PRIMEFACES

USER'S GUIDE

Authors

Çagatay Çivici
Yigit Darçın

Covers 1.1 and 2.1
Last Update: 26.07.2010

1. Introduction 9

What is PrimeFaces? 9

2. Setup 10

2.1 Download 10

2.2 Dependencies 11

2.3 Configuration 11

2.3.1 JSF 1.2 with PrimeFaces 1.x 11

2.3.2 JSF 2.0 with PrimeFaces 2.x 12

2.4 Hello World 13

3. Component Suite 14

3.1 AccordionPanel 14

3.2 AjaxStatus 17

3.3 AutoComplete 20

3.4 BreadCrumb 26

3.5 Calendar 29

3.6 Captcha 37

3.7 Carousel 40

3.8 Charts 46

3.8.1 Pie Chart 46

3.8.2 Line Chart 49

3.8.3 Column Chart 52

3.8.4 Stacked Column Chart 54

3.8.5 Bar Chart 56

3.8.6 StackedBar Chart 58

3.8.7 Chart Series 60

3.8.8 Skinning Charts 61

3.8.9 Real-Time Charts *64*

3.8.10 Interactive Charts *66*

3.9 Collector **68**

3.10 Color Picker **70**

3.11 Column **74**

3.12 CommandButton **75**

3.13 CommandLink **80**

3.14 ConfirmDialog **83**

3.15 ContextMenu **86**

3.16 Dashboard **89**

3.17 DataExporter **94**

3.18 DataGrid **97**

3.19 DataList **102**

3.20 DataTable **108**

3.21 Dialog **122**

3.22 Drag&Drop **127**

3.23 Dock **135**

3.24 Editor **137**

3.25 Effect **142**

3.26 FileDownload **145**

3.27 FileUpload **147**

3.28 Focus **153**

3.29 GMap **155**

3.30 GMapInfoWindow **170**

3.31 GraphicImage **171**

3.32 GraphicText 176

3.33 Growl 178

3.34 HotKey 181

3.35 IdleMonitor 184

3.36 ImageCompare 187

3.37 ImageCropper 189

3.38 ImageSwitch 193

3.39 Inplace 196

3.40 InputMask 198

3.41 Keyboard 201

3.42 Layout 209

3.43 LayoutUnit 217

3.44 LightBox 219

3.45 LinkButton 224

3.46 Media 226

3.47 Menu 229

3.48 Menubar 235

3.49 MenuButton 239

3.50 MenuItem 241

3.51 Message 243

3.52 Messages 245

3.53 NotificationBar 247

3.54 OutputPanel 250

3.55 Panel 252

3.56 Password Strength 257

3.57 PickList 262

3.58 Poll 267

3.59 Printer 270

3.60 ProgressBar 272

3.61 Push 276

3.62 Rating 277

3.63 RemoteCommand 280

3.64 Resizable 282

3.65 Resource 285

3.66 Resources 286

3.67 Schedule 288

3.68 ScheduleEventDialog 301

3.69 Sheet 303

3.70 Slider 304

3.71 Spinner 308

3.72 Submenu 313

3.73 Spreadsheet 314

3.74 Stack 318

3.75 Tab 320

3.76 TabView 321

3.77 Terminal 325

3.78 ThemeSwitcher 328

3.79 Tooltip 330

3.80 Tree 334

3.81 TreeNode 348

3.82 TreeTable 349

3.83 UIAjax 353

3.84 Watermark 356

3.85 Wizard 358

4. TouchFaces 364

4.1 Getting Started with TouchFaces 364

4.2 Views 366

4.3 Navigations 369

4.4 Ajax Integration 371

4.5 Sample Applications 372

4.6 TouchFaces Components 373

4.6.1 Application 373

4.6.2 NavBarControl 374

4.6.3 RowGroup 375

4.6.4 RowItem 376

4.6.5 Switch 377

4.6.6 TableView 379

4.6.7 View 380

5. Partial Rendering and Processing 381

5.1 Partial Rendering 381

5.1.1 Infrastructure 381

5.1.2 Using IDs 381

5.1.3 Notifying Users 384

5.1.4 Bits&Pieces 384

5.2 Partial Processing 384

5.2.1 Partial Validation 384

5.2.2 Keywords *385*

5.2.3 Using Ids *386*

5.2.4 Ajax vs Non-Ajax *386*

6. Ajax Push/Comet **387**

6.1 Atmosphere **387**

6.2 PrimeFaces Push **388**

6.2.1 Setup *388*

6.2.2. CometContext *389*

6.2.3 Push Component *389*

7. Javascript API **391**

7.1 PrimeFaces Global Object **391**

7.2 Namespaces **391**

7.3 Ajax API **392**

8. Skinning **395**

9.1 Applying a Theme **396**

9.2 Creating a New Theme **396**

9.3 How Skinning Works **397**

9. Utilities **399**

9.1 RequestContext **399**

9.2 EL Functions **401**

10. Integration with Java EE **403**

11. IDE Support **404**

11.1 NetBeans **404**

11.2 Eclipse **404**

12. Portlets 407

13. Project Resources 409

14. FAQ 410

1. Introduction

What is PrimeFaces?

