---	--

## Ajax Resize

If you'd like to get notified on the server side on resize events, define a *resizeListener* to process an *org.primefaces.event.ResizeEvent*. Optionally other component(s) on page can be updated after ajax resizing using *onResizeUpdate* option.

```
<h:inputTextarea id="area" value="Resize me if you need more space" />
<p:resizable for="area" resizeListener="#{resizeBean.handleResize}" />
```

```
public class ResizeBean {
    public void handleResize(ResizeEvent event) {
        int width = event.getWidth();
        int height = event.getHeight();
    }
}
```

## Client Side Callbacks

Resizable has three client side callbacks you can use to hook-in your javascript; *onStart*, *onResize* and *onStop*. All of these callbacks receive two parameters that provide various information about resize event.

```
<h:inputTextarea id="area" value="Resize me if you need more space" />
<p:resizable for="area" onStop="handleStop(event, ui)" />
```

```
function handleStop(event, ui) {
    //ui.helper = helper element as a jQuery object
    //ui.originalPosition = top, left position before resizing
    //ui.originalSize = width, height before resizing
    //ui.position = top, left after resizing
    //ui.size = width height of current size
}
```

## Skinning

Style Class	Applies
.ui-resizable	Element that is resizable
.ui-resizable-handle	Handle element
.ui-resizable-handle-{handlekey}	Particular handle element identified by handlekey like e, s, ne
.ui-resizable-proxy	Proxy helper element for visual feedback

## 3.74 Resource

Deprecated: Resource component has no use in PrimeFaces 2.2, use h:outputStylesheet and h:outputScript components instead.

Resource component manually adds resources like javascript and css bundled with PrimeFaces to a page.

### Info

Tag	<b>resource</b>
Component Class	<b>org.primefaces.component.resource.Resource</b>
Component Type	<b>org.primefaces.component.Resource</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.ResourceRenderer</b>
Renderer Class	<b>org.primefaces.component.resource.ResourceRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
name	null	String	Path of the resource

## 3.75 Resources

**Deprecated:** Resource component has no use in PrimeFaces 2.2, a page just needs to have standard h:head component instead.

Resources component renders all script and link tags necessary for PrimeFaces component to work.

### Info

Tag	<b>resources</b>
Tag Class	<b>org.primefaces.component.resources.ResourcesTag</b>
Component Class	<b>org.primefaces.component.resources.Resources</b>
Component Type	<b>org.primefaces.component.Resources</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.ResourcesRenderer</b>
Renderer Class	<b>org.primefaces.component.resources.ResourcesRenderer</b>

### Attributes

Name	Default	Type	Description
id	Assigned by JSF	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
exclude	null	String	Comma separated list of resources to be excluded.

## 3.76 Row

Row is a helper component for datatable.

### Info

Tag	<b>row</b>
Component Class	<b>org.primefaces.component.row.Row</b>
Component Type	<b>org.primefaces.component.Row</b>
Component Family	<b>org.primefaces.component</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean

### Getting Started with Row

See datatable grouping section for more information about how row is used.

## 3.77 RowEditor

RowToggler is a helper component for datatable.

### Info

Tag	<b>rowEditor</b>
Component Class	<b>org.primefaces.component.roweditor.RowEditor</b>
Component Type	<b>org.primefaces.component.RowEditor</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.RowEditorRenderer</b>
Renderer Class	<b>org.primefaces.component.roweditor.RowEditorRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean

### Getting Started with RowEditor

See inline editing section in datatable documentation for more information about usage.

## 3.78 RowExpansion

RowExpansion is a helper component of datatable used to implement expandable rows.

### Info

Tag	<b>rowExpansion</b>
Component Class	<b>org.primefaces.component.rowexpansion.RowExpansion</b>
Component Type	<b>org.primefaces.component.RowExpansion</b>
Component Family	<b>org.primefaces.component</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean

### Getting Started with RowExpansion

See datatable expandable rows section for more information about how rowExpansion is used.

## 3.79 RowToggler

RowToggler is a helper component for datatable.

### Info

Tag	<code>rowToggler</code>
Component Class	<code>org.primefaces.component.rowtoggler.RowToggler</code>
Component Type	<code>org.primefaces.component.RowToggler</code>
Component Family	<code>org.primefaces.component</code>
Renderer Type	<code>org.primefaces.component.RowTogglerRenderer</code>
Renderer Class	<code>org.primefaces.component.rowtoggler.RowTogglerRenderer</code>

### Attributes

Name	Default	Type	Description
<code>id</code>	<code>null</code>	String	Unique identifier of the component
<code>rendered</code>	<code>TRUE</code>	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
<code>binding</code>	<code>null</code>	Object	An el expression that maps to a server side UIComponent instance in a backing bean

### Getting Started with Row

See expandable rows section in datatable documentation for more information about usage.

## 3.80 Schedule

Schedule provides an Outlook Calendar, iCal like JSF component to manage events. Schedule is highly customizable featuring various views (month, day, week), built-in I18N, drag-drop, resize, customizable event dialog and skinning.



### Info

<b>Tag</b>	<b>schedule</b>
Component Class	<b>org.primefaces.component.schedule.Schedule</b>
Component Type	<b>org.primefaces.component.Schedule</b>
Component Family	<b>org.primefaces</b>
Renderer Type	<b>org.primefaces.component.ScheduleRenderer</b>
Renderer Class	<b>org.primefaces.component.schedule.ScheduleRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
widgetVar	null	String	Name of the client side widget.
value	null	Object	An org.primefaces.model.ScheduleModel instance representing the backed model
locale	null	Object	Locale for localization, can be String or a java.util.Locale instance

Name	Default	Type	Description
aspectRatio	null	Float	Ratio of calendar width to height, higher the value shorter the height is
view	month	String	The view type to use, possible values are 'month', 'agendaDay', 'agendaWeek', 'basicWeek', 'basicDay'
initialDate	null	Object	The initial date that is used when schedule loads. If omitted, the schedule starts on the current date
showWeekends	TRUE	Boolean	Specifies inclusion Saturday/Sunday columns in any of the views
style	null	String	Style of the main container element of schedule
styleClass	null	String	Style class of the main container element of schedule
editable	FALSE	Boolean	Defines whether calendar can be modified.
draggable	FALSE	Boolean	When true, events are draggable.
resizable	FALSE	Boolean	When true, events are resizable.
eventSelectListener	null	MethodExpr	A server side listener to invoke when an event is selected
dateSelectListener	null	MethodExpr	A server side listener to invoke when a date is selected
eventMoveListener	null	MethodExpr	A server side listener to invoke when a date is moved
eventResizeListener	null	MethodExpr	A server side listener to invoke when a date is resized
onEventSelectUpdate	null	String	Components to update with ajax when an event is selected, by default event dialog is updated
onDateSelectUpdate	null	String	Components to update with ajax when an empty date is selected, by default event dialog is updated
onEventMoveUpdate	null	String	Components to update with ajax when an event is moved.
onEventResizeUpdate	null	String	Components to update with ajax when an event is resized.
onEventSelectStart	null	String	Client side callback to execute when event select begins.
onEventSelectComplete	null	String	Client side callback to execute when event select completes.

Name	Default	Type	Description
onDateSelectStart	null	String	Client side callback to execute when date select begins.
onDateSelectComplete	null	String	Client side callback to execute when date select begins.
showHeader	TRUE	Boolean	Specifies visibility of header content.
leftHeaderTemplate	prev, next today	String	Content of left side of header.
centerHeaderTemplate	title	String	Content of center of header.
rightHeaderTemplate	month, agendaWeek, agendaDay	String	Content of right side of header.
allDaySlot	TRUE	Boolean	Determines if all-day slot will be displayed in agendaWeek or agendaDay views
slotMinutes	30	Integer	Interval in minutes in an hour to create a slot.
firstHour	6	Integer	First hour to display in day view.
minTime	null	String	Minimum time to display in a day view.
maxTime	null	String	Maximum time to display in a day view.
startWeekday	0	Integer	Specifies first day of week, by default it's 0 corresponding to sunday

## Getting started with Schedule

Schedule needs to be backed by a org.primefaces.model.ScheduleModel instance, a schedule model consists of org.primefaces.model.ScheduleEvent instances.

```
<p:schedule value="#{scheduleBean.model}" />
```

```
public class ScheduleBean {
    private ScheduleModel model;

    public ScheduleBean() {
        eventModel = new ScheduleModel<ScheduleEvent>();
        eventModel.addEvent(new DefaultScheduleEvent("title", new Date(),
            new Date()));
    }

    public ScheduleModel getModel() { return model; }
}
```

DefaultScheduleEvent is the default implementation of ScheduleEvent interface;

```
package org.primefaces.model;

import java.io.Serializable;
import java.util.Date;

public interface ScheduleEvent extends Serializable {

    public String getId();
    public void setId(String id);
    public Object getData();
    public String getTitle();
    public Date getStartDate();
    public Date getEndDate();
    public boolean isAllDay();
    public String getStyleClass();
}
```

Mandatory properties required to create a new event are the title, start date and end date. Other properties such as allDay get sensible default values. Table below describes each property in detail.

Property	Description
id	Used internally by PrimeFaces, you don't need to define it manually as id is auto-generated.
title	Title of the event.
startDate	Start date of type java.util.Date.
endDate	End date of type java.util.Date.
allDay	Flag indicating event is all day.
styleClass	Visual style class to enable multi resource display.
data	Optional data you can set to be represented by Event.

## Selecting an Event

EventSelectListener is invoked each time an event is clicked on an editable schedule.

```
<p:schedule value="#{scheduleBean.model}" editable="true"
            eventSelectListener="#{scheduleBean.onEventSelect}" />
```

```
public void onEventSelect(ScheduleEntrySelectEvent selectEvent) {
    ScheduleEvent event = selectEvent.getScheduleEvent();
}
```

*onEventSelect* listener above gets the selected event and sets it to ScheduleBean's event property to display. Optionally schedule has *onEventSelectUpdate* option to update any other component(s) on page, *onEventSelectStart* and *onEventSelectComplete* are client side callbacks to execute custom javascript.

## Selecting a Date

DateSelectListener is fired when an empty date is clicked which is useful to update the UI with selected date information. DateSelectListener in following example, resets the event and configures start/end dates to display in dialog.

```
<p:schedule value="#{scheduleBean.model}" editable="true"
    dateSelectListener="#{scheduleBean.onDateSelect}" />
```

```
public void onDateSelect(DateSelectEvent selectEvent) {
    Date selectedDate = selectEvent.getDate();
}
```

Optionally schedule has *onDateSelectEventUpdate* option to update any other component(s) on page, *onDateSelectStart* and *onDateSelectComplete* are client side callbacks to execute custom javascript.

## Moving an Event

Events can be dragged and dropped into new dates, to get notified about this with ajax, define a server side *eventMoveListener*.

```
<p:schedule value="#{scheduleBean.model}" editable="true"
    eventMoveListener="#{scheduleBean.onEventMove}" />
```

```
public void onEventMove(ScheduleEntryMoveEvent selectEvent) {
    ScheduleEvent event = selectEvent.getScheduleEvent();
    int dayDelta = selectEvent.getDayDelta();
    int minuteDelta = selectEvent.getMinuteDelta();
}
```

*org.primefaces.event.ScheduleEntryMoveEvent* passed to this listener provides useful information like the event that is moved and the difference in number of days/minutes. Note that by the time this listener invoked, schedule already updated moved event's start and end dates, the delta values are provided for information purposes so you can persist these information instantly.

Optionally schedule has *onEventMoveUpdate* option to update any other component(s) on page after an event is moved and defined eventMoveListener is invoked.

## Resizing an Event

Events can be resized, to get notified about this user interaction with ajax, define a server side eventResizeListener.

```
<p:schedule value="#{scheduleBean.model}" editable="true"
    eventResizeListener="#{scheduleBean.onEventResize}" />
```

```
public void onEventMove(ScheduleEntryResizeEvent selectEvent) {
    ScheduleEvent event = selectEvent.getScheduleEvent();
    int dayDelta = selectEvent.getDayDelta();
    int minuteDelta = selectEvent.getMinuteDelta();
}
```

*org.primefaces.event.ScheduleEntryResizeEvent* passed to this listener provides useful information like the event that is resized and the difference in number of days/minutes. Note that by the time this listener invoked, schedule already update moved event's end date, the delta values are provided for information purposes.

Optionally schedule has *onEventResizeUpdate* option to update any other component(s) on page after an event is resized and defined eventResizedListener is invoked.

## Ajax Updates

Schedule has a quite complex UI which is generated on-the-fly by the client side PrimeFaces.widget.Schedule widget to save bandwidth and increase page load performance. As a result when you try to update schedule like with a regular PrimeFaces PPR, you may notice a UI lag as the DOM will be regenerated and replaced.

Instead, Schedule provides a simple client side api and the *update* method. Whenever you call update, schedule will query it's server side ScheduleModel instance to check for updates, transport method used to load events dynamically is JSON, as a result this approach is much more effective than updating with regular PPR.

An example of this is demonstrated at editable schedule example, save button is calling *myschedule.update()* at oncomplete event handler.

## Editable Schedule

Let's put it altogether to come up a fully editable and complex schedule. We'll use event and date event hooks and a dialog to implement this.

```

<p:schedule value="#{scheduleBean.model}" editable="true" widgetVar="myschedule"
    eventSelectListener="#{scheduleBean.onEventSelect}"
    onEventSelectUpdate="eventDetails" onEventSelectComplete="eventDialog.show()"
    dateSelectListener="#{scheduleBean.onDateSelect}"
    onDateSelectUpdate="eventDetails" onDateSelectComplete="eventDialog.show()" />

<p:dialog widgetVar="eventDialog" header="Event Details"
    showEffect="clip" hideEffect="clip">
    <h:panelGrid id="eventDetails" columns="2">
        <h:outputLabel for="title" value="Title:" />
        <h:inputText id="title" value="#{scheduleBean.event.title}" required="true"/>

        <h:outputLabel for="from" value="From:" />
        <p:inputMask id="from" value="#{scheduleBean.event.startDate}" mask="99/99/9999">
            <f:convertDateTime pattern="dd/MM/yyyy" />
        </p:inputMask>

        <h:outputLabel for="to" value="To:" />
        <p:inputMask id="to" value="#{scheduleBean.event.endDate}" mask="99/99/9999">
            <f:convertDateTime pattern="dd/MM/yyyy" />
        </p:inputMask>

        <h:outputLabel for="allDay" value="All Day:" />
        <h:selectBooleanCheckbox id="allDay" value="#{scheduleBean.event.allDay}" />

        <p:commandButton type="reset" value="Reset" />
        <p:commandButton value="Save" actionListener="#{scheduleBean.addEvent}"
            oncomplete="myschedule.update();eventDialog.hide();"/>
    </h:panelGrid>
</p:dialog>

```

```

public class ScheduleBean {

    private ScheduleModel<ScheduleEvent> model;
    private ScheduleEventImpl event = new DefaultScheduleEvent();

    public ScheduleBean() {
        eventModel = new ScheduleModel<ScheduleEvent>();
    }

    public ScheduleModel<ScheduleEvent> getModel() { return model; }

    public ScheduleEventImpl getEvent() { return event; }

    public void setEvent(ScheduleEventImpl event) { this.event = event; }

    public void addEvent() {
        if(event.getId() == null)
            eventModel.addEvent(event);
        else
            eventModel.updateEvent(event);

        event = new DefaultScheduleEvent();      //reset dialog form
    }
}

```

```

public void onEventSelect(ScheduleEntrySelectEvent e) {
    event = e.getScheduleEvent();
}

public void onDateSelect(DateSelectEvent e) {
    event = new DefaultScheduleEvent("", e.getDate(), e.getDate());
}
}

```

## Lazy Loading

Schedule assumes whole set of events are eagerly provided in ScheduleModel, if you have a huge data set of events, lazy loading features would help to improve performance.

In lazy loading mode, only the events that belong to the displayed time frame are fetched whereas in default eager more all events need to be loaded.

```
<p:schedule value="#{scheduleBean.lazyModel}" />
```

To enable lazy loading of Schedule events, you just need to provide an instance of *org.primefaces.model.LazyScheduleModel* and implement the *loadEvents* methods. *loadEvents* method is called with new boundaries every time displayed timeframe is changed.

```

public class ScheduleBean {

    private ScheduleModel lazyModel;

    public ScheduleBean() {

        lazyModel = new LazyScheduleModel() {

            @Override
            public void loadEvents(Date start, Date end) {
                //addEvent(...);
                //addEvent(...);
            }
        };
    }

    public ScheduleModel getLazyModel() {
        return lazyModel;
    }
}

```

## Customizing Header

Header controls of Schedule can be customized based on templates, valid values of template options are;

- title: Text of current month/week/day information
- prev: Button to move calendar back one month/week/day.
- next: Button to move calendar forward one month/week/day.
- prevYear: Button to move calendar back one year
- nextYear: Button to move calendar forward one year
- today: Button to move calendar to current month/week/day.
- viewName: Button to change the view type based on the view type.

These controls can be placed at three locations on header which are defined with *leftHeaderTemplate*, *rightHeaderTemplate* and *centerTemplate* attributes.

```
<p:schedule value="#{scheduleBean.model}"
    leftHeaderTemplate="today"
    rightHeaderTemplate="prev,next"
    centerTemplate="month, agendaWeek, agendaDay"
/>
```

		<a href="#">today</a>	<a href="#">month</a>	<a href="#">week</a>	<a href="#">day</a>	<a href="#"></a>	<a href="#"></a>
Sun	Mon	Tue	Wed	Thu	Fri	Sat	
28	29	30	31	1	2	3	
4	5	6	7	8	9	10	

## Views

5 different views are supported, these are "month", "agendaWeek", "agendaDay", "basicWeek" and "basicDay".

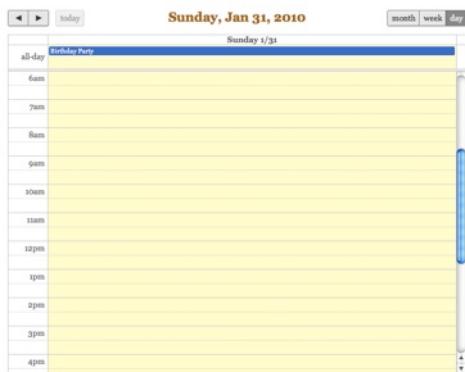
### agendaWeek

```
<p:schedule value="#{scheduleBean.model}" view="agendaWeek"/>
```

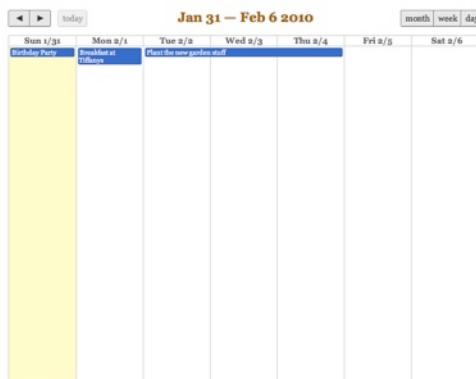
		<a href="#">today</a>	<b>Jan 31 – Feb 6 2010</b>					<a href="#">month</a>	<a href="#">week</a>	<a href="#">day</a>
Sun	Mon	Tue	Wed	Thu	Fri	Sat				
all-day	Birthday Party	Breakfast at Tiffany's	Plant the new garden stuff							
12am										
1am										
2am										
3am										
4am										
5am										
6am										
7am										
8am										
9am										
10am										

agendaDay

```
<p:schedule value="#{scheduleBean.model}" view="agendaDay"/>
```

basicWeek

```
<p:schedule value="#{scheduleBean.model}" view="basicWeek"/>
```

basicDay

```
<p:schedule value="#{scheduleBean.model}" view="basicDay"/>
```



## Locale Support

Schedule has built-in support for various languages and default is English. Locale information is retrieved from view locale and can be overridden to be a constant using locale attribute.

As view locale information is calculated by JSF, depending on user-agent information, schedule can automatically configure itself, as an example if the user is using a browser accepting primarily Turkish language, schedule will implicitly display itself in Turkish. Here is the full list of languages supported out of the box.

Key	Language
tr	Turkish
ca	Catalan
pt	Portuguese
it	Italian
fr	French
es	Spanish
de	German
ja	Japanese
fi	Finnish
hu	Hungarian

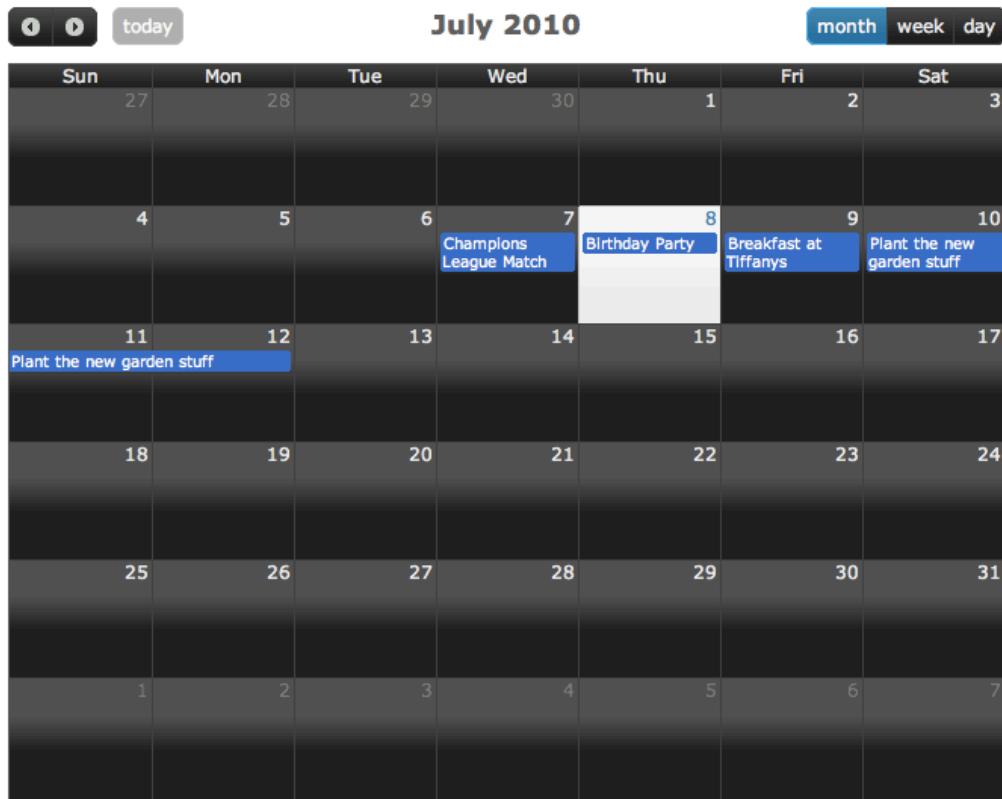
If you'd like to add a new language, feel free to apply a patch and contact PrimeFaces team for any questions. Each language translation is located in a javascript bundle object called PrimeFaces.widget.ScheduleResourceBundle. You can easily customize this object to add more languages in your application.

```
<script type="text/javascript">
    PrimeFaces.widget.ScheduleResourceBundle['key'] = {
        monthsNameShort: [],
        monthNames: [],
        dayNamesShort: [],
        today: "",
        month: "",
        week : "",
        day : "",
        allDayText : ""
    };
</script>
```

## Skinning

Schedule resides in a main container which *style* and *styleClass* attributes apply.

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.81 Separator

Separator displays a horizontal line to separate content.

---

### Info

Tag	<b>separator</b>
Component Class	<b>org.primefaces.component.separator.Separator</b>
Component Type	<b>org.primefaces.component.Separator</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.Separator</b>
Renderer Class	<b>org.primefaces.component.separator.Separator</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
title	null	String	Advisory tooltip information.
style	null	String	Inline style of the separator.
styleClass	null	String	Style class of the separator.

### Getting started with Separator

In its simplest form, separator is used as;

```
//content
<p:separator />
//content
```

## Dimensions

Separator renders a `<hr />` tag which style and styleClass options apply.

```
<p:separator style="width:500px;height:20px" />
```



## Skinning

As mentioned in dimensions section, style and styleClass options can be used to style the separator.

Following is the list of structural style classes;

Class	Applies
.ui-separator	Separator element

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.82 Slider

Slider is used to provide input with various customization options like orientation, display modes and skinning.



### Info

Tag	<code>slider</code>
Component Class	<code>org.primefaces.component.slider.Slider</code>
Component Type	<code>org.primefaces.component.Slider</code>
Component Family	<code>org.primefaces.component</code>
Renderer Type	<code>org.primefaces.component.SliderRenderer</code>
Renderer Class	<code>org.primefaces.component.slider.SliderRenderer</code>

### Attributes

Name	Default	Type	Description
<code>id</code>	null	String	Unique identifier of the component
<code>rendered</code>	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
<code>binding</code>	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
<code>for</code>	null	String	Id of the input text that the slider will be used for
<code>display</code>	null	String	Id of the component to display the slider value.
<code>minValue</code>	0	Integer	Minimum value of the slider
<code>maxValue</code>	100	Integer	Maximum value of the slider
<code>style</code>	null	String	Inline style of the container element
<code>styleClass</code>	null	String	Style class of the container element
<code>animate</code>	TRUE	Boolean	Boolean value to enable/disable the animated move when background of slider is clicked
<code>type</code>	horizontal	String	Sets the type of the slider, "horizontal" or "vertical".
<code>step</code>	1	Integer	Fixed pixel increments that the slider move in
<code>disabled</code>	FALSE	Boolean	Disables or enables the slider.

Name	Default	Type	Description
onSlideStart	null	String	Client side callback to execute when slide begins.
onSlide	null	String	Client side callback to execute during sliding.
onSlideEnd	null	String	Client side callback to execute when slide ends.
slideEndListener	null	MethodExpr	Server side method to execute when slide ends.
onSlideEndUpdate	null	String	Component(s) to update after ajax sliding.

## Getting started with Slider

Slider requires an input component to work with, *for* attribute is used to set the id of the input text component whose input will be provided by the slider.

```
public class SliderBean {

    private int number;

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }
}
```

```
<h:inputText id="number" value="#{sliderBean.number}" />
<p:slider for="number" />
```

## Display Value

Using *display* feature, you can present a readonly display value and still use slider to provide input, in this case *for* should refer to a hidden input to bind the value.

```
<h:inputHidden id="number" value="#{sliderBean.number}" />
<h:outputText value="Set ratio to %" />
<h:outputText id="output" value="#{sliderBean.number}" />

<p:slider for="number" display="output" />
```



## Vertical Slider

By default slider's orientation is horizontal, vertical sliding is also supported and can be set using the *type* attribute.

```
<h:inputText id="number" value="#{sliderController.number}" />
<p:slider for="number" type="vertical" minValue="0" maxValue="200"/>
```



## Step

Step factor defines the interval between each point during sliding. Default value is one and it is customized using *step* option.

```
<h:inputText id="number" value="#{sliderBean.number}" />
<p:slider for="number" step="10" />
```

## Animation

Sliding is animated by default, if you want to turn it off animate attribute set the *animate* option to false.

## Boundaries

Maximum and minimum boundaries for the sliding is defined using *minValue* and *maxValue* attributes. Following slider can slide between -100 and +100.

```
<h:inputText id="number" value="#{sliderBean.number}" />
<p:slider for="number" minValue="-100" maxValue="100"/>
```

## Client Side Callbacks

Slider provides three callbacks to hook-in your custom javascript, `onSlideStart`, `onSlide` and `onSlideEnd`. All of these callbacks receive two parameters; slide event and the ui object containing information about the event.

```
<h:inputText id="number" value="#{sliderBean.number}" />
<p:slider for="number" onSlideEnd="handleSlideEnd(event, ui)"/>
```

```
function handleSlideEnd(event, ui) {
    //ui.helper = Handle element of slider
    //ui.value = Current value of slider
}
```

## Ajax Slider

In case you need to invoke a method on a managed bean, bind a `slideEndListener` which takes an `org.primefaces.event.SlideEndEvent` as a parameter. Optionally `onSlideEndUpdate` option can be used to update other components on page after slide ends.

```
<h:inputText id="number" value="#{sliderBean.number}" />
<p:slider for="number" slideEndListener="#{sliderBean.onSlideEnd}"
          onSlideEndUpdate="msgs" />
<p:messages id="msgs" />
```

```
public class SliderBean {
    private int number;

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    public void onSlideEnd(SlideEndEvent event) {
        int value = event.getValue();
        //add faces message
    }
}
```

## Client Side API

Widget: *PrimeFaces.widget.Slider*

Method	Params	Return Type	Description
getValue()	-	Number	Returns the current value
setValue(value)	value: Value to set	void	Updates slider value with provided one.
disable()	index: Index of tab to disable	void	Disables slider.
enable()	index: Index of tab to enable	void	Enables slider.

## Skinning

Slider resides in a main container which *style* and *styleClass* attributes apply. These attributes are handy to specify the dimensions of the slider.

Following is the list of structural style classes;

Class	Applies
.ui-slider	Main container element
.ui-slider-horizontal	Main container element of horizontal slider
.ui-slider-vertical	Main container element of vertical slider
.ui-slider-handle	Slider handle

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.83 Spacer

Spacer is used to put spaces between elements.

### Info

Tag	<b>spacer</b>
Component Class	<b>org.primefaces.component.spacer.Spacer</b>
Component Type	<b>org.primefaces.component.Spacer</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.SpacerRenderer</b>
Renderer Class	<b>org.primefaces.component.spacer.SpacerRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
title	null	String	Advisory tooltip information.
style	null	String	Inline style of the spacer.
styleClass	null	String	Style class of the spacer.
width	null	String	Width of the space.
height	null	String	Height of the space.

### Getting started with Spacer

Spacer is used by either specifying width or height of the space.

Spacer in this example separates this text <p:spacer width="100" height="10"> and <p:spacer width="100" height="10"> this text.

Spacer in this example separates this text and this text.

## 3.84 Spinner

Spinner is an input component to provide a numerical input via increment and decrement buttons.



### Info

Tag	<b>spinner</b>
Component Class	<b>org.primefaces.component.spinner.Spinner</b>
Component Type	<b>org.primefaces.component.Spinner</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.SpinnerRenderer</b>
Renderer Class	<b>org.primefaces.component.spinner.SpinnerRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Value of the component than can be either an EL expression or a literal text
converter	null	Converter/String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id
immediate	FALSE	Boolean	Boolean value that specifies the lifecycle phase the valueChangeEvents should be processed, when true the events will be fired at "apply request values", if immediate is set to false, valueChange Events are fired in "process validations" phase
required	FALSE	Boolean	Marks component as required
validator	null	Method Expr	A method binding expression that refers to a method validating the input

Name	Default	Type	Description
valueChangeListener	null	Method Expr	A method binding expression that refers to a method for handling a valuechangeevent
requiredMessage	null	String	Message to be displayed when required field validation fails.
converterMessage	null	String	Message to be displayed when conversion fails.
validatorMessage	null	String	Message to be displayed when validation fields.
widgetVar	null	String	Name of the client side widget.
stepFactor	1	Double	Stepping factor for each increment and decrement
min	null	Double	Minimum boundary value
max	null	Double	Maximum boundary value
prefix	null	String	Prefix of the input
suffix	null	String	Suffix of the input
showOn	null	String	Defines when the the buttons would be displayed.
width	null	Integer	Width of the buttons
accesskey	null	String	Access key that when pressed transfers focus to the input element.
alt	null	String	Alternate textual description of the input field.
autocomplete	null	String	Controls browser autocomplete behavior.
dir	null	String	Direction indication for text that does not inherit directionality. Valid values are LTR and RTL.
disabled	FALSE	Boolean	Disables input field
label	null	String	A localized user presentable name.
lang	null	String	Code describing the language used in the generated markup for this component.
maxlength	null	Integer	Maximum number of characters that may be entered in this field.
onblur	null	String	Client side callback to execute when input element loses focus.
onchange	null	String	Client side callback to execute when input element loses focus and its value has been modified since gaining focus.
onclick	null	String	Client side callback to execute when input element is clicked.
ondblclick	null	String	Client side callback to execute when input element is double clicked.

Name	Default	Type	Description
onfocus	null	String	Client side callback to execute when input element receives focus.
onkeydown	null	String	Client side callback to execute when a key is pressed down over input element.
onkeypress	null	String	Client side callback to execute when a key is pressed and released over input element.
onkeyup	null	String	Client side callback to execute when a key is released over input element.
onmousedown	null	String	Client side callback to execute when a pointer button is pressed down over input element
onmousemove	null	String	Client side callback to execute when a pointer button is moved within input element.
onmouseout	null	String	Client side callback to execute when a pointer button is moved away from input element.
onmouseover	null	String	Client side callback to execute when a pointer button is moved onto input element.
onmouseup	null	String	Client side callback to execute when a pointer button is released over input element.
onselect	null	String	Client side callback to execute when text within input element is selected by user.
readonly	FALSE	Boolean	Flag indicating that this component will prevent changes by the user.
size	null	Integer	Number of characters used to determine the width of the input element.
style	null	String	Inline style of the input element.
styleClass	null	String	Style class of the input element.
tabindex	null	Integer	Position of the input element in the tabbing order.
title	null	String	Advisory tooltip information.

## Getting Started with Spinner

Spinner is an input component and used just like a standard input text.

```
public class SpinnerBean {
    private int number;
    //getter and setter
}
```

```
<p:spinner value="#{spinnerBean.number}" />
```

## Step Factor

Other than integers, spinner also support decimals so the fractional part can be controlled with spinner as well. For decimals use the stepFactor attribute to specify stepping amount. Following example uses a stepFactor of 0.25.

```
<p:spinner value="#{spinnerBean.number}" stepFactor="0.25"/>
```

```
public class SpinnerBean {  
    private double number;  
    //getter and setter  
}
```

Output of this spinner would be;



After an increment happens a couple of times.



## Prefix and Suffix

Prefix and Suffix options enable placing fixed strings on input field. Note that you would need to use a converter to avoid conversion errors since prefix/suffix will also be posted.

```
<p:spinner value="#{spinnerBean.number}" prefix="$">  
    <f:convertNumber currencySymbol="$" type="currency" />  
</p:spinner>
```



## Boundaries

In order to restrict the boundary values, use *min* and *max* options.

```
<p:spinner value="#{spinnerBean.number}" min="0" max="100"/>
```

## Button Width

Button width is specified in pixels and customized with *width* option.

```
<p:spinner value="#{spinnerBean.number}" width="32" />
```



## Ajax Spinner

Spinner can be ajaxified using client behaviors like f:ajax or p:ajax. In example below, an ajax request is done to update the outputtext with new value whenever a spinner button is clicked.

```
<p:spinner value="#{spinnerBean.number}">
    <p:ajax update="display" />
</p:spinner>

<h:outputText id="display" value="#{spinnerBean.number}" />
```

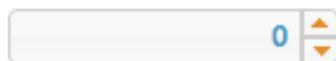
## Skinning

Spinner consists of an input field and two buttons, input field is styled using *style* and *styleClass* options.

Following is the list of structural style classes;

Class	Applies
.ui-spinner	Main container element of spinner
.ui-spinner-buttons	Container of buttons
.ui-spinner-up	Increment button
.ui-spinner-down	Decrement button

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.85 Submenu

Submenu is nested in menu components and represents a sub menu items.

### Info

<b>Tag</b>	<b>submenu</b>
Component Class	<b>org.primefaces.component.submenu.Submenu</b>
Component Type	<b>org.primefaces.component.Submenu</b>
Component Family	<b>org.primefaces.component</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
label	null	String	Label of the submenu header.
icon	null	String	Icon of a submenu, see menuitem to see how it is used

### Getting started with Submenu

Please see Menu or Menubar section to find out how submenu is used with the menus.

## 3.86 Stack

Stack is a navigation component that mimics the stacks feature in Mac OS X.



### Info

Tag	<b>stack</b>
Component Class	<b>org.primefaces.component.stack.Stack</b>
Component Type	<b>org.primefaces.component.Stack</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.StackRenderer</b>
Renderer Class	<b>org.primefaces.component.stack.StackRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
icon	null	String	An optional image to contain stacked items.
openSpeed	300	String	Speed of the animation when opening the stack.
closeSpeed	300	Integer	Speed of the animation when opening the stack.
widgetVar	null	String	Javascript variable name of the client side widget.
model	null	MenuModel	MenuModel instance to create menus programmatically

## Getting started with Stack

Each item in the stack is represented with menuitems. Stack below has five items with different icons and labels.

```
<p:stack icon="/images/stack/stack.png">
    <p:menuitem value="Aperture" icon="/images/stack/aperture.png" url="#" />
    <p:menuitem value="Photoshop" icon="/images/stack/photoshop.png" url="#" />
    //...
</p:stack>
```

Initially stack will be rendered in collapsed mode;



## Location

Stack is a fixed positioned element and location can be change via css. There's one important css selector for stack called *.ui-stack*. Override this style to change the location of stack.

```
.ui-stack {
    bottom: 28px;
    right: 40px;
}
```

## Dynamic Menus

Menus can be created programmatically as well, see the dynamic menus part in menu component section for more information and an example.

## Skinning

Class	Applies
.ui-stack	Main container element of stack
.ui-stack ul li	Each item in stack
.ui-stack ul li img	Icon of a stack item
.ui-stack ul li span	Label of a stack item

## 3.87 Tab

Tab is a generic container component used by other PrimeFaces components such as tabView or accordionPanel.

### Info

Tag	<b>tabView</b>
Component Class	<b>org.primefaces.component.TabView.Tab</b>
Component Type	<b>org.primefaces.component.Tab</b>
Component Family	<b>org.primefaces.component</b>

### Attributes

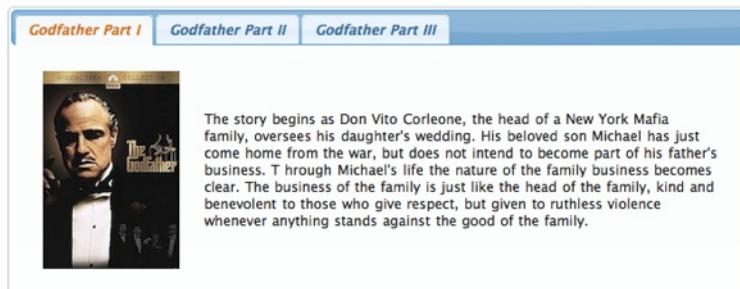
Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
title	null	Boolean	Title text of the tab

### Getting started with the Tab

See the sections of components who utilize tab component for more information.

## 3.88 TabView

TabView is a tabbed panel component featuring client side tabs, dynamic content loading with ajax and content transition effects.