PrimeFaces is an open source component suite for Java Server Faces featuring 90+ 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.

- 90+ 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)
- Compatible and Lightweight.
- Skinning Framework with 25+ pre-designed themes.
- Extensive documentation.

Prime Technology

PrimeFaces is maintained by Prime Technology, a Turkish software development company specialized in Agile consulting, Enterprise Java and outsource software development. 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

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>1.1.RC1 or 2.1.RC1</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 1.2+ implementation as mandatory dependencies. Other than these required dependencies, there're some optional libraries for certain features.

Dependency	Version *	Type	Description
JSF runtime	1.2.x or 2.x	Required	Apache MyFaces or Sun Mojarra
itext	1.4.8	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

2.3.1 JSF 1.2 with PrimeFaces 1.x

Resource Servlet

Resource Servlet must be configured in web.xml.

```
<servlet>
   <servlet-name>Resource Servlet</servlet-name>
   <servlet-class>org.primefaces.resource.ResourceServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>Resource Servlet</servlet-name>
   <url-pattern>/primefaces_resource/*</url-pattern>
</servlet-mapping>
```

Resources Component

Resource component needs to be present on a page that has PrimeFaces components, this component outputs the link and script tags that are necessary for PrimeFaces components to work. The ideal place to put resources component would be the html head element.

```
<head>
     <p:resources />
</head>
```

A tip regarding p:resources is to add this component to the facelets or jsp template once, so that it gets added to each page automatically using the application.

2.3.2 JSF 2.0 with PrimeFaces 2.x

Resource Servlet

Although PrimeFaces 2.x uses JSF2 resource APIs to place resources on page, due to limitations of JSF2 resource loading mechanism, PrimeFaces Resource Servlet is required to stream the resources from the bundle. If you're running PrimeFaces in a Servlet 3.0 environment like Glassfish V3, this servlet is auto-registered so you don't need to configure it manually.

```
<servlet>
   <servlet-name>Resource Servlet</servlet-name>
   <servlet-class>org.primefaces.resource.ResourceServlet</servlet-class>
</servlet>

<servlet-mapping>
   <servlet-name>Resource Servlet</servlet-name>
   <url-pattern>/primefaces_resource/*</url-pattern>
</servlet-mapping>
```

Allowing Text Children

When using Mojarra 2.x, enable allowTextChildren configuration.

```
<context-param>
     <param-name>com.sun.faces.allowTextChildren</param-name>
     <param-value>true</param-value>
</context-param>
```

2.4 Hello World

That is all for configuration, now define the taglib to import PrimeFaces in your pages and try a component to test if setup is working.

Taglib

If you're a facelets user, the xml namespace configuration would be;

```
xmlns:p="http://primefaces.prime.com.tr/ui"
```

If you're using jsp the taglib definition is;

```
<%@ taglib uri="http://primefaces.prime.com.tr/ui" prefix="p" %>
```

Try a component

For JSF 1.2 and PrimeFaces 1.x an example page would be;

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

And with JSF 2.0 and PrimeFaces 2.x. (Note that you don't need p:resources but need h:head in this case);

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

3. Component Suite

3.1 AccordionPanel

AccordionPanel is a container component that renders it’s children in seperate tabs and displays an animation when a tab is being collapsed or expanded.