### Info

Tag	<b>tabView</b>
Component Class	<b>org.primefaces.component.tabview.TabView</b>
Component Type	<b>org.primefaces.component.TabView</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.TabViewRenderer</b>
Renderer Class	<b>org.primefaces.component.tabview.TabViewRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
widgetVar	null	String	Variable name of the client side widget.
activeIndex	0	Integer	Index of the active tab.
effect	null	String	Name of the transition effect.
effectDuration	null	String	Duration of the transition effect.
dynamic	FALSE	Boolean	Specifies the toggleMode.

Name	Default	Type	Description
cache	TRUE	Boolean	When tab contents are lazy loaded by ajax toggleMode, caching only retrieves the tab contents once and subsequent toggles of a cached tab does not communicate with server. If caching is turned off, tab contents are reloaded from server each time tab is clicked.
collapsible	FALSE	Boolean	Specifies if all tabs can be collapsed.
event	click	String	Dom event to use to activate a tab.
tabChangeListener	null	MethodExpr	Server side listener to invoke when a tab changes.
onTabChangeUpdate	null	String	Component(s) to update after invoking a tabChangeListener.
onTabChange	null	String	Client side callback to execute when a tab is clicked.
onTabShow	null	String	Client side callback to execute when a tab is shown.
style	null	String	Inline style of the main container.
styleClass	null	String	Style class of the main container.

## Getting started with the TabView

TabView requires one more child tab components to display.

```
<p:tabView>
    <p:tab title="Tab One">
        <h:outputText value="Lorem" />
    </p:tab>
    <p:tab title="Tab Two">
        <h:outputText value="Ipsum" />
    </p:tab>
    <p:tab title="Tab Three">
        <h:outputText value="Dolor" />
    </p:tab>
</p:tabView>
```

## Dynamic Tabs

There're two toggleModes in tabview, *non-dynamic* (default) and *dynamic*. By default, all tab contents are rendered to the client, on the other hand in dynamic mode, only the active tab contents are rendered and when an inactive tab header is selected, content is loaded with ajax. Dynamic mode is handy in reducing page size, since inactive tabs are lazy loaded, pages will load faster. To enable dynamic loading, simply set *dynamic* option to true.

```
<p:tabView dynamic="true">
    //tabs
</p:tabView>
```

## Content Caching

Dynamically loaded tabs cache their contents by default, by doing so, reactivating a tab doesn't result in an ajax request since contents are cached. If you want to reload content of a tab each time a tab is selected, turn off caching by *cache* to false.

```
<p:tabView dynamic="true" cache="false">
    //tabs
</p:tabView>
```

## Effects

Content transition effects are controlled with *effect* and *effectDuration* attributes. *opacity*, *height* and *width* are available choices for effect to use. EffectDuration specifies the speed of the effect, *slow*, *normal* (default) and *fast* are the valid options.

```
<p:tabView effect="opacity" effectDuration="fast">
    //tabs
</p:tabView>
```

## TabChangeListener

In case you need to invoke a server side method when an inactive tab is clicked, bind a *tabChangeListener*. Your method will be invoked by passing an *org.primefaces.event.TabChangeEvent* as a parameter, optionally other component(s) on page can be updated with *onTabChangeUpdate* option.

```
<p:tabView tabChangeListener="#{tabBean.onChange}" onTabChangeUpdate="msgs">
    //tabs
</p:tabView>

<p:messages id="msgs" />
```

```
public class TabBean {

    public void onChange(TabChangeEvent event) {
        Tab newTab = event.getTab();
        //add facesmessage
    }
}
```

For both dynamic loading and tabChangeListener features to work, at least one form needs to present on page, location of the form does not matter.

## Client Side Callbacks

Similar to tabChangeListener on the server side, tabview has two callbacks for client side as well. *onTabChange* is executed when an inactive tab is clicked and *onTabShow* is executed when an inactive tab becomes active to be shown. Both events get two parameters containing information about the state.

```
<p:tabView onTabChange="handleTabChange(event, ui)">
    //tabs
</p:tabView>
```

```
function handleTabChange(event, ui) {
    //ui.tab = title element of the selected tab
    //ui.panel = container element of the selected tab
    //ui.index = index of the selected tab
}
```

## Client Side API

Widget: *PrimeFaces.widget.TabView*

Method	Params	Return Type	Description
select(index)	index: Index of tab to display	void	Activates tab with given index
selectTab(index)	index: Index of tab to display	void	(Deprecated, use select instead) Activates tab with given index
disable(index)	index: Index of tab to disable	void	Disables tab with given index
enable(index)	index: Index of tab to enable	void	Enables tab with given index
add(url, label, index)	url: Local url to load content label: Title of the new tab index: Index to add new tab	void	Adds a new tab by loading content from a local url
remove(index)	index: Index of tab to remove	void	Removes tab with given index
getLength()	-	Number	Returns the number of tabs
getActiveIndex()	-	Number	Returns index of current tab

## Skinning

Following is the list of structural style classes;

Class	Applies
.ui-tabs	Main tabview container element
.ui-tabs-nav	Main container of tab headers
.ui-tabs-panel	Each tab container

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.89 Terminal

Terminal is an ajax powered web based terminal that brings desktop terminals to JSF.

```
Welcome to PrimeFaces Terminal, how are you today?
prime $ date
Wed Jan 12 13:29:13 EET 2011
prime $ greet Optimus
Hello Optimus
prime $ xyz
xyz not found
prime $ |
```

### Info

<b>Tag</b>	<b>terminal</b>
Component Class	<b>org.primefaces.component.terminal.Terminal</b>
Component Type	<b>org.primefaces.component.Terminal</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.TerminalRenderer</b>
Renderer Class	<b>org.primefaces.component.terminal.TerminalRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
width	null	String	Width of the terminal
height	null	String	Height of the terminal
welcomeMessage	null	String	Welcome message to be displayed on initial load.
prompt	prime \$	String	Primary prompt text.

Name	Default	Type	Description
commandHandler	null	MethodExpr	Method to be called with arguments to process.
widgetVar	null	String	Name of the client side widget.

## Getting started with the Terminal

A command handler is necessary to interpret commands entered in terminal.

```
<p:terminal commandHandler="#{terminalBean.handleCommand}" />
```

```
public class TerminalBean {

    public String handleCommand(String command, String[] params) {
        if(command.equals("greet"))
            return "Hello " + params[0];
        else if(command.equals("date"))
            return new Date().toString();
        else
            return command + " not found";
    }
}
```

Whenever a command is sent to the server, handleCommand method is invoked with the command name and the command arguments as a String array.

## Focus

To add focus on terminal, use client side api, following example shows how to add focus on a terminal nested inside a dialog;

```
<p:commandButton type="Show Terminal" type="button"
    onclick="dlg.show();term.focus();"/>

<p:dialog widgetVar="dlg" width="600" height="400" header="Terminal">
    <p:terminal widgetVar="term"
        commandHandler="#{terminalBean.handleCommand}" width="590px" />
</p:dialog>
```

## 3.90 ThemeSwitcher

ThemeSwitcher enables switching PrimeFaces themes on the fly with no page refresh.



### Info

Tag	<b>themeSwitcher</b>
Component Class	<b>org.primefaces.component.themeswitcher.ThemeSwitcher</b>
Component Type	<b>org.primefaces.component.ThemeSwitcher</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.ThemeSwitcherRenderer</b>
Renderer Class	<b>org.primefaces.component.themeswitcher.ThemeSwitcherRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
theme	null	String	Default theme to load.
width	150	Integer	Width of switcher menu.

Name	Default	Type	Description
height	200	Integer	Height of switcher menu.
buttonHeight	14	Integer	Height of switcher button.
initialText	Switch Theme	String	Initial text to display before a theme is chosen.
buttonPreText	Theme:	String	Prefix text displayed on button.
onSelect	null	String	Client side callback to execute when a theme is selected.
widgetVar	null	String	Name of the client side widget.

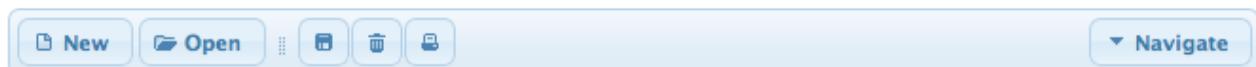
## Getting started with the ThemeSwitcher

In it's simplest form, themeSwitcher is used with no required setting. ThemeSwitcher loads the selected themes from jQuery UI project page so online connection is required.

```
<p:themeSwitcher />
```

## 3.91 Toolbar

Toolbar is a horizontal grouping component for commands and other content.



### Info

Tag	<b>toolbar</b>
Component Class	<b>org.primefaces.component.toolbar.Toolbar</b>
Component Type	<b>org.primefaces.component.Toolbar</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.ToolbarRenderer</b>
Renderer Class	<b>org.primefaces.component.toolbar.ToolbarRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
style	null	String	Inline style of the container element.
styleClass	null	String	Style class of the container element.

### Getting Started with the Toolbar

Toolbar has two placeholders(left and right) that are defined with toolbarGroup component.

```
<p:toolbar>
    <p:toolbarGroup align="left">
    </p:toolbarGroup>

    <p:toolbarGroup align="right">
    </p:toolbarGroup>
</p:toolbar>
```

Any number of components can be placed inside toolbarGroups. Additionally p:divider component can be used to separate items in toolbar. Here is an example;

```
<p:toolbar>
    <p:toolbarGroup align="left">
        <p:commandButton type="push" value="New" image="ui-icon-document" />
        <p:commandButton type="push" value="Open" image="ui-icon-folder-open"/>

        <p:divider />

        <p:commandButton type="push" title="Save" image="ui-icon-disk"/>
        <p:commandButton type="push" title="Delete" image="ui-icon-trash"/>
        <p:commandButton type="push" title="Print" image="ui-icon-print"/>
    </p:toolbarGroup>

    <p:divider />

    <p:toolbarGroup align="right">
        <p:menuButton value="Navigate">
            <p:menuitem value="Home" url="#" />
            <p:menuitem value="Logout" url="#" />
        </p:menuButton>
    </p:toolbarGroup>
</p:toolbar>
```

## Skinning

Toolbar resides in a container element which *style* and *styleClass* options apply.

Following is the list of structural style classes;

Style Class	Applies
.ui-toolbar	Main container
.ui-toolbar .ui-divider	Divider in a toolbar
.ui-toolbar-group-left	Left toolbarGroup container
.ui-toolbar-group-right	Right toolbarGroup container

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



## 3.92 ToolbarGroup

ToolbarGroup is a helper component for Toolbar component to define placeholders.

### Info

Tag	<b>toolbarGroup</b>
Component Class	<b>org.primefaces.component.toolbar.ToolbarGroup</b>
Component Type	<b>org.primefaces.component.ToolbarGroup</b>
Component Family	<b>org.primefaces.component</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
align	null	String	Defines the alignment within toolbar, valid values are <i>left</i> and <i>right</i> .
style	null	String	Inline style of the container element.
styleClass	null	String	Style class of the container element.

### Getting Started with the ToolbarGroup

See toolbar documentation for more information about how Toolbar Group is used.

## 3.93 Tooltip

Tooltip goes beyond the legacy html title attribute by providing custom effects, events, html content and advance theme support.



### Info

Tag	<b>tooltip</b>
Component Class	<b>org.primefaces.component.tooltip.Tooltip</b>
Component Type	<b>org.primefaces.component.Tooltip</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.TooltipRenderer</b>
Renderer Class	<b>org.primefaces.component.tooltip.TooltipRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Value of the component than can be either an EL expression or a literal text
converter	null	Converter/ String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id
widgetVar	null	String	Name of the client side widget.
global	FALSE	Boolean	A global tooltip converts each title attribute to a tooltip.
targetPosition	bottomRight	String	The corner of the target element to position the tooltip by.
position	topLeft	String	The corner of the tooltip to position the target's position.

Name	Default	Type	Description
showEvent	mouseover	String	Event displaying the tooltip.
showDelay	140	Integer	Delay time for displaying the tooltip.
showEffect	fade	String	Effect to be used for displaying.
showEffectLength	100	Integer	Time in milliseconds to display the effect.
hideEvent	mouseout	String	Event hiding the tooltip.
hideDelay	0	Integer	Delay time for hiding the tooltip.
hideEffect	fade	String	Effect to be used for hiding.
hideEffectLength	100	Integer	Time in milliseconds to process the hide effect.
for	null	String	Id of the component to attach the tooltip.
forElement	null	String	Id of the html element to attach the tooltip.

## Getting started with the Tooltip

Tooltip is used by nesting it as a child of its target. Tooltip below sets a tooltip on the input field.

```
<h:inputSecret id="pwd" value="#{myBean.password}" />
<p:tooltip for="pwd" value="Password must contain only numbers"/>
```

## Global Tooltip

One powerful feature of tooltip is using title attributes of other JSF components to create the tooltips, in this case you only need to place one tooltip to your page. This would also perform better compared to defining a tooltip for each component.

```
<p:tooltip global="true" />
```

## Effects

Showing and Hiding of tooltip along with the effect durations can be customized easily.

```
<h:inputSecret id="pwd" value="#{myBean.password}" />
<p:tooltip for="pwd" value="Password must contain only numbers"
    showEffect="slide" hideEffect="slide"
    showEffectLength="2000" hideEffectLength="2000"/>
```

*fade, slide and grow* are available options for effects.

## Events

A tooltip is shown on mouseover event and hidden when mouse is out by default. If you need to change this behaviour use the showEvent and hideEvent feature. Tooltip below is displayed when the input gets the focus and hidden with onblur.

```
<h:inputSecret id="pwd" value="#{myBean.password}" />
<p:tooltip for="pwd" value="Password must contain only numbers"
    showEvent="focus" hideEvent="blur"/>
```

## Delays

There're sensable defaults for each delay to display the tooltips and these can be configured easily as follows;

```
<h:inputSecret id="pwd" value="#{myBean.password}" />
<p:tooltip for="pwd" value="Password must contain only numbers"
    showDelay="2000" hideDelay="2000"/>
```

Tooltip above waits for 2 seconds to show and hide itself.

## Html Content

Another powerful feature of tooltip is the ability to display custom content as a tooltip not just plain texts. An example is as follows;

```
<h:outputLink id="lnk" value="#">  
    <h:outputText value="PrimeFaces Home" />  
</h:outputLink>  
  
<p:tooltip for="lnk">  
    <p:graphicImage value="/images/prime_logo.png" />  
    <h:outputText value="Visit PrimeFaces Home" />  
</p:tooltip>
```

## Ajax Updates

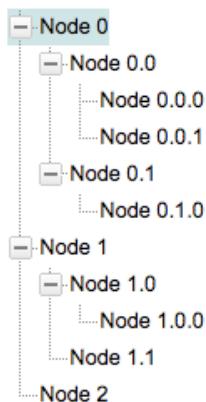
Global tooltips will cause duplicate elements on page as they are appended to document body, use inline tooltips instead if you need to update tooltips as well.

## Skinning

Tooltip is styled with global skinning selectors, see main skinning section for more information.

## 3.94 Tree

Tree is used for displaying hierarchical data and creating site navigations.



### Info

Tag	<code>tree</code>
Component Class	<code>org.primefaces.component.tree.Tree</code>
Component Type	<code>org.primefaces.component.Tree</code>
Component Family	<code>org.primefaces.component</code>
Renderer Type	<code>org.primefaces.component.TreeRenderer</code>
Renderer Class	<code>org.primefaces.component.tree.TreeRenderer</code>

### Attributes

Name	Default	Type	Description
<code>id</code>	null	String	Unique identifier of the component
<code>rendered</code>	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
<code>binding</code>	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
<code>value</code>	null	Object	A TreeNode instance as the backing model.
<code>var</code>	null	String	Name of the request-scoped variable that'll be used to refer each treenode data during rendering
<code>dynamic</code>	FALSE	Boolean	Specifies the ajax/client toggleMode

Name	Default	Type	Description
expandAnim	null	String	Animation to be displayed on node expand, valid values are "FADE_IN" or "FADE_OUT"
collapseAnim	null	String	Animation to be displayed on node collapse, valid values are "FADE_IN" or "FADE_OUT"
nodeSelectListener	null	MethodExpr	Method expression to listen node select events
nodeExpandListener	null	MethodExpr	Method expression to listen node expand events
nodeCollapseListener	null	MethodExpr	Method expression to listen node collapse events
cache	TRUE	Boolean	Specifies caching on dynamically loaded nodes. When set to true expanded nodes will be kept in memory.
widgetVar	null	String	Javascript variable name of the wrapped widget
onNodeClick	null	String	Javascript event to process when a tree node is clicked.
expanded	FALSE	Boolean	When set to true, all nodes will be displayed as expanded on initial page load.
update	null	String	Id(s) of component(s) to update after node selection
onselectStart	null	String	Javascript event handler to process before instant ajax selection request.
onselectComplete	null	String	Javascript event handler to process after instant ajax selection request.
selection	null	Object	TreeNode array to reference the selections.
style	null	String	Style of the main container element of tree
styleClass	null	String	Style class of the main container element of tree
propagateSelectionUp	FALSE	Boolean	Specifies if selection will be propagated up to the parents of clicked node
propagateSelectionDown	FALSE	Boolean	Specifies if selection will be propagated down to the children of clicked node
SelectionMode	null	String	Defines the selectionMode

## Getting started with the Tree

Tree is populated with a `org.primefaces.model.TreeNode` instance which corresponds to the root. TreeNode API has a hierarchical data structure and represents the data to be populated in tree.

```
public class TreeBean {

    private TreeNode root;

    public TreeBean() {
        root = new TreeNode("Root", null);
        TreeNode node0 = new TreeNode("Node 0", root);
        TreeNode node1 = new TreeNode("Node 1", root);
        TreeNode node2 = new TreeNode("Node 2", root);

        TreeNode node00 = new TreeNode("Node 0.0", node0);
        TreeNode node01 = new TreeNode("Node 0.1", node0);

        TreeNode node10 = new TreeNode("Node 1.0", node1);
        TreeNode node11 = new TreeNode("Node 1.1", node1);

        TreeNode node000 = new TreeNode("Node 0.0.0", node00);
        TreeNode node001 = new TreeNode("Node 0.0.1", node00);
        TreeNode node010 = new TreeNode("Node 0.1.0", node01);

        TreeNode node100 = new TreeNode("Node 1.0.0", node10);
    }

    public TreeNode getModel() {
        return root;
    }
}
```

Once model is instantiated via TreeNodes, bind the model to the tree as the value and specify a UI `treeNode` component as a child to display the nodes.

```
<p:tree value="#{treeBean.root}" var="node">
    <p:treeNode>
        <h:outputText value="#{node}" />
    </p:treeNode>
</p:tree>
```

## TreeNode vs p:TreeNode

You might get confused about the `TreeNode` and the `p:treeNode` component. `TreeNode` API is used to create the node model and consists of `org.primefaces.model.TreeNode` instances, on the other hand `<p:treeNode />` tag represents a component of type `org.primefaces.component.tree.UITreeNode`. You can bind a `TreeNode` to a particular `p:treeNode` using the `type` name. Document Tree example in upcoming section demonstrates a sample usage.

## TreeNode API

TreeNode has a simple API to use when building the backing model. For example if you call node.setExpanded(true) on a particular node, tree will render that node as expanded.

Property	Type	Description
type	String	type of the treeNode name, default type name is "default".
data	Object	Encapsulated data
children	List<TreeNode>	List of child nodes
parent	TreeNode	Parent node
expanded	Boolean	Flag indicating whether the node is expanded or not

## Dynamic Tree

Tree is non-dynamic by default and toggling happens on client-side. In order to enable ajax toggling set dynamic setting to true.

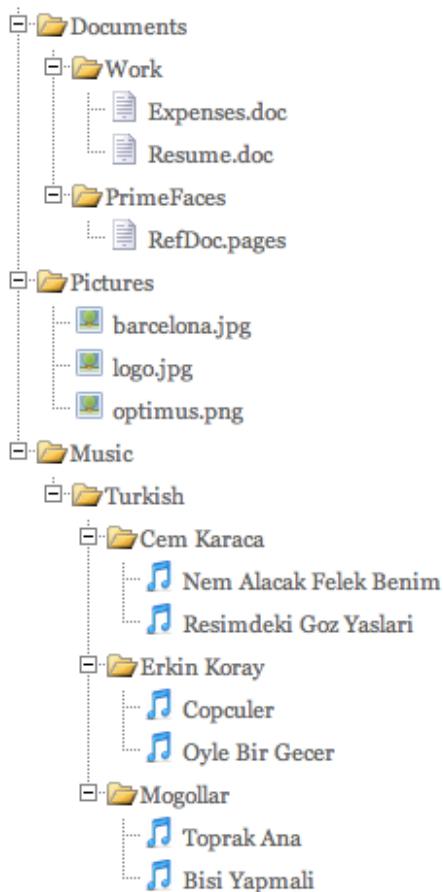
```
<p:tree value="#{treeBean.root}" var="node" dynamic="true">
    <p:treeNode>
        <h:outputText value="#{node}" />
    </p:treeNode>
</p:tree>
```

*Non-Dynamic:* When toggling is set to client all the treenodes in model are rendered to the client and tree is created, this mode is suitable for relatively small datasets and provides fast user interaction. On the otherhand it's not suitable for large data since all the data is sent to the client.

*Dynamic:* Dynamic mode uses ajax to fetch the treenodes from server side on demand, compared to the client toggling, dynamic mode has the advantage of dealing with large data because only the child nodes of the root node is sent to the client initially and whole tree is lazily populated. When a node is expanded, tree only loads the children of the particular expanded node and send to the client for display.

## Multiple TreeNode Types

It's a common requirement to display different TreeNode types with a different UI (eg icon). Suppose you're using tree to visualize a company with different departments and different employees, or a document tree with various folders, files each having a different formats (music, video). In order to solve this, you can place more than one `<p:treeNode />` components each having a different type and use that "type" to bind TreeNode's in your model. Following example demonstrates a document explorer. To begin with here is the final output;



Document Explorer is implemented with four different `<p:treeNode />` components and additional CSS skinning to visualize expanded/closed folder icons.

## Tree Definition

```
<p:tree value="#{documentsController.root}" var="doc">
    <p:treeNode>
        <h:outputText value="#{doc}" />
    </p:treeNode>

    <p:treeNode type="document">
        <h:outputText value="#{doc}" styleClass="documentStyle"/>
    </p:treeNode>

    <p:treeNode type="picture">
        <h:outputText value="#{doc}" styleClass="pictureStyle"/>
    </p:treeNode>

    <p:treeNode type="mp3">
        <h:outputText value="#{doc}" styleClass="mp3Style"/>
    </p:treeNode>
</p:tree>
```

```

public class DocumentsController {

    private TreeNode root;

    public DocumentsController() {
        root = new TreeNode("root", null);

        TreeNode documents = new TreeNode("Documents", root);
        TreeNode pictures = new TreeNode("Pictures", root);
        TreeNode music = new TreeNode("Music", root);

        TreeNode work = new TreeNode("Work", documents);
        TreeNode primefaces = new TreeNode("PrimeFaces", documents);

        //Documents
        TreeNode expenses = new TreeNode("document", "Expenses.doc", work);
        TreeNode resume = new TreeNode("document", "Resume.doc", work);
        TreeNode refdoc = new TreeNode("document", "RefDoc.pages", primefaces);

        //Pictures
        TreeNode barca = new TreeNode("picture", "barcelona.jpg", pictures);
        TreeNode primelogo = new TreeNode("picture", "logo.jpg", pictures);
        TreeNode optimus = new TreeNode("picture", "optimus.png", pictures);

        //Music
        TreeNode turkish = new TreeNode("Turkish", music);

        TreeNode cemKaraca = new TreeNode("Cem Karaca", turkish);
        TreeNode erkinKoray = new TreeNode("Erkin Koray", turkish);
        TreeNode mogollar = new TreeNode("Mogollar", turkish);

        TreeNode nemalacak = new TreeNode("mp3", "Nem Alacak Felek Benim", cemKaraca);
        TreeNode resimdeki = new TreeNode("mp3", "Resimdeki Goz Yaslari", cemKaraca);

        TreeNode copculer = new TreeNode("mp3", "Copculer", erkinKoray);
        TreeNode oylebirgecer = new TreeNode("mp3", "Oyle Bir Gecer", erkinKoray);

        TreeNode toprakana = new TreeNode("mp3", "Toprak Ana", mogollar);
        TreeNode bisiyapmali = new TreeNode("mp3", "Bisi Yapmali", mogollar);
    }

    public TreeNode getRoot() {
        return root;
    }
}

```

```
.nodeContent { margin-left:20px; }
.documentStyle {background: url(doc.png) no-repeat; }
.pictureStyle {background: url(picture.png) no-repeat; }
.mp3Style {background: url(mp3.png) no-repeat; }

/* Folder Theme */
.ygtvtn {background:url(tn.gif) 0 0 no-repeat; width:17px;height:22px; }
.ygtvtm {background:url(tm.gif) 0 0 no-repeat; width:34px;height:22px;
cursor:pointer}
.ygtvtmh {background:url(tmh.gif) 0 0 no-repeat; width:34px; height:22px;
cursor:pointer}
.ygtvtp {background:url(tp.gif) 0 0 no-repeat; width:34px; height:22px;
cursor:pointer}
.ygtvtph { background: url(tph.gif) 0 0 no-repeat; width:34px; height:22px;
cursor:pointer }
.ygtvln { background: url(ln.gif) 0 0 no-repeat; width:17px; height:22px; }
.ygtvlm { background: url(lm.gif) 0 0 no-repeat; width:34px; height:22px;
cursor:pointer }
.ygtvlmh { background: url(lmh.gif) 0 0 no-repeat; width:34px; height:22px;
cursor:pointer }
.ygtvlp { background: url(lp.gif) 0 0 no-repeat; width:34px; height:22px;
cursor:pointer }
.ygtvlph { background: url(lph.gif) 0 0 no-repeat; width:34px; height:22px;
cursor:pointer }
```

Integration between a TreeNode and a p:treeNode is the type attribute, for example music files in document explorer are represented using TreeNodes with type "mp3", there's also a p:treeNode component with same type "mp3". This results in rendering all music nodes using that particular p:treeNode representation which displays a note icon. Similarly document and pictures have their own p:treeNode representations.

Folders on the other hand have various states like open, closed, open mouse over, closed mouseover and more. These states are easily skinned with predefined CSS selectors, see skinning section for more information.

## Event Handling

Tree is an interactive component, it can notify both client and server side listeners about certain events. There're currently three events supported, node select, expand and collapse. For example when a node is expanded and a server side nodeExpandListener is defined on tree, the particular java method is executed with the NodeExpandEvent. Following tree has three listeners;

```
<p:tree value="#{treeBean.model}" dynamic="true"
    nodeSelectListener="#{treeBean.onNodeSelect}"
    nodeExpandListener="#{treeBean.onNodeExpand}"
    nodeCollapseListener="#{treeBean.onNodeCollapse}">
    ...
</p:tree>
```

The server side listeners are simple method expressions like;

```

public void onNodeSelect(NodeSelectEvent event) {
    String node = event.getTreeNode().getData().toString();
    logger.info("Selected:" + node);
}

public void onNodeExpand(NodeExpandEvent event) {
    String node = event.getTreeNode().getData().toString();
    logger.info("Expanded:" + node);
}

public void onNodeCollapse(NodeCollapseEvent event) {
    String node = event.getTreeNode().getData().toString();
    logger.info("Collapsed:" + node);
}

```

Event listeners are also useful when dealing with huge amount of data. The idea for implementing such a use case would be providing only the root and child nodes to the tree, use event listeners to get the selected node and add new nodes to that particular tree at runtime.

## Selection Modes

Node selection is a built-in feature of tree and it supports three different modes. Selection should be a TreeNode for single case and an array of TreeNodes for multiple and checkbox cases, tree finds the selected nodes and assign them to your selection model.

*single*: Only one at a time can be selected.

*multiple*: Multiple nodes can be selected.

*checkbox*: Multiple selection is done with checkbox UI.

```

<p:tree value="#{treeBean.root}" var="node"
         selectionMode="single|multiple|checkbox"
         selection="#{treeBean.selectedNodes}">
    <p:treeNode>
        <h:outputText value="#{node}" />
    </p:treeNode>
</p:tree>

```

```

public class TreeBean {

    private TreeNode root;

    private TreeNode[] selectedNodes;

    public TreeBean() {
        root = new TreeNode("Root", null);
        //populate nodes
    }

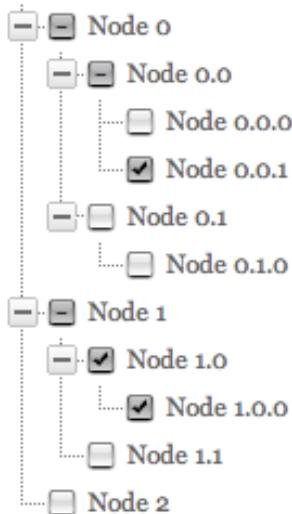
    //getters and setters
}

```

Note that for checkbox based selection, following CSS might be necessary to add for better indentation.

```
.ygtv-checkbox .ygtv-highlight0 .ygtvcontent,
.ygtv-checkbox .ygtv-highlight1 .ygtvcontent,
.ygtv-checkbox .ygtv-highlight2 .ygtvcontent {
    padding-left:20px;
}
```

That's it, now the checkbox based tree looks like below. When the form is submitted with a command component like a button, selected nodes will be populated in selectedNodes property of TreeBean.



## Instant Node Selection with Ajax

Another common requirement is to click on a tree node and display detailed data represented by that node instantly. This is quite easy to implement with tree. Following example displays selected node information in a dialog when node is clicked;

```
public class TreeBean {

    private TreeNode root;
    private TreeNode[] selectedNodes;

    public void onNodeSelect(NodeSelectEvent event) {
        selectedNode = event.getTreeNode();
    }

    //getters, setters and build of tree model
}
```

```

<h:form>
    <p:tree value="#{treeBean.model}"
        nodeSelectListener="#{treeBean.onNodeSelect}"
        selectionMode="single"
        selection="#{treeBean.selectedNodes}"
        update="detail"
        onselectStart="dlg.show"
        onselectComplete="dlg.hide()">

        <p:treeNode>
            <h:outputText value="#{node}" />
        </p:treeNode>
    </p:tree>

    <p:dialog header="Selected Node" widgetVar="dlg" width="250px">
        <h:outputText id="detail"
            value="#{treeBean.selectedNode.data}" />
    </p:dialog>

</h:form>

```

When a node is selected, tree makes an ajax request that executes the `nodeSelectListener`, after that the component defined with the `update` attribute is updated with the partial response. Optional `onselectStart` and `onselectComplete` attributes are handy to execute custom javascript.

## Selection Propagation

Selection propagation is controlled via two attributes named `propagateSelectionDown` and `propagateSelectionUp`. Both are false by default.

## Node Caching

When caching is turned on by default, dynamically loaded nodes will be kept in memory so re-expanding a node will not trigger a server side request. In case it's set to false, collapsing the node will remove the children and expanding it later causes the children nodes to be fetched from server again. Note that when caching is turned on collapse are not notified on the server side and expand events are executed only once.

## Animations

Expand and Collapse operations can be animated using `expandAnim` and `collapseAnim`. There're two valid values for these attributes, `FADE_IN` and `FADE_OUT`.

```

<p:tree value="#{treeBean.root}" var="node" dynamic="true"
    expandAnim="FADE_IN" collapseAnim="FADE_OUT" >
    <p:treeNode>
        <h:outputText value="#{node}" />
    </p:treeNode>
</p:tree>

```

## Handling Node Click

If you need to execute custom javascript when a treenode is clicked, use the *onNodeClick* attribute. Your javascript method will be processed with passing an object containing node information as a parameter.

```
<p:tree value="#{treeBean.root}" onNodeClick="handleNodeClick">
    ...
</p:tree>
```

```
function handleNodeClick(args) {
    alert("You clicked:" + args.node);
}
```

## Expand by default

If you need all nodes to be displayed as expanded on initial page load, set the expanded setting to true.

```
<p:tree value="#{treeBean.root}" expanded="true">
    ...
</p:tree>
```

## Skinning

Treeview has certain css selectors for nodes, for full list selectors visit;

<http://developer.yahoo.com/yui/treeview/#style>

## 3.95 TreeNode

TreeNode is used with Tree component to represent a node in tree.

### Info

Tag	<b>treeNode</b>
Component Class	<b>org.primefaces.component.tree.UITreeNode</b>
Component Type	<b>org.primefaces.component.UITreeNode</b>
Component Family	<b>org.primefaces.component</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
type	default	String	Type of the tree node
styleClass	null	String	Style class to apply a particular tree node type

### Getting started with the TreeNode

TreeNode is used with Tree component, refer to Tree section for more information.

## 3.96 TreeTable

Treetable is used for displaying hierarchical data in tabular format.

Name	Size	Type	Options
▼ Documents	-	Folder	
▶ Work	-	Folder	
▶ PrimeFaces	-	Folder	
▶ Pictures	-	Folder	
▼ Music	-	Folder	
▶ Turkish	-	Folder	
▶ Cem Karaca	-	Folder	
▶ Erkin Koray	-	Folder	
▶ Mogollar	-	Folder	

### Info

Tag	<b>treeTable</b>
Component Class	<b>org.primefaces.component.treetable.TreeTable</b>
Component Type	<b>org.primefaces.component.TreeTable</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.TreeTableRenderer</b>
Renderer Class	<b>org.primefaces.component.treetable.TreeTableRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	A TreeNode instance as the backing model.
var	null	String	Name of the request-scoped variable used to refer each treenode.
style	null	String	Inline style of the container element.

Name	Default	Type	Description
styleClass	null	String	Style class of the container element.
readOnly	FALSE	Boolean	Nodes are displayed as readonly and expanded.
expanded	FALSE	Boolean	Nodes are displayed as expanded.
widgetVar	null	String	Variable name of the client side widget.

## Getting started with the TreeTable

Similar to the Tree, TreeTable is populated with an *org.primefaces.model.TreeNode* instance that corresponds to the root node. TreeNode API has a hierarchical data structure and represents the data to be populated in tree. For an example, model to be displayed is a collection of documents.

```
public class Document {
    private String name;
    private String size;
    private String type;

    //getters, setters
}
```

```
<p:treeTable value="#{documentsController.root}" var="document">
    <p:column>
        <f:facet name="header">
            Name
        </f:facet>
        <h:outputText value="#{document.name}" />
    </p:column>

    <p:column>
        <f:facet name="header">
            Size
        </f:facet>
        <h:outputText value="#{document.size}" />
    </p:column>

    <p:column>
        <f:facet name="header">
            Type
        </f:facet>
        <h:outputText value="#{document.type}" />
    </p:column>
</p:treeTable>
```

```

public class DocumentsController {

    private TreeNode root;

    public DocumentsController() {
        root = new DefaultTreeNode("root", null);

        TreeNode documents = new DefaultTreeNode(new Document("Documents", "-", "Folder"), root);
        TreeNode pictures = new DefaultTreeNode(new Document("Pictures", "-", "Folder"), root);
        TreeNode music = new DefaultTreeNode(new Document("Music", "-", "Folder"), root);

        TreeNode work = new DefaultTreeNode(new Document("Work", "-", "Folder"), documents);
        TreeNode primefaces = new DefaultTreeNode(new Document("PrimeFaces", "-", "Folder"),
            documents);

        //Documents
        TreeNode expenses = new DefaultTreeNode("document", new Document("Expenses.doc", "30 KB",
            "Word Document"), work);
        TreeNode resume = new DefaultTreeNode("document", new Document("Resume.doc", "10 KB", "Word
Document"), work);
        TreeNode refdoc = new DefaultTreeNode("document", new Document("RefDoc.pages", "40 KB",
            "Pages Document"), primefaces);

        //Pictures
        TreeNode barca = new DefaultTreeNode("picture", new Document("barcelona.jpg", "30 KB", "JPEG
Image"), pictures);
        TreeNode primelogo = new DefaultTreeNode("picture", new Document("logo.jpg", "45 KB", "JPEG
Image"), pictures);
        TreeNode optimus = new DefaultTreeNode("picture", new Document("optimusprime.png", "96 KB",
            "PNG Image"), pictures);

        //Music
        TreeNode turkish = new DefaultTreeNode(new Document("Turkish", "-", "Folder"), music);

        TreeNode cemKaraca = new DefaultTreeNode(new Document("Cem Karaca", "-", "Folder"),
            turkish);
        TreeNode erkinKoray = new DefaultTreeNode(new Document("Erkin Koray", "-", "Folder"),
            turkish);
        TreeNode mogollar = new DefaultTreeNode(new Document("Mogollar", "-", "Folder"), turkish);

        TreeNode nemalacak = new DefaultTreeNode("mp3", new Document("Nem Alacak Felek Benim", "1500
KB", "Audio File"), cemKaraca);
        TreeNode resimdeki = new DefaultTreeNode("mp3", new Document("Resimdeki Gozyaslari", "2400
KB", "Audio File"), cemKaraca);

        TreeNode copculer = new DefaultTreeNode("mp3", new Document("Copculer", "2351 KB", "Audio
File"), erkinKoray);
        TreeNode oylebirgecer = new DefaultTreeNode("mp3", new Document("Oyle bir Gecer", "1794 KB",
            "Audio File"), erkinKoray);

        TreeNode toprakana = new DefaultTreeNode("mp3", new Document("Toprak Ana", "1536 KB", "Audio
File"), mogollar);
        TreeNode bisiyapmali = new DefaultTreeNode("mp3", new Document("Bisi Yapmali", "2730 KB",
            "Audio File"), mogollar);
    }

    public TreeNode getRoot() {
        return root;
    }
}

```

## Expanded by Default

If you'd like to render the treeTable as expanded by default, set expanded option to true.

```
<p:treeTable value="#{documentsController.root}" var="document"
    expanded="true">
    ...
</p:treeTable>
```

## ReadOnly by Default

When readOnly mode is enabled, toggle arrows are not rendered and treeTable is displayed as expanded.

```
<p:treeTable value="#{documentsController.root}" var="document"
    readOnly="true">
    ...
</p:treeTable>
```

Name	Size	Type
Documents	-	Folder
Work	-	Folder
Expenses.doc	30 KB	Word Document
Resume.doc	10 KB	Word Document
PrimeFaces	-	Folder
RefDoc.pages	40 KB	Pages Document
Pictures	-	Folder
barcelona.jpg	30 KB	JPEG Image
logo.jpg	45 KB	JPEG Image
optimusprime.png	96 KB	PNG Image
Music	-	Folder
Turkish	-	Folder
Cem Karaca	-	Folder
Nem Alacak Felek Benim	1500 KB	Audio File
Resimdeki Gozyaslar	2400 KB	Audio File
Erkin Koray	-	Folder
Copuler	2351 KB	Audio File
Oyle bir Gecer	1794 KB	Audio File
Mogollar	-	Folder
Toprak Ana	1536 KB	Audio File
Bisi Yapmali	2730 KB	Audio File

## Skinning

TreeTable content resides in a container element which style and styleClass attributes apply. Following is the list of structural style classes;

Class	Applies
.ui-treetable	Main container element (table)
.ui-treetable-header	Column header container
.ui-treetable-header-label	Column header label
.ui-treetable-data	Body element of the table containing data

As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;

Name	Size	Type	Options
▼ Documents	-	Folder	
▶ Work	-	Folder	
▶ PrimeFaces	-	Folder	
▶ Pictures	-	Folder	
▶ Music	-	Folder	

## 3.97 Watermark

Watermark displays a hint on an input field.