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. T hrough 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	accordionPanel
Tag Class	org.primefaces.component.accordionpanel.AccordionpanelTag
Component Class	org.primefaces.component.accordionpanel.Accordionpanel
Component Type	org.primefaces.component.AccordionPanel
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.AccordionPanelRenderer
Renderer Class	org.primefaces.component.accordionpanel.AccordionPanelRenderer

Attributes

Name	Default	Type	Description
id	Assigned by JSF	String	Unique identifier of the component

Name	Default	Type	Description
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.
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 enables, accordion panel fills the height of it's parent.
event	click	String	Client side event to toggle the tabs.
widgetVar	null	String	Variable name of the client side widget.

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>
     <p:tab title="Third Tab Title">
          any set of components...
     </p:tab>

     ... any number of tabs

</p:accordionPanel>
```

Events

By default toggling of content happens when a tab header is clicked, you can also specify the client side to trigger toggling. For example for hover;

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

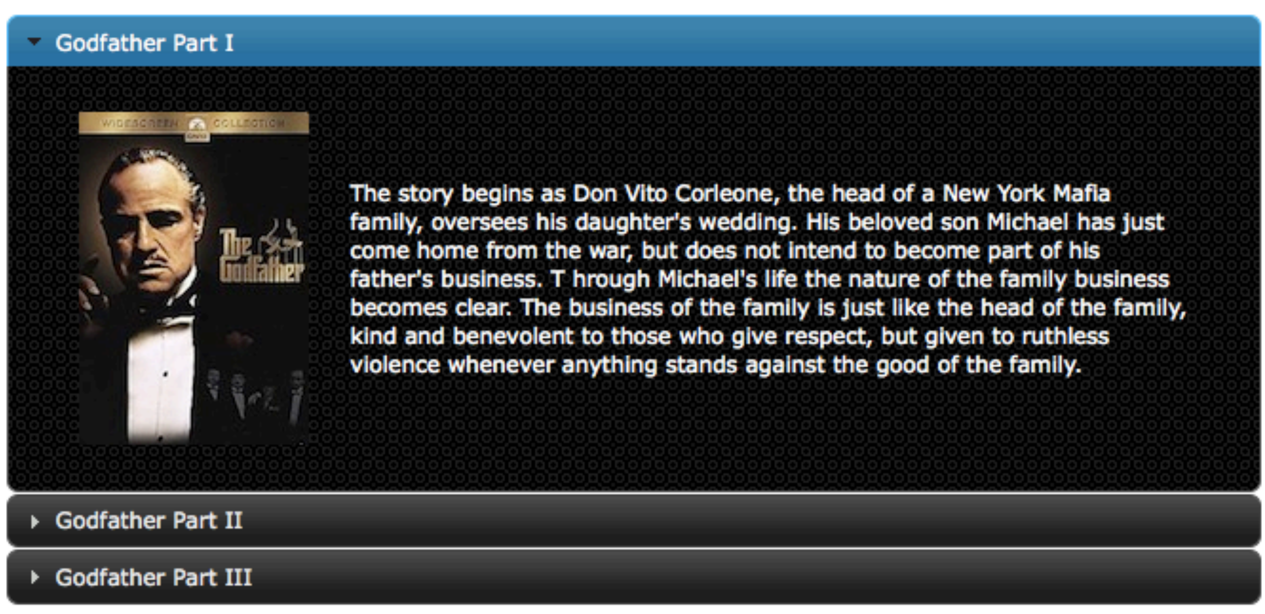
Skinning

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

Following is the list of structural style classes;

Class	Applies
.ui-accordion	Main container element
.ui-accordion-header	Tab header
.ui-accordion-content	Tab content
.ui-accordion-content-active	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;



3.2 AjaxStatus

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

Info

Tag	ajaxStatus
Tag Class	org.primefaces.component.ajaxstatus.AjaxStatusTag
Component Class	org.primefaces.component.ajaxstatus.AjaxStatus
Component Type	org.primefaces.component.AjaxStatus
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.AjaxStatusRenderer
Renderer Class	org.primefaces.component.ajaxstatus.AjaxStatusRenderer

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
onstart	null	String	Javascript event handler to be executed after ajax requests start.
oncomplete	null	String	Javascript event handler to be executed after ajax requests complete.
onprestart	null	String	Javascript event handler to be executed before ajax requests start.
onsuccess	null	String	Javascript event handler to be executed after ajax requests is completed succesfully.
onerror	null	String	Javascript event handler to be executed when an ajax request fails.

Name	Default	Type	Description
style	null	String	Style of the html element containing the facets.
styleClass	null	String	Style class of the html element containing the facets.
widgetVar	null	String	Name of the client side widget.

Getting started with AjaxStatus

AjaxStatus uses facets to represent the ajax 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”>
		<h:outputText value=”Loading...” />
	</f:facet>

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

More callbacks

Other than start and complete there're three more callback facets you can use. These are; prestart, success and error.