### Info

Tag	<b>watermark</b>
Component Class	<b>org.primefaces.component.watermark.Watermark</b>
Component Type	<b>org.primefaces.component.Watermark</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.WatermarkRenderer</b>
Renderer Class	<b>org.primefaces.component.watermark.WatermarkRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	0	Integer	Text of watermark.
for	null	String	Id of the component to attach the watermark
forElement	null	String	jQuery selector to attach the watermark

### Getting started with Watermark

Watermark requires a target of the input component, one way is to use for attribute.

```
<h:inputText id="txt" value="#{bean.searchKeyword}" />
<p:watermark for="txt" value="Search with a keyword" />
```

## Form Submissions

Watermark is set as the text of an input field which shouldn't be sent to the server when an enclosing form is submitted. This would result in updating bean properties with watermark values. Watermark component is clever enough to handle this case, by default in non-ajax form submissions, watermarks are cleared. However ajax submissions required a little manual effort.

```
<h:inputText id="txt" value="#{bean.searchKeyword}" />  
  
<p:watermark for="txt" value="Search with a keyword" />  
  
<h:commandButton value="Submit" />  
<p:commandButton value="Submit" onclick="PrimeFaces.cleanWatermarks()"  
oncomplete="PrimeFaces.showWatermarks()" />
```

## Skinning

There's only one css style class applying watermark which is '*.ui-watermark*', you can override this class to bring in your own style.

## 3.98 Wizard

Wizard provides an ajax enhanced UI to implement a workflow easily in a single page. Wizard consists of several child tab components where each tab represents a step in the process.

The screenshot shows a wizard component with four tabs: Personal, Address, Contact, and Confirmation. The Personal tab is active and displays a form titled "Personal Details". The form includes fields for Firstname, Lastname, and Age, each with a required asterisk. There is also a checkbox labeled "Skip to last". A "Next" button is located at the bottom right of the step.

### Info

Tag	<b>wizard</b>
Component Class	<b>org.primefaces.component.wizard.Wizard</b>
Component Type	<b>org.primefaces.component.Wizard</b>
Component Family	<b>org.primefaces.component</b>
Renderer Type	<b>org.primefaces.component.WizardRenderer</b>
Renderer Class	<b>org.primefaces.component.wizard.WizardRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
step	0	String	Id of the current step in flow
effect	TRUE	Boolean	Specifies whether animation should be used or not during transition.
effectSpeed	normal	String	Duration of effect.
style	null	String	Style of the main wizard container element.
styleClass	null	String	Style class of the main wizard container element.

Name	Default	Type	Description
flowListener	null	MethodExpr	Server side listener to invoke when wizard attempts to go forward or back.
showNavBar	TRUE	Boolean	Specifies visibility of default navigator arrows.
onback	null	String	Javascript event handler to be invoked when flow goes back.
onnext	null	String	Javascript event handler to be invoked when flow goes forward.
nextLabel	null	String	Label of next navigation button.
backLabel	null	String	Label of back navigation button.
showStepStatus	TRUE	Boolean	Specifies visibility of default step title bar.
widgetVar	null	String	Name of the client side widget

## Getting Started with Wizard

Each step in the flow is represented with a tab. As an example following wizard is used to create a new user in a total of 4 steps where last step is for confirmation of the information provided in first 3 steps. To begin with create your backing bean, it's important that the bean lives across multiple requests so avoid a request scope bean. Optimal scope for wizard is viewScope which is built-in with JSF 2.0.

```
public class UserWizard {

    private User user = new User();

    public User getUser() {
        return user;
    }

    public void setUser(User user) {
        this.user = user;
    }

    public void save(ActionEvent actionEvent) {
        //Persist user
        FacesMessage msg = new FacesMessage("Successful",
            "Welcome :" + user.getFirstname());
        FacesContext.getCurrentInstance().addMessage(null, msg);
    }
}
```

*User* is a simple pojo with properties such as firstname, lastname, email and etc. Following wizard requires 3 steps to get the user data; Personal Details, Address Details and Contact Details. Note that last tab contains read-only data for confirmation and the submit button.

```

<h:form>

    <p:wizard>
        <p:tab id="personal">
            <p:panel header="Personal Details">

                <h:messages errorClass="error"/>

                <h:panelGrid columns="2">
                    <h:outputText value="Firstname: *" />
                    <h:inputText value="#{userWizard.user.firstname}" required="true"/>

                    <h:outputText value="Lastname: *" />
                    <h:inputText value="#{userWizard.user.lastname}" required="true"/>

                    <h:outputText value="Age: " />
                    <h:inputText value="#{userWizard.user.age}" />
                </h:panelGrid>
            </p:panel>
        </p:tab>

        <p:tab id="address">
            <p:panel header="Address Details">

                <h:messages errorClass="error"/>

                <h:panelGrid columns="2" columnClasses="label, value">
                    <h:outputText value="Street: " />
                    <h:inputText value="#{userWizard.user.street}" />

                    <h:outputText value="Postal Code: " />
                    <h:inputText value="#{userWizard.user.postalCode}" />

                    <h:outputText value="City: " />
                    <h:inputText value="#{userWizard.user.city}" />
                </h:panelGrid>
            </p:panel>
        </p:tab>

        <p:tab id="contact">
            <p:panel header="Contact Information">

                <h:messages errorClass="error"/>

                <h:panelGrid columns="2">
                    <h:outputText value="Email: *" />
                    <h:inputText value="#{userWizard.user.email}" required="true"/>

                    <h:outputText value="Phone: " />
                    <h:inputText value="#{userWizard.user.phone}"/>

                    <h:outputText value="Additional Info: " />
                    <h:inputText value="#{userWizard.user.info}"/>
                </h:panelGrid>
            </p:panel>
        </p:tab>
    </p:wizard>

```

```

<p:tab id="confirm">
    <p:panel header="Confirmation">

        <h:panelGrid id="confirmation" columns="6">
            <h:outputText value="Firstname: " />
            <h:outputText value="#{userWizard.user.firstname}" />

            <h:outputText value="Lastname: " />
            <h:outputText value="#{userWizard.user.lastname}" />

            <h:outputText value="Age: " />
            <h:outputText value="#{userWizard.user.age}" />

            <h:outputText value="Street: " />
            <h:outputText value="#{userWizard.user.street}" />

            <h:outputText value="Postal Code: " />
            <h:outputText value="#{userWizard.user.postalCode}" />

            <h:outputText value="City: " />
            <h:outputText value="#{userWizard.user.city}" />

            <h:outputText value="Email: " />
            <h:outputText value="#{userWizard.user.email}" />

            <h:outputText value="Phone " />
            <h:outputText value="#{userWizard.user.phone}" />

            <h:outputText value="Info: " />
            <h:outputText value="#{userWizard.user.info}" />

            <h:outputText />
            <h:outputText />
        </h:panelGrid>

        <p:commandButton value="Submit" actionListener="#{userWizard.save}" />
    </p:panel>
</p:tab>

</p:wizard>
</h:form>

```

## AJAX and Partial Validations

Switching between steps is based on ajax, meaning each step is loaded dynamically with ajax. Partial validation is also built-in, by this way when you click next, only the current step is validated, if the current step is valid, next tab's contents are loaded with ajax. Validations are not executed when flow goes back.

## Navigations

Wizard provides two icons to interact with; next and prev. Please see the skinning wizard section to know more about how to change the look and feel of a wizard.

## Custom UI

By default wizard displays right and left arrows to navigate between steps, if you need to come up with your own UI, set *showNavBar* to false and use the provided the client side api.

```
<p:wizard showNavBar="false" widgetVar="wiz">
    ...
</p:wizard>

<h:outputLink value="#" onclick="wiz.next();">Next</h:outputLink>
<h:outputLink value="#" onclick="wiz.back();">Back</h:outputLink>
```

## Ajax FlowListener

If you'd like get notified on server side when wizard attempts to go back or forward, define a flowListener.

```
<p:wizard flowListener="#{userWizard.handleFlow}">
    ...
</p:wizard>
```

```
public String handleFlow(FlowEvent event) {
    String currentStepId = event.getCurrentStep();
    String stepToGo = event.getNextStep();

    if(skip)
        return "confirm";
    else
        return event.getNextStep();
}
```

Steps here are simply the ids of tab, by using a flowListener you can decide which step to display next so wizard does not need to be linear always.

## Client Side Callbacks

Wizard is equipped with onback and onnext attributes, in case you need to execute custom javascript after wizard goes back or forth. You just need to provide the names of javascript functions as the values of these attributes.

```
<p:wizard onnext="alert('Next')" onback="alert('Back')">
    ...
</p:wizard>
```

## Client Side API

Widget: *PrimeFaces.widget.Wizard*

Method	Params	Return Type	Description
next()	-	void	Proceeds to next step.
back()	-	void	Goes back in flow.
getStepIndex()	-	Number	Returns the index of current step.

## Skinning Wizard

Wizard reside in a div container element which *style* and *styleClass* attributes apply. Additionally a couple of css selectors exist for controlling the look and feel important parts of the wizard like the navigators. Following is the list.

Selector	Applies
.ui-wizard	Main container element
.ui-wizard-content	Container element of content
.ui-wizard-navbar	Container of navigation controls
.ui-wizard-nav-back	Back navigation control
.ui-wizard-nav-next	Forward navigation control

Here is an example based on a different theme.

The screenshot shows a PrimeFaces Wizard component with four tabs: Personal, Address, Contact, and Confirmation. The Personal tab is currently selected. Below it, there is a panel titled "Personal Details" containing input fields for Firstname, Lastname, and Age, along with a checkbox for "Skip to last". At the bottom right of the panel is a "Next" button.

# 4. TouchFaces

TouchFaces is at proof of concept state as of 2.2 and will be production ready in PrimeFaces 3.0. It will be rebranded as PrimeFaces Mobile and powered by jQuery Mobile.

TouchFaces is the UI kit for developing mobile web applications with JSF. It mainly targets devices with webkit browsers such as iPhone, all Android phones, Palm, Nokia S60 and etc. TouchFaces is included in PrimeFaces and no additional configuration is required other than the touchfaces taglib. TouchFaces is built on top of the jqTouch jquery plugin.

## 4.1 Getting Started with TouchFaces

There're a couple of special components belonging to the touchfaces namespace. Lets first create an example JSF page called touch.xhtml with the touchfaces namespace.

```
<f:view xmlns="http://www.w3.org/1999/xhtml"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:i="http://primefaces.prime.com.tr/touch">

</f:view>
```

Next step is defining the *<i:application />* component.

```
<f:view xmlns="http://www.w3.org/1999/xhtml"
        xmlns:f="http://java.sun.com/jsf/core"
        xmlns:i="http://primefaces.prime.com.tr/touch">

    <i:application>
    </i:application>
</f:view>
```

### Themes

TouchFaces ships with two built-in themes, default and dark. Themes can be customized using the theme attribute of the application. "Notes" sample app using the dark theme whereas other apps have the default iphone theme.

```
<i:application theme="dark">
    //content
</i:application>
```

## Application Icon

iPhone has a nice feature allowing users to add web apps to their home screen so that later they can launch these apps just like a native iphone app. To assign an icon to your TouchFaces app use the icon attribute of the application component. It's important to use an icon of size 57x57 to get the best results.

```
<i:application icon="translate.png">  
    //content  
</i:application>
```

Here's an example demonstrating how it looks when you add your touchfaces app to your home screen.



That's it, you now have the base for your mobile web application. Next thing is building the UI with views.

## 4.2 Views

TouchFaces models each screen in a application as "views" and a view is created with the `<i:view />` component. Each view must have an id and an optional title. You can have as many views as you want inside an application. To set a view as the home view use a convention and set the id of the view as "home".

```
<f:view xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:i="http://primefaces.prime.com.tr/touch">

    <i:application>
        <i:view id="home" title="Home Page">
            //Home view content
        </i:view>
    </i:application>

</f:view>
```

When you run this page, only the home view would be displayed, a view can be built with core JSF and components and TouchFaces specific components like tableView, rowGroups, rowItems and more.

### TableViews

TableView is a useful control in iPhone sdk and touchfaces includes a tableview as well to provide a similar feature. TableView consists of rowGroups and rowItems. Here's a sample tableView;

```
<f:view xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:i="http://primefaces.prime.com.tr/touch">

    <i:application>
        <i:view id="home" title="Home Page">
            <i:tableView>
                <i:rowGroup title="Group Title">
                    <i:rowItem value="Row 1"/>
                    <i:rowItem value="Row 2"/>
                </i:rowGroup>
            </i:tableView>
        </i:view>
    </i:application>
</f:view>
```



## Group Display Modes

A rowgroup can be displayed in a couple of different ways default way is ‘rounded’ which is used in previous example. Full list of possible values are;

- rounded
- edgetoedge
- plastic
- metal

Following list uses edgetoedge display mode;

```
<i:tableView>
    <i:rowGroup title="Group Title" display="edgetoedge">
        <i:rowItem value="Row 1"/>
        <i:rowItem value="Row 2"/>
    </i:rowGroup>
</i:tableView>
```



## 4.3 Navigations

TouchFaces navigations are based on conventions and some components has the ability to trigger a navigation. An example is rowItem, using the view attribute you can specify which view to display when the rowItem is clicked. Also TouchFaces client side api provides useful navigation utilities.

```
<i:view>
    <i:tableView display="regular">
        <i:rowGroup title="Group Title">
            <i:rowItem value="Other View" view="otherview"/>
        </i:rowGroup>
    </i:tableView>
</i:view>

<i:view id="otherview" title="Other view">
    //Other view content
</i:view>
```

### NavBarControl

You can also place navBarControls at the navigation bar for use cases such as navigation back and displaying another view. NavBarControl's are used as facets, following control is placed at the left top corner and used to go back to a previous view.

```
<i:view id="otherview" title="Other view">
    <f:facet name="leftNavBar">
        <i:navBarControl label="Home" view="home" />
    </f:facet>
    //view content
</i:view>
```



Similarly a navBarControl to place the right side of the navigation bar use *rightNavBar* facet.

### Navigation Effects

View transition animation is defined by *effect* option;

```
<f:facet name="leftNavBar">
    <i:navBarControl label="Settings" view="settings" effect="flip"/>
</f:facet>
```

Default animation used when navigation to a view is "slide".

- slide
- slideup
- flip
- dissolve
- fade
- flip
- pop
- swap
- cube

## TouchFaces Navigation API

TouchFaces client side object provides two useful navigation methods;

- goTo(viewName, animation)
- goBack()

Example below demonstrates how to execute a java method with p:commandLink and go to another view after ajax request is completed.

```
<p:commandLink actionListener="#{bean.value}" update="comp"
    oncomplete="TouchFaces.goTo('otherview', 'flip')"/>
```

## 4.4 Ajax Integration

TouchFaces is powered by PrimeFaces PPR infrastructure, this allows loading views with ajax, do ajax form submissions and other ajax use cases. Also rowItem component has built-in support for ajax and can easily load other views dynamically with ajax before displaying them. An example would be;

```
<i:view>
    <i:tableView display="regular">
        <i:rowGroup title="Group Title">
            <i:rowItem value="Other View" view="otherview"
                       actionListener="#{bean.action}" update="table"/>
        </i:rowGroup>
    </i:tableView>
</i:view>

<i:view id="otherview" title="Other view">
    <i:tableView id="table" display="regular">
        <i:tableView>
    </i:tableView>
</i:view>
```

## 4.5 Sample Applications

There're various sample applications developed with TouchFaces, these apps are also deployed online so you can check them with your mobile device (preferably iphone, ipod touch or an android phone). Source codes are also available in PrimeFaces svn repository.

We strongly recommend using these apps as references since each of them use a different feature of TouchFaces.



## 4.6 TouchFaces Components

This section includes detailed tag information of TouchFaces Components.

### 4.6.1 Application

#### Info

Tag	<b>application</b>
Component Class	<b>org.primefaces.touch.component.applicaiton.Application</b>
Component Type	<b>org.primefaces.touch.Application</b>
Component Family	<b>org.primefaces.touch</b>
Renderer Type	<b>org.primefaces.touch.component.ApplicationRenderer</b>
Renderer Class	<b>org.primefaces.touch.component.application.ApplicationRenderer</b>

#### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
theme	null	String	Theme of the app, "default" or "dark".
icon	null	String	Icon of the app.

## 4.6.2 NavBarControl

### Info

Tag	<b>navBarControl</b>
Component Class	<b>org.primefaces.touch.component.navbarcontrol.NavBarControl</b>
Component Type	<b>org.primefaces.touch.NavBarControl</b>
Component Family	<b>org.primefaces.touch</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
label	null	String	Label of the item.
view	null	String	Id of the view to be displayed.
type	back	String	Type of the display, "back" or "button".
effect	null	String	Effect to be used when displaying the view navigated to.

## 4.6.3 RowGroup

### Info

Tag	<b>rowGroup</b>
Component Class	<b>org.primefaces.touch.component.rowgroup.RowGroup</b>
Component Type	<b>org.primefaces.touch.RowGroup</b>
Component Family	<b>org.primefaces.touch</b>
Renderer Type	<b>org.primefaces.touch.component.RowGroupRenderer</b>
Renderer Class	<b>org.primefaces.touch.component.rowgroup.RowGroupRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
title	null	String	Optional title of the row group.

## 4.6.4 RowItem

### Info

Tag	<b>rowItem</b>
Component Class	<b>org.primefaces.touch.component.rowitem.RowItem</b>
Component Type	<b>org.primefaces.touch.RowItem</b>
Component Family	<b>org.primefaces.touch</b>
Renderer Type	<b>org.primefaces.touch.component.RowItemRenderer</b>
Renderer Class	<b>org.primefaces.touch.component.rowitem.RowItemRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
view	null	String	Id of the view to be displayed.
url	null	String	Optional external url link.
update	null	String	Client side of the component(s) to be updated after the partial request.
value	null	String	Label of the item.
action	null	MethodExpr	A method expression that'd be processed in the partial request caused by uiajax.
actionListener	null	MethodExpr	An actionlistener that'd be processed in the partial request caused by uiajax.
immediate	FALSE	Boolean	Boolean value that determines the phaseId, when true actions are processed at apply_request_values, when false at invoke_application phase.

## 4.6.5 Switch

### Info

Tag	<b>switch</b>
Component Class	<b>org.primefaces.touch.component.switch.Switch</b>
Component Type	<b>org.primefaces.touch.Switch</b>
Component Family	<b>org.primefaces.touch</b>
Renderer Type	<b>org.primefaces.touch.component.SwitchRenderer</b>
Renderer Class	<b>org.primefaces.touch.component.switch.SwitchRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	Object	Value of the component than can be either an EL expression of a literal text
converter	null	Converter/ String	An el expression or a literal text that defines a converter for the component. When it's an EL expression, it's resolved to a converter instance. In case it's a static text, it must refer to a converter id
immediate	FALSE	Boolean	Boolean value that specifies the lifecycle phase the valueChangeEvents should be processed, when true the events will be fired at "apply request values", if immediate is set to false, valueChange Events are fired in "process validations" phase
required	FALSE	Boolean	Marks component as required
validator	null	MethodExpr	A method binding expression that refers to a method validationg the input
valueChangeListener	null	MethodExpr	A method binding expression that refers to a method for handling a valuchangeevent
requiredMessage	null	String	Message to be displayed when required field validation fails.

Name	Default	Type	Description
converterMessage	null	String	Message to be displayed when conversion fails.
validatorMessage	null	String	Message to be displayed when validation fields.

## 4.6.6 TableView

### Info

Tag	<b>tableView</b>
Component Class	<b>org.primefaces.touch.component.tableview.TableView</b>
Component Type	<b>org.primefaces.touch.TableView</b>
Component Family	<b>org.primefaces.touch</b>
Renderer Type	<b>org.primefaces.touch.component.TableViewRenderer</b>
Renderer Class	<b>org.primefaces.touch.component.tableview.TableViewRenderer</b>

### Attributes

Name	Default	Type	Description
id	null	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean

## 4.6.7 View

### Info

Tag	<code>view</code>
Component Class	<code>org.primefaces.touch.component.view.View</code>
Component Type	<code>org.primefaces.touch.View</code>
Component Family	<code>org.primefaces.touch</code>
Renderer Type	<code>org.primefaces.touch.component.ViewRenderer</code>
Renderer Class	<code>org.primefaces.touch.component.viewrenderer.ViewRenderer</code>

### Attributes

Name	Default	Type	Description
<code>id</code>	null	String	Unique identifier of the component.
<code>rendered</code>	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
<code>binding</code>	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
<code>title</code>	regular	String	Optional title of the view.

# 5. Partial Rendering and Processing

PrimeFaces provides a partial rendering and view processing feature based on standard JSF 2 APIs to enable choosing what to process in JSF lifecycle and what to render in the end with ajax.

## 5.1 Partial Rendering

In addition to components like autoComplete, datatable, slider with built-in ajax capabilities, PrimeFaces also provides a generic PPR (Partial Page Rendering) mechanism to update JSF components with ajax. Several components are equipped with the common PPR attributes (e.g. update, process, onstart, oncomplete).

### 5.1.1 Infrastructure

PrimeFaces Ajax Framework is based on standard server side APIs of JSF 2. There are no additional artifacts like custom AjaxViewRoot, AjaxStateManager, AjaxViewHandler, Servlet Filters, HtmlParsers, PhaseListeners and so on. PrimeFaces aims to keep it clean, fast and lightweight.

On client side rather than using client side API implementations of JSF implementations like Mojarra and MyFaces, PrimeFaces scripts are based on the most popular javascript library; jQuery which is far more tested, stable regarding ajax, dom handling, dom tree traversing than a JSF implementations scripts.

### 5.1.2 Using IDs

#### Getting Started

When using PPR you need to specify which component(s) to update with ajax. If the component that triggers PPR request is at the same namingcontainer (eg. form) with the component(s) it renders, you can use the server ids directly. In this section although we'll be using commandButton, same applies to every component that's capable of PPR such as commandLink, poll, remoteCommand and etc.

```
<h:form>
    <p:commandButton update="display" />
    <h:outputText id="display" value="#{bean.value}" />
</h:form>
```

#### PrependId

Setting prependId setting of a form has no effect on how PPR is used.

```
<h:form prependId="false">
    <p:commandButton update="display" />
    <h:outputText id="display" value="#{bean.value}" />
</h:form>
```

## ClientId

It is also possible to define the client id of the component to update.

```
<h:form id="myform">
    <p:commandButton update="myform:display" />
    <h:outputText id="display" value="#{bean.value}" />
</h:form>
```

## Different NamingContainers

If your page has different naming containers (e.g. two forms), you also need to add the container id to search expression so that PPR can handle requests that are triggered inside a namingcontainer that updates another namingcontainer. Following is the suggested way using separator char as a prefix, note that this uses same search algorithm as standard JSF 2 implementation;

```
<h:form id="form1">
    <p:commandButton update=":form2:display" />
</h:form>

<h:form id="form2">
    <h:outputText id="display" value="#{bean.value}" />
</h:form>
```

Using absolute clientIds will also work as a PrimeFaces extension however we might remove it in a future release to align with JSF spec.

```
<h:form id="form1">
    <p:commandButton update="form2:display" />
</h:form>
<h:form id="form2">
    <h:outputText id="display" value="#{bean.value}" />
</h:form>
```

## Multiple Components

Multiple Components to update can be specified with providing a list of ids separated by a comma, whitespace or even both.

### *Comma*

```
<h:form>
    <p:commandButton update="display1,display2" />
    <h:outputText id="display1" value="#{bean.value1}" />
    <h:outputText id="display2" value="#{bean.value2}" />
</h:form>
```

### *WhiteSpace*

```
<h:form>
    <p:commandButton update="display1 display2" />
    <h:outputText id="display1" value="#{bean.value1}" />
    <h:outputText id="display2" value="#{bean.value2}" />
</h:form>
```

## Keywords

There are a couple of reserved keywords which serve as helpers.

Keyword	Description
@this	Component that triggers the PPR is updated
@parent	Parent of the PPR trigger is updated.
@form	Encapsulating form of the PPR trigger is updated
@none	PPR does not change the DOM with ajax response.

Example below updates the whole form.

```
<h:form>
    <p:commandButton update="@form" />
    <h:outputText value="#{bean.value}" />
</h:form>
```

Keywords can also be used together with explicit ids, so update="@form, display" is also supported.

### 5.1.3 Notifying Users

ajaxStatus is the component to notify the users about the status of **global** ajax requests. See the ajaxStatus section to get more information about the component.

#### Global vs Non-Global

By default ajax requests are global, meaning if there is an ajaxStatus component present on page, it is triggered.

If you want to do a "silent" request not to trigger ajaxStatus instead, set global to false. An example with commandButton would be;

```
<p:commandButton value="Silent" global="false" />  
<p:commandButton value="Notify" global="true" />
```

### 5.1.4 Bits&Pieces

#### PrimeFaces Ajax Javascript API

See the javascript section 8.3 to learn more about the PrimeFaces Javascript Ajax API.

## 5.2 Partial Processing

In Partial Page Rendering, only specified components are rendered, similarly in Partial Processing only defined components are processed. Processing means executing Apply Request Values, Process Validations, Update Model and Invoke Application JSF lifecycle phases only on defined components.

This feature is a simple but powerful enough to do group validations, avoiding validating unwanted components, eliminating need of using immediate and many more use cases. Various components such as commandButton, commandLink are equipped with process attribute, in examples we'll be using commandButton.

### 5.2.1 Partial Validation

A common use case of partial process is doing partial validations, suppose you have a simple contact form with two dropdown components for selecting city and suburb, also there's an inputText which is required. When city is selected, related suburbs of the selected city is populated in suburb dropdown.

```
<h:form>
    <h:selectOneMenu id="cities" value="#{bean.city}">
        <f:selectItems value="#{bean.cityChoices}" />
        <p:ajax listener="#{bean.populateSuburbs}" update="suburbs"
               process="@all"/>
    </h:selectOneMenu>

    <h:selectOneMenu id="suburbs" value="#{bean.suburb}">
        <f:selectItems value="#{bean.suburbChoices}" />
    </h:selectOneMenu>

    <h:inputText value="#{bean.email}" required="true"/>
</h:form>
```

When the city dropdown is changed an ajax request is sent to execute populateSuburbs method which populates suburbChoices and finally update the suburbs dropdown. Problem is populateSuburbs method will not be executed as lifecycle will stop after process validations phase to jump render response as email input is not provided. Reason is p:ajax has @all as the value stating to process every component on page but there is no need to process the inputText.

The solution is to define what to process in p:ajax. As we're just making a city change request, only processing that should happen is cities dropdown.

```

<h:form>
    <h:selectOneMenu id="cities" value="#{bean.city}">
        <f:selectItems value="#{bean.cityChoices}" />
        <p:ajax actionListener="#{bean.populateSuburbs}"
            event="change" update="suburbs" process="@this"/>
    </h:selectOneMenu>

    <h:selectOneMenu id="suburbs" value="#{bean.suburb}">
        <f:selectItems value="#{bean.suburbChoices}" />
    </h:selectOneMenu>

    <h:inputText value="#{bean.email}" required="true"/>
</h:form>

```

That is it, now `populateSuburbs` method will be called and suburbs list will be populated. Note that default value for `process` option is `@this` already for `p:ajax` as stated in AjaxBehavior documentation, it is explicitly defined here to give a better understanding of how partial processing works.

## 5.2.2 Keywords

Just like PPR, Partial processing also supports keywords.

Keyword	Description
<code>@this</code>	Component that triggers the PPR is processed.
<code>@parent</code>	Parent of the PPR trigger is processed.
<code>@form</code>	Encapsulating form of the PPR trigger is processed
<code>@none</code>	No component is processed, useful to revert changes to form.
<code>@all</code>	Whole component tree is processed just like a regular request.

Important point to note is, when a component is specified to process partially, children of this component is processed as well. So for example if you specify a panel, all children of that panel would be processed in addition to the panel itself.

```

<p:commandButton process="panel" />

<p:panel id="panel">
    //Children
</p:panel>

```

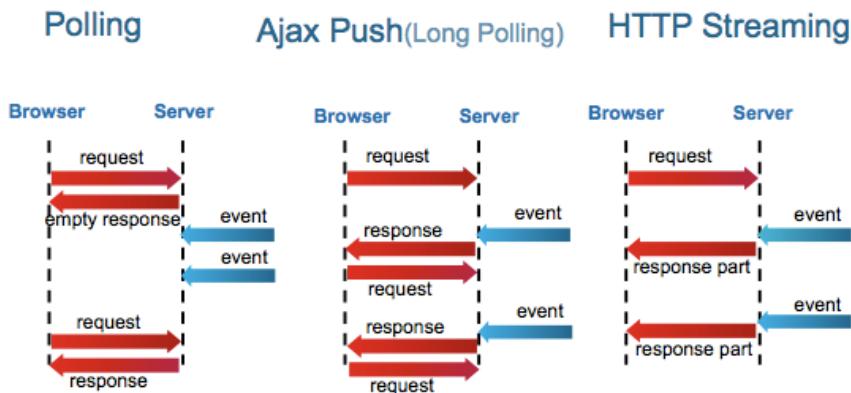
## 5.2.3 Using Ids

Partial Process uses the same technique applied in PPR to specify component identifiers to process. See section 5.1.2 for more information about how to define ids in process specification using commas and whitespaces.

# 6. Ajax Push/Comet

Prime Push is at proof of concept state as of 2.2 and will be production ready in PrimeFaces 3.0.

Comet is a model allowing a web server to push data to the browsers. Auctions and chat are well known example use cases of comet technique. Comet can be implemented in a couple of ways. Following is a schema describing these techniques.



**Polling:** Regular polling is not real comet, basically browser sends request to server based on a specific interval. This approach has nothing to do with comet and just provided for comparison.

**Long-Polling:** Browsers requests are suspended and only resumed when server decides to push data, after the response is retrieved browsers connects and begins to waiting for data again.

**Http Streaming:** With this approach, response is never committed and client always stays connected, push data is streamed to the client to process.

Current version of PrimeFaces is based on http-streaming, long-polling support will be added very soon in upcoming releases. PrimeFaces Push is built-on top of Atmosphere Framework. Next section describes atmosphere briefly.

## 6.1 Atmosphere

Atmosphere is a comet framework that can run on any application server supporting servlet 2.3+. Each container provides their own proprietary solution (Tomcat's CometProcessor, JBoss's HttpEvent, Glassfish Grizzly etc), Servlet 3.0 aims to unify these apis with a standard javax.servlet.AsyncListener.

Atmosphere does all the hard work, deal with container differences, browser compatibility, broadcasting of events and many more. See atmosphere home page for more information.

<<http://atmosphere.dev.java.net>

## 6.2 PrimeFaces Push

PrimeFaces aims to simplify developing comet applications with JSF, an example for this would be the PrimeFaces chat sample app that can easily be created with a couple of lines.

### 6.2.1 Setup

#### Comet Servlet

First thing to do is to configure the PrimeFaces Comet Servlet. This servlet handles the JSF integration and Atmosphere.

```
<servlet>
    <servlet-name>Comet Servlet</servlet-name>
    <servlet-class>org.primefaces.comet.PrimeFacesCometServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>Comet Servlet</servlet-name>
    <url-pattern>/primefaces_comet/*</url-pattern>
</servlet-mapping>
```

#### Atmosphere Libraries

PrimeFaces needs at least version of 0.5.1, you can download atmosphere from atmosphere homepage, you'll also need the atmosphere-compat-\* libraries. You can find these libraries at;

<http://download.java.net/maven/2/org/atmosphere/>

#### context.xml

If you're running tomcat, you'll also need a context.xml under META-INF.

```
<?xml version="1.0" encoding="UTF-8"?>
<Context>
    <Loader delegate="true"/>
</Context>
```

### 6.2.2. CometContext

Main element of PrimeFaces Push on server side is the *org.primefaces.comet.CometContext* which has a simple api to push data to browsers.

```
/**
 * @param channel Unique name of communication channel
 * @param data Information to be pushed to the subscribers as json
 */
CometContext.publish(String channel, Object data);
```

### 6.2.3 Push Component

<p:push /> is a PrimeFaces component that handles the connection between the server and the browser, it has two attributes you need to define.

```
<p:push channel="chat" onpublish="handlePublish"/>
```

channel: Name of the channel to connect and listen.

onpublish: Javascript event handler to be called when server sends data.

### 6.2.4 Putting it together: A Chat application

In this section, we'll develop a simple chat application with PrimeFaces, let's begin with the backing bean.

```
public class ChatController implements Serializable {

    private String message;
    private String username;
    private boolean loggedIn;

    public void send(ActionEvent event) {
        CometContext.publish("chat", username + ": " + message);
        message = null;
    }

    public void login(ActionEvent event) {
        FacesContext.getCurrentInstance().addMessage(null, new FacesMessage
("You're logged in!"));
        loggedIn = true;
        CometContext.publish("chat", username + " has logged in.");
    }

    //getters&setters
}
```

And the chat.xhtml;

```

...
<head>
    <script type="text/javascript">
        function handlePublish(response) {
            $('#display').append(response.data + '<br />');
        }
    </script>
</head>

<body>

<p:outputPanel id="display" />

<h:form prependId="false">

    <p:growl id="growl" />

    <p:panel header="Sign in" rendered="#{!chatController.loggedIn}">
        <h:panelGrid columns="3" >
            <h:outputText value="Username:" />
            <h:inputText value="#{chatController.username}" />
            <p:commandButton value="Login"
                actionListener="#{chatController.login}"
                oncomplete="$('display').slideDown()"/>
        </h:panelGrid>
    </p:panel>

    <p:panel header="Signed in as : #{chatController.username}"
        rendered="#{chatController.loggedIn}" toggleable="true">
        <h:panelGrid columns="3">
            <h:outputText value="Message:" />
            <h:inputText id="txt" value="#{chatController.message}" />
            <p:commandButton value="Send"
                actionListener="#{chatController.send}"
                oncomplete="$('txt').val('');"/>
        </h:panelGrid>
    </p:panel>
</h:form>

<p:push channel="chat" onpublish="handlePublish" />

</body>
...

```

Published object is serialized as JSON, passed to publish handlers and is accessible using *response.data*.

# 7. Javascript API

PrimeFaces renders unobtrusive javascript which cleanly separates behavior from the html. Client side engine is powered by jQuery version 1.4.4 which is the latest at the time of the writing.

## 7.1 PrimeFaces Namespace

*PrimeFaces* is the main javascript object providing utilities and namespace.

Method	Description
escapeClientId(id)	Escaped JSF ids with semi colon to work with jQuery selectors
addSubmitParam(el, name, param)	Adds hidden request parameters dynamically to the element.
cleanWatermarks()	Watermark component extension, cleans all watermarks on page before submitting the form.
showWatermarks()	Shows watermarks on form.

To be compatible with other javascript entities on a page, PrimeFaces defines two javascript namespaces;

### **PrimeFaces.widget.\***

Contains custom PrimeFaces widgets like;

- PrimeFaces.widget.DataTable
- PrimeFaces.widget.Tree
- PrimeFaces.widget.Poll
- and more...

Most of the components have a corresponding client side widget with same name.

### **PrimeFaces.ajax.\***

PrimeFaces.ajax namespace contains the ajax API which is described in next section.

## 7.2 Ajax API

PrimeFaces Ajax Javascript API is powered by jQuery and optimized for JSF. Whole API consists of three properly namespaced simple javascript functions.

### PrimeFaces.ajax.AjaxRequest

Sends ajax requests that execute JSF lifecycle and retrieve partial output. Function signature is as follows;

```
PrimeFaces.ajax.AjaxRequest(url, config, parameters);
```

*url*: URL to send the request.

*config*: Configuration options.

*params*: Parameters to send.

### Configuration Options

Option	Description
formId	Id of the form element to serialize.
async	Flag to define whether request should go in ajax queue or not, default is false.
global	Flag to define if p:ajaxStatus should be triggered or not, default is true.
update	Component(s) to update with ajax.
process	Component(s) to process in partial request.
source	(Required) Client id of the source component causing the request.
onstart()	Javascript callback to process before sending the ajax request, return false to cancel the request.
onsuccess(data, status, xhr, args)	Javascript callback to process when ajax request returns with success code. Takes four arguments, xml response, status code, xmlhttprequest and optional arguments provided by RequestContext API.
onerror(xhr, status, exception)	Javascript callback to process when ajax request fails. Takes three arguments, xmlhttprequest, status string and exception thrown if any.
oncomplete(xhr, status, args)	Javascript callback to process when ajax request completes. Takes three arguments, xmlhttprequest, status string and optional arguments provided by RequestContext API.

- oncomplete is triggered after onsuccess.

## Parameters

You can send any number of parameters as the third argument of request function as a javascript object.

## Examples

Suppose you have a JSF page called *createUser* with a simple form and some input components.

```
<h:form id="userForm">
    <h:inputText id="username" value="#{userBean.user.name}" />
    ... More components
</h:form>
```

You can post all the information in form with ajax using;

```
PrimeFaces.ajax.AjaxRequest('/myapp/createUser.jsf',
{
    formId: 'userForm',
    source: 'userForm',
    process: 'userForm'
});
```

More complex example with additional options;

```
PrimeFaces.ajax.AjaxRequest('/myapp/createUser.jsf',
{
    formId: 'userForm',
    source: 'userForm',
    process: 'userForm',
    update: 'msgs',
    oncomplete:function(xhr, status) {alert('Done');}
},
{
    'param_name1': 'value1',
    'param_name2': 'value2'
});
});
```

We recommend using p:remoteComponent instead of low level javascript api as it generates the same with much less effort and less possibility to do an error.