```
<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=”Done!” />
	</f:facet>
</p:ajaxStatus>
```

Custom Events

If you want to execute custom javascript instead of the default usage with facets, use on* event handlers. These are the event handler versions of facets.

```
<p:ajaxStatus onstart=”alert(‘Start’)” oncomplete=”alert(‘End’)”/>
```

Animations

Generally, it’s fancier to display animated gifs with ajax requests rather than plain texts.

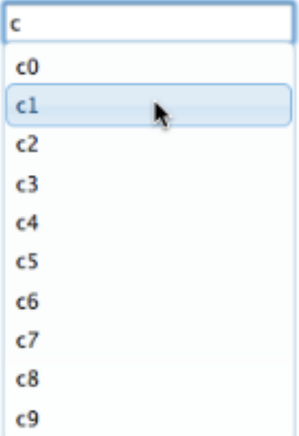
```
<p:ajaxStatus>
     <f:facet name=”start”>
          <h:graphicImage value=”ajaxloading.gif” />
     </f:facet>

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

Skinning AjaxStatus

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

3.3 AutoComplete

AutoComplete provides suggestions while an input is being typed.



Info

Tag	**autoComplete**
Tag Class	**org.primefaces.component.autocomplete.AutoCompleteTag**
Component Class	**org.primefaces.component.autocomplete.AutoComplete**
Component Type	**org.primefaces.component.AutoComplete**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.AutoCompleteRenderer**
Renderer Class	**org.primefaces.component.autocomplete.AutoCompleteRenderer**

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 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	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	MethodExpression	A method expression that refers to a method validationg the input.
valueChangeListener	null	MethodExpression	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 fields.
widgetVar	null	String	Name of the client side widget.
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	MethodExpression	Method to be called to fetch the 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.

Name	Default	Type	Description
forceSelection	FALSE	Boolean	When enabled, autoComplete only accepts input from the selection list.
selectListener	null	MethodExpression	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	Javascript event handler to be called before ajax request to load suggestions begins.
oncomplete	null	String	Javascript event handler to be called after ajax request to load suggestions completes.
disabled	FALSE	Boolean	Disables the autocomplete.

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 AutoCompleteBean {

	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;
	}

	//getters and setters
}
```

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=”#{autoCompleteBean.selectedPlayer}”
      completeMethod=”#{autoCompleteBean.completePlayer}”
      var=”player”
      itemLabel=”#{player.name}”
      itemValue=”#{player}”
      converter=”playerConverter”/>
```

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

public class AutoCompleteBean {

      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 = readFromDB(query);

            return players;
      }
}
```

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 containing information about selected item is passed as a parameter. 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 AutoCompleteBean {

     public void handleSelect(SelectEvent event) {
           Object selectedItem = event.getObject();
           //add faces message
     }

     //Getters, Setters and completeMethod
}
```

Client Side Callbacks

onstart and *oncomplete* allow executing 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)" />
```

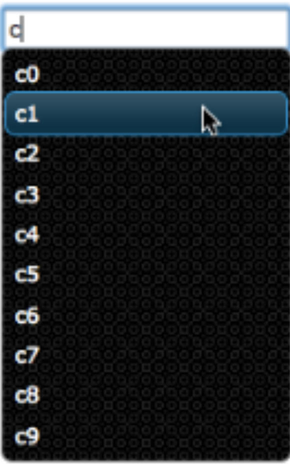
onstart callback gets a *request* parameter and oncomplete gets a *response* parameter, these parameters contain useful information. For example *request.term* is the query string and *response.results* is the suggestion list in json format.

Skinning

Following is the list of structural style classes;

Class	Applies
.ui-autocomplete-input	Input field
.ui-menu	Suggestion menu
.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;



3.4 BreadCrumb

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

Sports 〉 Football 〉 Countries 〉 Spain 〉 F.C. Barcelona 〉 Squad 〉 Lionel Messi

Info

Tag	breadCrumb
Tag Class	org.primefaces.component.breadcrumb.BreadCrumbTag
Component Class	org.primefaces.component.breadcrumb.BreadCrumb
Component Type	org.primefaces.component.BreadCrumb
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.BreadCrumbRenderer
Renderer Class	org.primefaces.component.breadcrumb.BreadCrumbRenderer

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
expandedEndItems	1	Integer	Number of expanded menutitems 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.

Name	Default	Type	Description
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.
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>
```