## PrimeFaces.ajax.AjaxResponse

PrimeFaces.ajax.AjaxResponse updates the specified components if any and synchronizes the client side JSF state. DOM updates are implemented using jQuery which uses a very fast algorithm.

## PrimeFaces.ajax.AjaxUtils

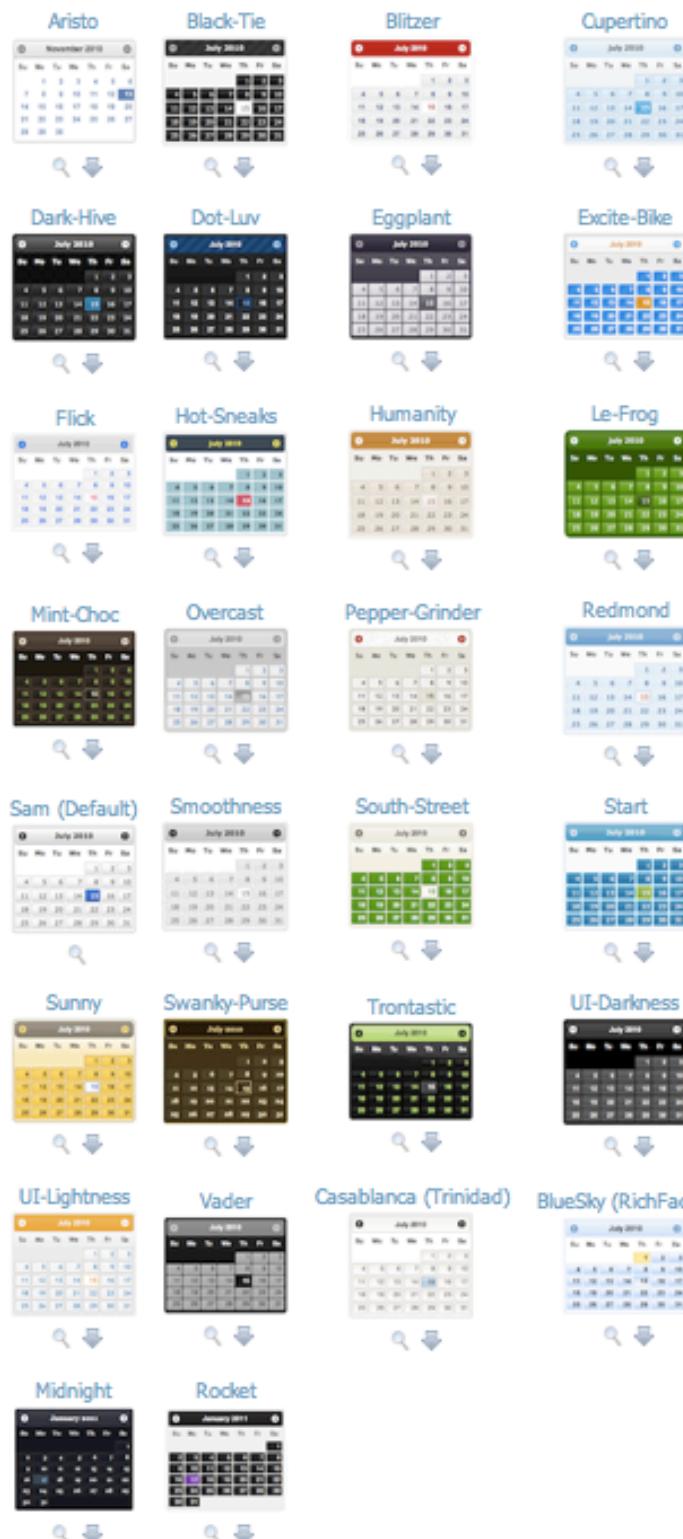
AjaxUtils contains useful utilities like encoding client side JSF viewstate, serializing a javascript object literal to a request query string and more.

Method	Description
encodeViewState	Encodes value held by javax.faces.ViewState hidden parameter.
updateState	Syncs serverside state with client state.
serialize(literal)	Serializes a javascript object literal like {name:'R10', number:10} to "name=R10&number=10"

# 8. Themes

PrimeFaces is integrated with powerful ThemeRoller CSS Framework. Currently there are 30 pre-designed themes that you can preview and download from PrimeFaces theme gallery.

<http://www.primefaces.org/themes.html>



## 8.1 Applying a Theme

Applying a theme to your PrimeFaces project is very easy. Each theme is packaged as a jar file, download the theme you want to use, add it to the classpath of your application and then define primefaces.THEME context parameter at your deployment descriptor (web.xml) with the theme name as the value.

### Download

Each theme is available for manual download at PrimeFaces Theme Gallery. If you are a maven user, define theme artifact as;

```
<dependency>
    <groupId>org.primefaces.themes</groupId>
    <artifactId>aristo</artifactId>
    <version>1.0.0</version>
</dependency>
```

artifactId is the name of the theme as defined at Theme Gallery page.

### Configure

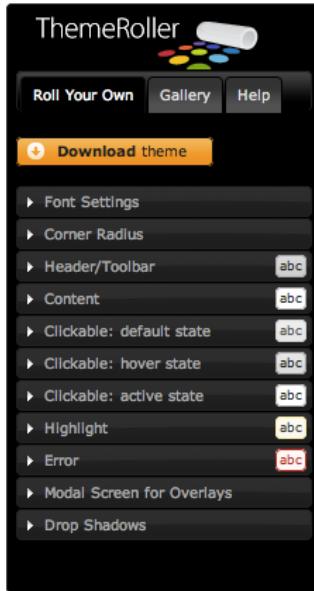
Once you've downloaded the theme, configure PrimeFaces to use it.

```
<context-param>
    <param-name>primefaces.THEME</param-name>
    <param-value>aristo</param-value>
</context-param>
```

That's it, you don't need to manually add any css to your pages or anything else, PrimeFaces will handle everything for you.

## 8.2 Creating a New Theme

If you'd like to create your own theme instead of using the pre-defined ones, that is easy as well because ThemeRoller provides a powerful and easy to use online visual tool.



Applying your own custom theme is a bit different than applying pre-built themes of Theme Gallery. There are two ways, one is manual installation and other is using PrimeFaces Theme API.

### Manual Installation

The theme package you've downloaded from ThemeRoller will have a css file and images folder, one way is to extract the contents to a folder in your application like %webroot%/themes and add the css file of the theme to your pages. Best place to add the css file is your page template so you only add it once. Suppose you've extracted the contents of the theme to your %webroot%/themes/mytheme folder. The name of the css would be something like *jquery-ui-{version}.custom.css*, for simplicity rename it to *theme.css* and then add it via link tag (*h:outputStylesheet* is not supported).

```
<link type="text/css" rel="stylesheet"
      href="{CONTEXT_PATH}/themes/mytheme/theme.css">
```

Next step is to configure PrimeFaces not to add its default sam skin to avoid a page having to themes applied. Set *primefaces.THEME* parameter to none to do this.

```
<context-param>
    <param-name>primefaces.THEME</param-name>
    <param-value>none</param-value>
</context-param>
```

We've created a short video tutorial about this available at <http://vimeo.com/14235640>.

## PrimeFaces Theme API

Theme API is the integrated way of applying your custom themes to your project, this approach requires you to create a jar file and add it to the classpath of your application. jar file must have the following folder structure.

```
- jar
  - META-INF
    - resources
      - primefaces-yourtheme
        - theme.css
        - images
```

Image references in your theme.css must also be converted to an expression that JSF resource loading can understand, example would be;

```
url("images/ui-bg_highlight-hard_100_f9f9f9_1x100.png")
```

should be;

```
url("#{resource['primefaces-yourtheme:images/ui-bg_highlight-hard_100_f9f9f9_1x100.png']}")
```

Then you can use your theme just like a pre-defined PrimeFaces Theme.

```
<context-param>
  <param-name>primefaces.THEME</param-name>
  <param-value>yourtheme</param-value>
</context-param>
```

We appreciate if you share your theme with PrimeFaces Community, contact us via [contact@prime.com.tr](mailto:contact@prime.com.tr).

## 8.3 How Themes Work

Powered by ThemeRoller, PrimeFaces separates structural css from skinning css.

### *Structural CSS*

These style classes define the skeleton of the components and include css properties such as margin, padding, display type, dimensions and positioning.

### *Skinning CSS*

Skinning defines the look and feel properties like colors, border colors, background images.

### Skinning Selectors

ThemeRoller features a couple of skinning selectors, most important of these are;

Selector	Applies
.ui-widget	All PrimeFaces components
.ui-widget-header	Header section of a component
.ui-widget-content	Content section of a component
.ui-state-default	Default class of a clickable
.ui-state-hover	Hover class of a clickable
.ui-state-active	When a clickable is selected
.ui-state-disabled	Disabled elements.
.ui-state-highlight	Highlighted elements.
.ui-icon	An element to represent an icon.

These classes are not aware of structural css like margins and paddings, mostly they only define colors. This clean separation brings great flexibility in theming because you don't need to know each and every skinning selectors of components to change their style.

For example Panel component's header section has the *.ui-panel-titlebar* structural class, to change the color of a panel header you don't need to about this class as *.ui-widget-header* also that defines the panel colors also applies to the panel header.

## 8.4 Theming Tips

- Default font size of themes might be bigger than expected, to change the font-size of PrimeFaces components globally, use the .ui-widget style class. An example of smaller fonts;

```
.ui-widget, .ui-widget .ui-widget {  
    font-size: 90% !important;  
}
```

- To create a themeable application, keep the user's selected theme in a session scoped bean and use EL to change it dynamically.

```
<context-param>  
    <param-name>primefaces.THEME</param-name>  
    <param-value>#{userPreferences.theme}</param-value>  
</context-param>
```

- When creating your own theme with themeroller tool, select one of the pre-designed themes that is close to the color scheme you want and customize that to save time.
- If you are using Apache Trinidad or JBoss RichFaces, PrimeFaces Theme Gallery includes Trinidad's Casablanca and RichFaces's BlueSky theme. You can use these themes to make PrimeFaces look like Trinidad or RichFaces components during migration.

# 9. Utilities

## 9.1 RequestContext

RequestContext is a simple utility that provides useful goodies such as adding parameters to ajax callback functions.

RequestContext can be obtained similarly to FacesContext.

```
RequestContext requestContext = RequestContext.getCurrentInstance();
```

### RequestContext API

Method	Description
isAjaxRequest()	Returns a boolean value if current request is a PrimeFaces ajax request.
addCallBackParam(String name, Object value)	Adds parameters to ajax callbacks like oncomplete.
addPartialUpdateTarget(String target);	Specifies component(s) to update at runtime.

### Callback Parameters

There may be cases where you need values from backing beans in ajax callbacks. Suppose you have a form in a p:dialog and when the user ends interaction with form, you need to hide the dialog or if there're any validation errors, form needs to be open. If you only add dialog.hide() to the oncomplete event of a p:commandButton in dialog, it'll always hide the dialog even it still needs to be open.

Callback Parameters are serialized to JSON and provided as an argument in ajax callbacks.

```
<p:commandButton actionListener="#{bean.validate}"
    oncomplete="handleComplete(xhr, status, args)" />
```

```
public void validate(ActionEvent actionEvent) {
    //isValid = calculate isValid
    RequestContext requestContext = RequestContext.getCurrentInstance();
    requestContext.addCallbackParam("isValid", true or false);
}
```

isValid parameter will be available in handleComplete callback as;

```
<script type="text/javascript">
    function handleComplete(xhr, status, args) {
        var isValid = args.isValid;
        if(isValid)
            dialog.hide();
    }
</script>
```

You can add as many callback parameters as you want with `addCallbackParam` API. Each parameter is serialized as JSON and accessible through `args` parameter so pojos are also supported just like primitive values.

Following example sends a pojo called `User` that has properties like `firstname` and `lastname` to the client.

```
public void validate(ActionEvent actionEvent) {
    //isValid = calculate isValid
    RequestContext requestContext = RequestContext.getCurrentInstance();
    requestContext.addCallbackParam("isValid", true or false);
    requestContext.addCallbackParam("user", user);
}
```

```
<script type="text/javascript">
    function handleComplete(xhr, status, args) {
        var firstname = args.user.firstname;
        var lastname = args.user.lastname;
    }
</script>
```

## Default validationFailed

By default `validationFailed` callback parameter is added implicitly, the value of this parameter is true only when a validation error happens at `processValidations` phase of JSF lifecycle.

## Runtime Partial Update Configuration

There may be cases where you need to define which component(s) to update at runtime rather than specifying it declaratively at compile time. `addPartialUpdateTarget` method is added to handle this case. In example below, button `actionListener` decides which part of the page to update on-the-fly.

```
<p:commandButton value="Save" actionListener="#{bean.save}" />

<p:panel id="panel"> ... </p:panel>

<p:dataTable id="table"> ... </p:panel>
```

```
public void save(ActionEvent actionEvent) {  
    //boolean outcome = ...  
    RequestContext requestContext = RequestContext.getCurrentInstance();  
  
    if(outcome)  
        requestContext.addPartialUpdateTarget("panel");  
    else  
        requestContext.addPartialUpdateTarget("table");  
}
```

When the save button is clicked, depending on the outcome, you can either configure the datatable or the panel to be updated with ajax response.

## 9.2 EL Functions

PrimeFaces provides built-in EL extensions that are helpers to common use cases.

### Common Functions

Function	Description
component('id')	Returns clientId of the component with provided server id parameter. This function is useful if you need to work with javascript.
widgetVar('id')	Provides the widgetVar of a component.

Component

```
<h:form id="form1">
    <h:inputText id="name" />
</h:form>

//#{p:component('name')} returns 'form1:name'
```

WidgetVar

```
<p:dialog id="dlg">
    //contents
</p:dialog>

<p:commandButton type="button" value="Show" onclick="#{p:widgetVar('dlg')}.show()" />
```

### Page Authorization

Function	Description
ifGranted(String role)	Returns a boolean value if user has given role or not.
ifAllGranted(String roles)	Returns a boolean value if has all of the given roles or not.
ifAnyGranted(String roles)	Returns a boolean value if has any of the given roles or not.
ifNotGranted(String roles)	Returns a boolean value if has all of the given roles or not.
remoteUser()	Returns the name of the logged in user.
userPrincipal()	Returns the principal instance of the logged in user.

```
<p:commandButton rendered="#{p:ifGranted('ROLE_ADMIN')}" />  
<h:inputText disabled="#{p:ifGranted('ROLE_GUEST')}" />  
<p:inputMask rendered="#{p:ifAllGranted('ROLE_EDITOR, ROLE_READER')}" />  
<p:commandButton rendered="#{p:ifAnyGranted('ROLE_ADMIN, ROLE_EDITOR')}" />  
<p:commandButton rendered="#{p:ifNotGranted('ROLE_GUEST')}" />  
<h:outputText value="Welcome: #{p:remoteUser}" />
```

# 10. Portlets

PrimeFaces supports portlet environments based on JSF 2 and Portlet 2 APIs. A portlet bridge is necessary to run a JSF application as a portlet and we've tested the bridge from portletfaces project. A kickstart example is available at PrimeFaces examples repository;

<http://primefaces.googlecode.com/svn/examples/trunk/prime-portlet>

## 10.1 Dependencies

Only necessary dependency compared to a regular PrimeFaces application is the JSF bridge, 2.0.0-BETA3 is the latest version of portletfaces at the time of writing. Here's maven dependencies configuration from portlet sample.

```
<dependencies>

    <dependency>
        <groupId>javax.faces</groupId>
        <artifactId>jsf-api</artifactId>
        <version>2.0</version>
    </dependency>

    <dependency>
        <groupId>com.sun.faces</groupId>
        <artifactId>jsf-impl</artifactId>
        <version>2.0.4-b09</version>
    </dependency>

    <dependency>
        <groupId>org.primefaces</groupId>
        <artifactId>primefaces</artifactId>
        <version>2.2</version>
    </dependency>

    <dependency>
        <groupId>javax.portlet</groupId>
        <artifactId>portlet-api</artifactId>
        <version>2.0</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>org.portletfaces</groupId>
        <artifactId>portletfaces-bridge</artifactId>
        <version>2.0.0-BETA3</version>
    </dependency>

</dependencies>
```

## 10.2 Configuration

### portlet.xml

Portlet configuration file should be located under WEB-INF folder. This portlet has two modes, view and edit.

```
<?xml version="1.0"?>
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd"
version="2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd
http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd">
<portlet>
  <portlet-name>1</portlet-name>
  <display-name>PrimeFaces Portlet</display-name>
  <portlet-class>org.portletfaces.bridge.GenericFacesPortlet</portlet-class>
  <init-param>
    <name>javax.portlet.faces.defaultViewId.view</name>
    <value>/view.xhtml</value>
  </init-param>
  <init-param>
    <name>javax.portlet.faces.defaultViewId.edit</name>
    <value>/edit.xhtml</value>
  </init-param>
  <supports>
    <mime-type>text/html</mime-type>
    <portlet-mode>view</portlet-mode>
    <portlet-mode>edit</portlet-mode>
  </supports>
  <portlet-info>
    <title>PrimeFaces Portlet</title>
    <short-title>PrimeFaces Portlet</short-title>
    <keywords>JSF 2.0</keywords>
  </portlet-info>
</portlet>
</portlet-app>
```

### web.xml

Faces Servlet is only necessary to initialize JSF framework internals.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/
j2ee/web-app_2_5.xsd" version="2.5">
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
</web-app>
```

## faces-config.xml

An empty faces-config.xml seems to be necessary otherwise bridge is giving an error.

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
javaee/web-facesconfig_2_0.xsd"
    version="2.0">

</faces-config>
```

## liferay-portlet.xml

Liferay portlet configuration file is an extension to standard portlet configuration file.

```
<?xml version="1.0"?>
<liferay-portlet-app>
    <portlet>
        <portlet-name>1</portlet-name>
        <instanceable>true</instanceable>
        <ajaxable>false</ajaxable>
    </portlet>
</liferay-portlet-app>
```

## liferay-display.xml

Display configuration is used to define the location of your portlet in liferay menu.

```
<?xml version="1.0"?>
<!DOCTYPE display PUBLIC "-//Liferay//DTD Display 5.1.0//EN" "http://www.liferay.com/
dtd/liferay-display_5_1_0.dtd">

<display>
    <category name="category.sample">
        <portlet id="1" />
    </category>
</display>
```

## Pages

That is it for the configuration, a sample portlet page is a partial version of the regular page to provide only the content without html and body tags.

*edit.xhtml*

```

<f:view xmlns="http://www.w3.org/1999/xhtml"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:ui="http://java.sun.com/jsf/facelets"
    xmlns:p="http://primefaces.prime.com.tr/ui">

    <h:head></h:head>

    <h:form>

        <h:panelGrid id="grid" columns="2" cellpadding="10px">

            <f:facet name="header">
                <p:messages id="messages" />
            </f:facet>

            <h:outputText value="Total Amount: " />
            <h:outputText value="#{gambitController.amount}" />

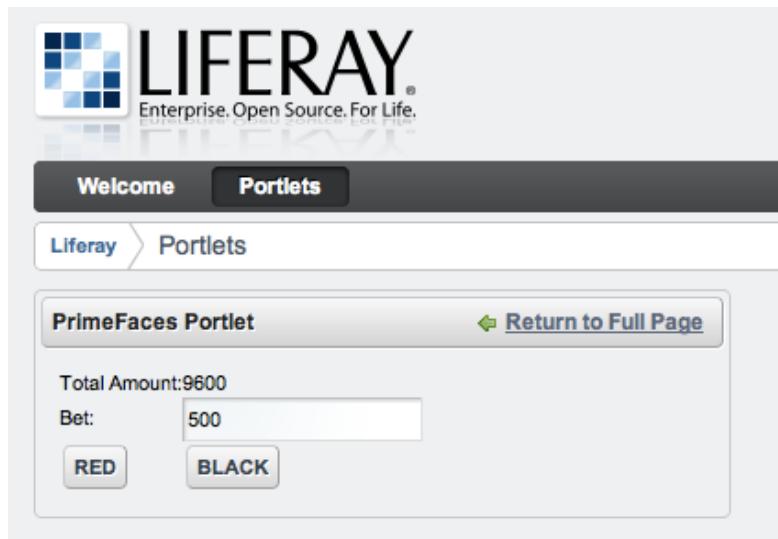
            <h:outputText value="Bet:" />
            <h:inputText value="#{gambitController.bet}" />

            <p:commandButton value="RED"
                actionListener="#{gambitController.playRed}" update="@parent" />
            <p:commandButton value="BLACK"
                actionListener="#{gambitController.playBlack}" update="@parent" />
        </h:panelGrid>

    </h:form>

</f:view>

```



# 11. Integration with Java EE

PrimeFaces is all about front-end and can be backed by your favorite enterprise application framework. Following frameworks are fully supported;

- Spring Core (JSF Centric JSF-Spring Integration)
- Spring WebFlow (Spring Centric JSF-Spring Integration)
- EJBs

We've created [sample applications](#) to demonstrate several technology stacks involving PrimeFaces and JSF at the front layer. Source codes of these applications are available at the PrimeFaces subversion repository and they're deployed online time to time.

## CDI and EJBs

PrimeFaces fully supports a JAVA EE 6 environment with CDI and EJBs.

## Spring WebFlow

We as PrimeFaces team work closely with Spring WebFlow team, PrimeFaces is suggested by SpringSource as the preferred JSF component suite for SWF applications. SpringSource repository has two samples based on SWF-PrimeFaces; a [small showcase](#) and [booking-faces](#) example.

## Seam 2

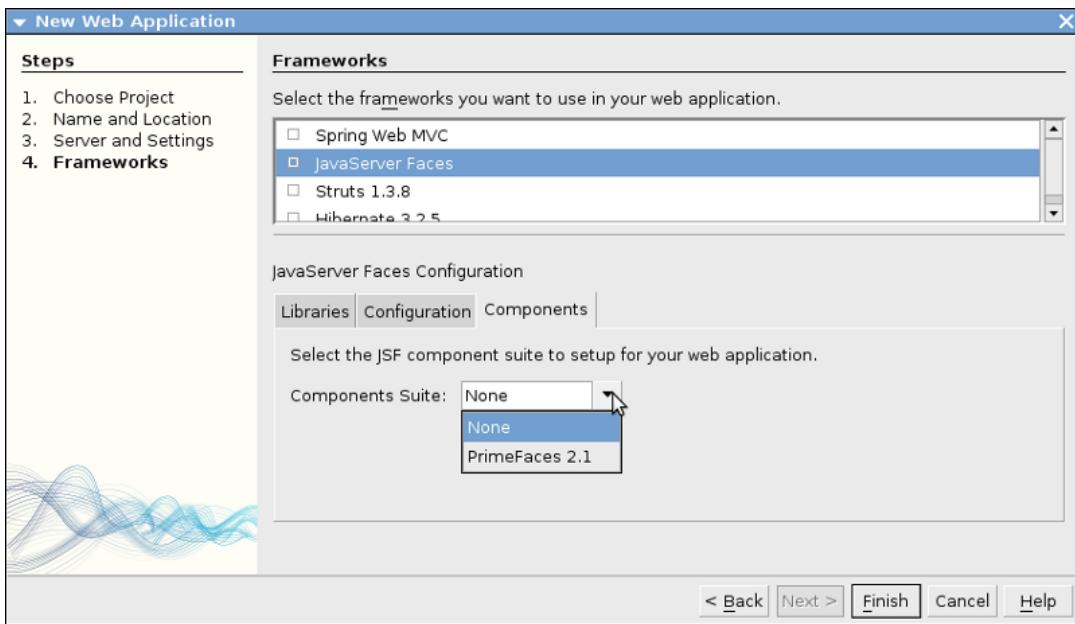
Seam 2 does not officially support JSF 2 thus PrimeFaces however following solution is known to be working;

<https://github.com/heyoulin/seam2jsf2>

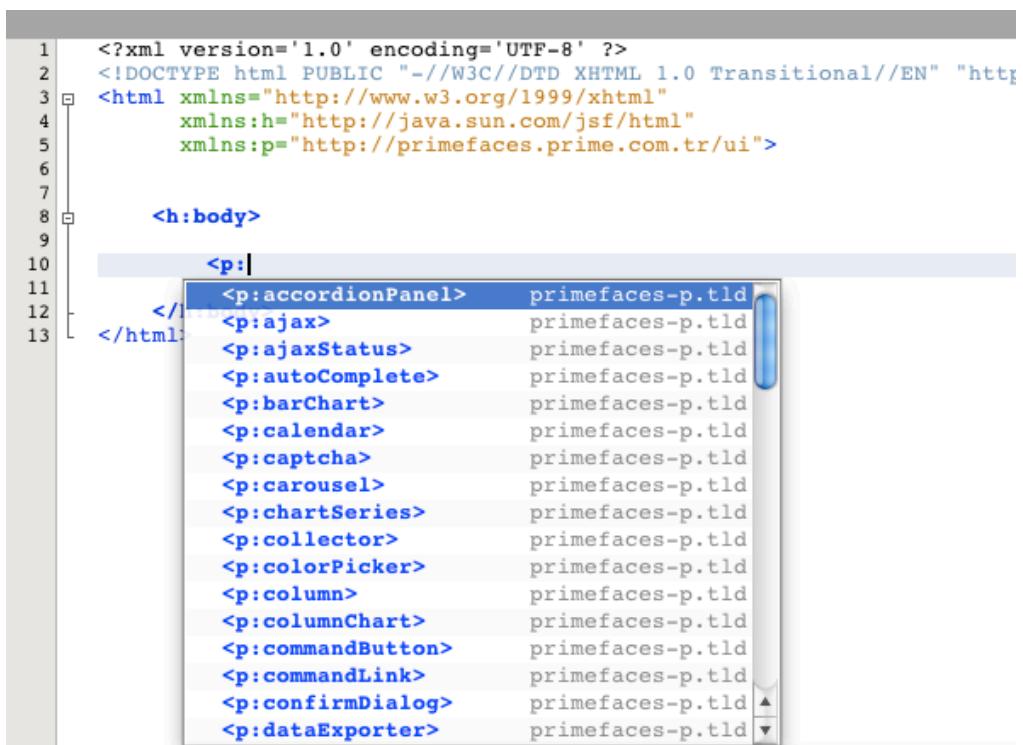
# 12. IDE Support

## 12.1 NetBeans

NetBeans 7.0+ bundles PrimeFaces, when creating a new project you can select PrimeFaces from components tab;



Code completion is supported by NetBeans 6.9+ ;



A screenshot of the Eclipse IDE interface. On the left is a code editor window displaying an XHTML file with JSF tags. A tooltip is open over the tag `<p:accordionPanel>`, listing several attributes: `activeIndex`, `binding`, `id`, `multipleSelection`, `rendered`, `speed`, `style`, and `styleClass`. The code editor shows the following XML structure:

```

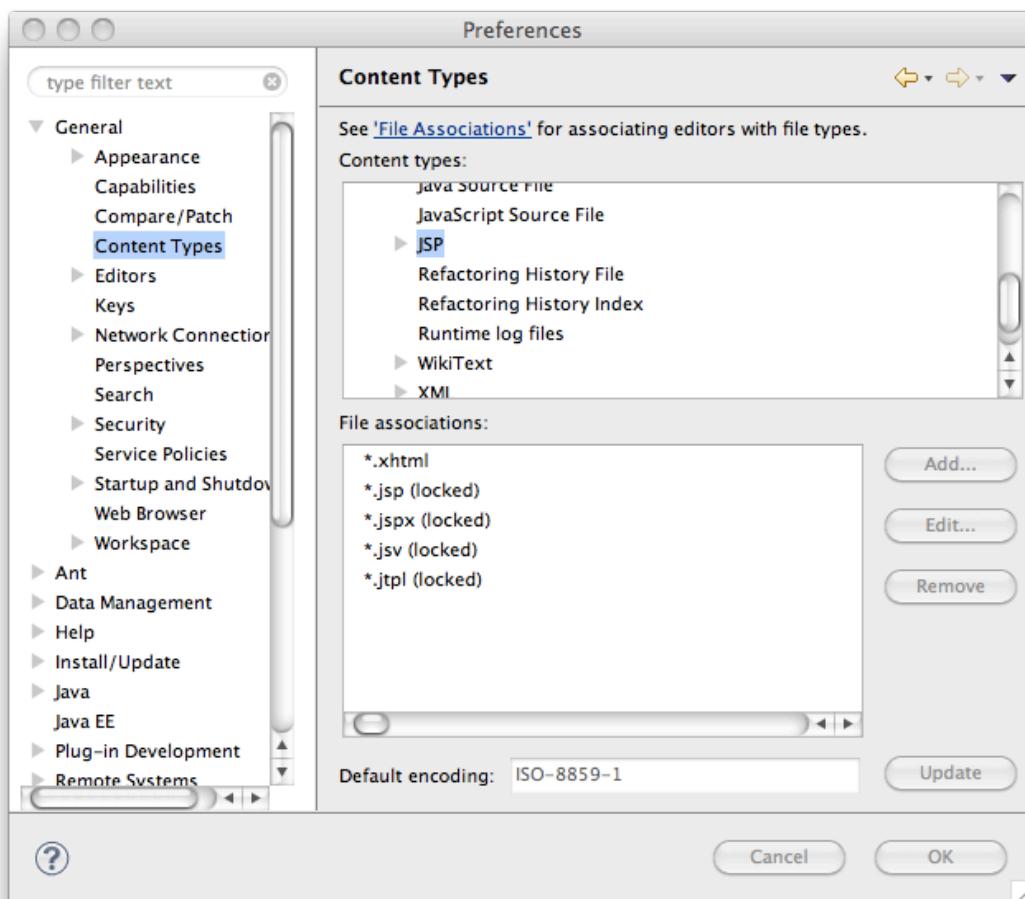
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "h
3  <html xmlns="http://www.w3.org/1999/xhtml"
4      xmlns:h="http://java.sun.com/jsf/html"
5      xmlns:p="http://primefaces.prime.com.tr/ui">
6
7
8      <h:body>
9
10     <p:accordionPanel> | 
11
12     </h:body>
13 </html>

```

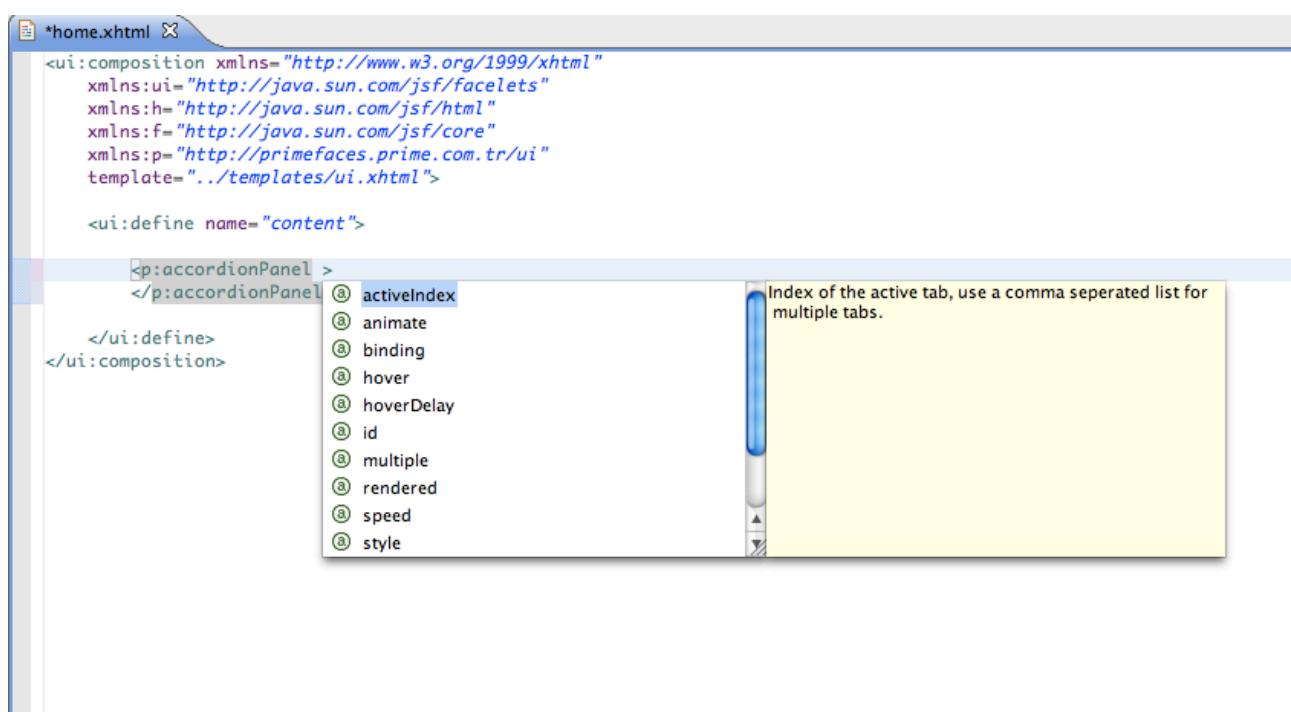
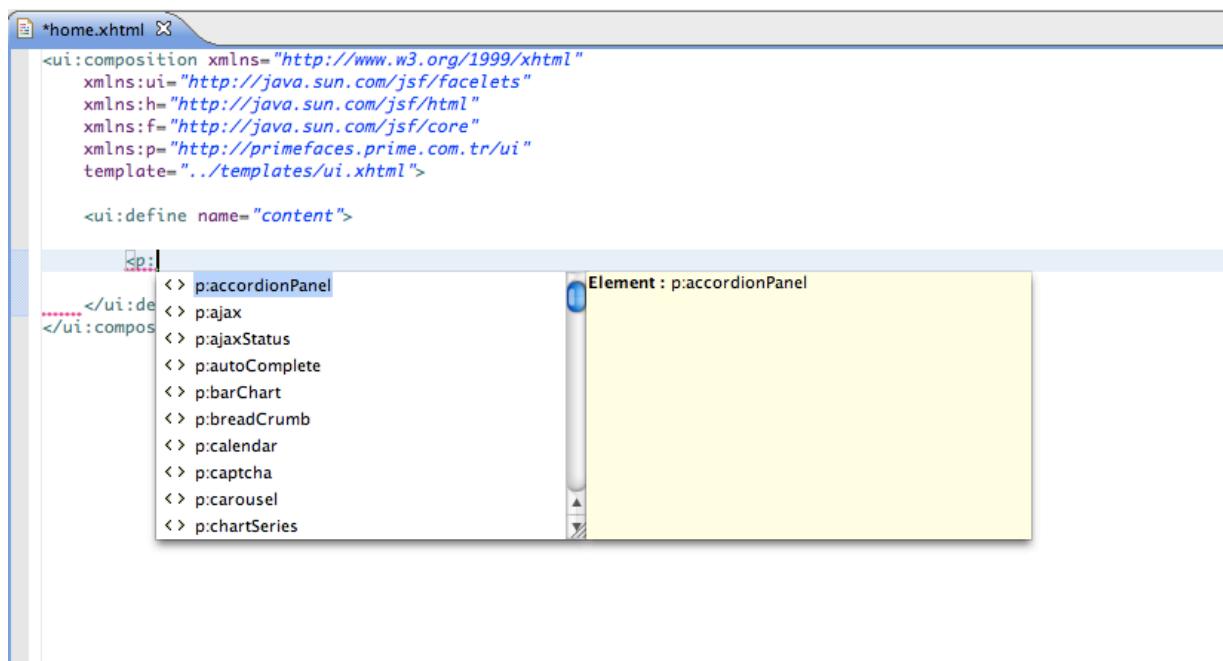
## 12.2 Eclipse

Code completion works out of the box for Eclipse Helios when JSF facet is enabled, older Eclipse requires a little hack to enable completion support with Facelets.

Open *Preferences* -> *General* -> *Content Types* -> *Text* -> *JSP* and add \*.xhtml extension to the list.



With this setting, PrimeFaces components can get tag/attribute completion when opened with jsp editor for Eclipse versions prior to Helios.



# 13. Project Resources

## Documentation

This guide is the main resource for documentation, for additional documentation like apidocs, taglib docs, wiki and more please visit;

<http://www.primefaces.org/documentation.html>

## Support Forum

PrimeFaces discussions take place at the support forum. Forum is public to everyone and registration is required to do a post.

<http://primefaces.prime.com.tr/forum>

## Source Code

PrimeFaces source is at google code subversion repository.

<http://primefaces.googlecode.com/svn>

## Issue Tracker

PrimeFaces issue tracker uses google code's issue management system. Please use the forum before creatin an issue instead.

<http://code.google.com/p/primefaces/issues/list>

## Online Demo

PrimeFaces ShowCase demo is deployed online at;

<http://www.primefaces.org/showcase>

## Social Networks

You can follow PrimeFaces on twitter using [@primefaces](#), join the Facebook and [LinkedIn](#) groups.

# 14. FAQ

## 1. Who develops PrimeFaces?

PrimeFaces is developed and maintained by Prime Technology, a Turkish software development company specialized in Agile Software Development, JSF and Java EE.

## 2. How can I get support?

Support forum is the main area to ask for help, it's publicly available and free registration is required before posting. Please do not email the developers of PrimeFaces directly and use support forum instead.

## 3. Is enterprise support available?

Yes, enterprise support is also available. Please visit support page on PrimeFaces website for more information.

<http://www.primefaces.org/support.html>

## 4. I'm using x component library in my project, can primefaces be compatible?

Component Suite compatibility is a goal of JSF 2.0, PrimeFaces should work with any component suite that fully supports JSF 2.

## 5. Where is the source for the example demo applications?

Source code of demo applications are in the svn repository of PrimeFaces at /examples/trunk folder. Nightly snapshot builds of each sample application are deployed at Prime Technology Maven Repository.

## 6. With facelets some components like charts do not work in Safari or Chrome but there's no problem with Firefox.

The common reason is the response mimeType when using with PrimeFaces with *facelets*. You need to make sure responseType is "text/html". With facelets you can use the <f:view contentType="text/html"> to enforce this setting.

## 7. My page does not navigate with PrimeFaces commandButton and commandLink.?

If you'd like to navigate within an ajax request, use redirect instead of forward or set ajax to false.

## 8. Where can I get an unreleased snapshot?

Nightly snapshot builds of a future release is deployed at <http://repository.prime.com.tr>.

## 9. What is the license PrimeFaces have?

PrimeFaces is free to use and licensed under Apache License V2.

## 10. Can I use PrimeFaces in a commercial software?

Yes, Apache V2 License is a commercial friendly library. PrimeFaces does not bundle any third party software that conflicts with Apache.

THE END