Animation Configuration

Duration of effects can be customizated using several attributes. Here’s an example;

```
<p:breadCrumb preview=”true” expandEffectDuration=”1000”
                             collapseEffectDuration=”1000”
                             initialCollapseEffectDuration=”1000”>
     //menuitems
</p:breadCrumb>
```

Durations are defined in milliseconds.

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;

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	Seperator 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;

3.5 Calendar

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

Info

Tag	calendar
Tag Class	org.primefaces.component.calendar.CalendarTag
Component Class	org.primefaces.component.calendar.Calendar
Component Type	org.primefaces.component.Calendar
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.CalendarRenderer
Renderer Class	org.primefaces.component.calendar.CalendarRenderer

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	java.util.Date	Value of the component

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	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	MethodExpression	A method binding expression that refers to a method validationg the input
valueChangeListener	null	ValueChangeListe ner	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.
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	inline|popup, Defines how the calendar will be displayed; "inline" only displays a calendar, "popup" displays an input text and a popup button
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

Name	Default	Type	Description
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.
selectListener	null	MethodExpression	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 client side widget

Getting started with Calendar

A java.util.Date type property needs to be defined as the value of the calendar.

```
public class DateBean {

      private Date date;

      //Getter and Setter
}
```

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

Display Modes

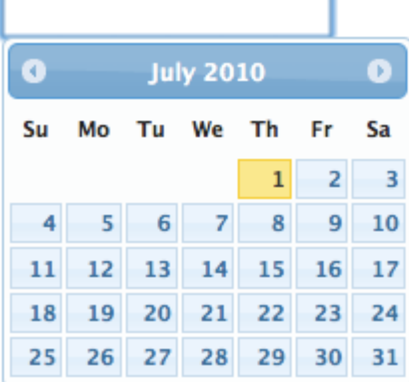
Calendar has two main display modes, “popup”(default) and “inline”.

Inline

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

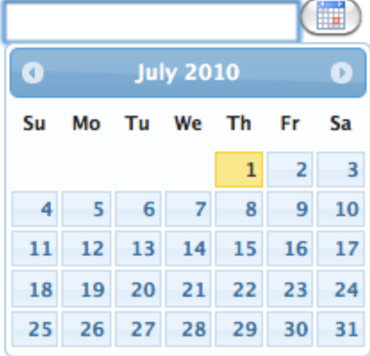
Popup

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



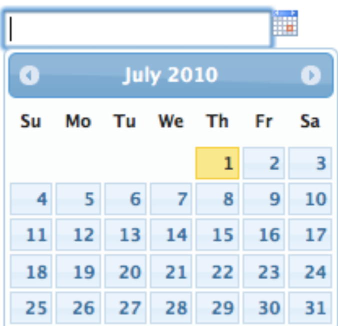
Popup Button

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



Popup Icon Only

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



L11N

Calendar has a built-in converter so there's no need to define a datetime converter. Default pattern is "MM/dd/yyyy" and pattern attribute allows changing the pattern used.

```
<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"/>
```

I18N

By default locale information is retrieved from the view's locale and can be overriden 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"/>
```

52 languages are supported out of the box;

• Afrikaans • Albanian • Arabic • Armenian • Azerbaijani • Basque • Bosnian • Bulgarian • Catalan • Chinese Hong Kong • Chinese Simplified • Chinese Traditional • Croatian • Czech • Danish • Dutch • English UK • English US • Esperanto • Estonian • Faroese • Farsi/Persian • Finnish • French • French/Swiss • German • Greek	• Hebrew • Hungarian • Icelandic • Indonesian • Italian • Japanese • Korean • Latvian • Lithuanian • Malaysian • Norwegian • Polish • Portuguese/Brazilian • Romanian • Russian • Serbian • Serbian (sprski jezik) • Slovak • Slovenian • Spanish • Swedish • Tamil • Thai • Turkish • Ukranian • Vietnamese

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"/>
```

Ajax Selection

Calendar supports instant ajax selection, 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="07/20/2010"/>
```

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;

3.6 Captcha

Captcha is a form validation component based on Recaptcha API.

Info

Tag	**captcha**
Tag Class	**org.primefaces.component.captcha.CaptchaTag**
Component Class	**org.primefaces.component.captcha.Captcha**
Component Type	**org.primefaces.component.Captcha**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.CaptchaRenderer**
Renderer Class	**org.primefaces.component.captcha.CaptchaRenderer**

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 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	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	MethodExpression	A method binding expression that refers to a method validationg 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.
publicKey	null	String	Public recaptcha key for a specific domain
theme	red	String	Theme of the captcha.
language	en	String	Key of the supported languages.

Getting Started with Captcha

Catpcha uses reCaptcha api as the underlying captcha mechanism and captcha has a built-in captcha validator that always checks the value entered with reCaptcha. First thing to do is to sign up to reCaptcha and gain public&private keys. Once you have the keys for your domain, add your private key to web deployment descriptor as follows.

```
<context-param>
    <param-name>org.primefaces.component.captcha.PRIVATE_KEY</param-name>
    <param-value>YOUR_PRIVATE_KEY</param-value>
 </context-param>
```

Once the private key is configured, you can start using captca as;

```
<p:captcha publicKey=”YOUR_PUBLIC_KEY”/>
```

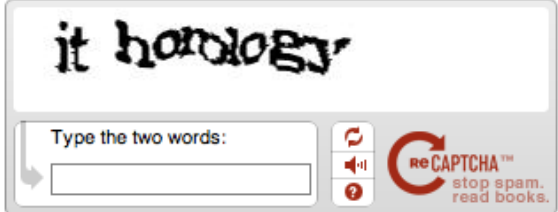
Themes

Captcha supports several themes, note that custom styling is not yet supported. Following are the valid built-in themes.

- red (default)
- white
- blackglass
- clean

Themes are applied via the theme attribute.

```
<p:captcha publicKey=”YOUR_PUBLIC_KEY” theme=”white”/>
```



Languages

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

```
<p:captcha publicKey=”YOUR_PUBLIC_KEY” language=”tr”/>
```

Overriding Validation Messages

By default captcha displays it’s own validation messages, this can be easily overriden by the JSF message bundle mechanism. Corresponding keys are;

Summary	org.primefaces.component.captcha.CaptchaValidator.INVALID
Detail	org.primefaces.component.captcha.CaptchaValidator.INVALID_detail

3.7 Carousel

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

Info

Tag	carousel
Tag Class	org.primefaces.component.carousel.CarouselTag
Component Class	org.primefaces.component.carousel.Carousel
Component Type	org.primefaces.component.Carousel
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.CarouselRenderer
Renderer Class	org.primefaces.component.carousel.CarouselRenderer

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	A value expression that referes to a collection instance to be listed
var	null	String	Name of the request scope based iterator name
rows	3	Integer	Number of visible items per page

Name	Default	Type	Description
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
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	Javascript variable name of the wrapped widget

Getting started with Carousel

Calendar has two main use-cases; data and general content display. To begin with data iteration we will be using a list of cars to display with carousel.

```
 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 carousel to the car list;

```
public class CarBean {

	private List<Car> cars;

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

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

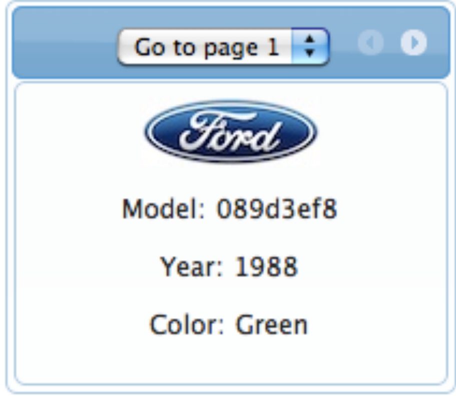
```
<p:carousel value="#{carBean.cars}" var="car">
	<p:graphicImage value="/images/cars/#{car.manufacturer}.jpg"/>
	<h:outputText value="Model: #{car.model}" />
	<h:outputText value="Year: #{car.year}" />
	<h:outputText value="Color: #{car.color}" />
</p:carousel>
```

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.

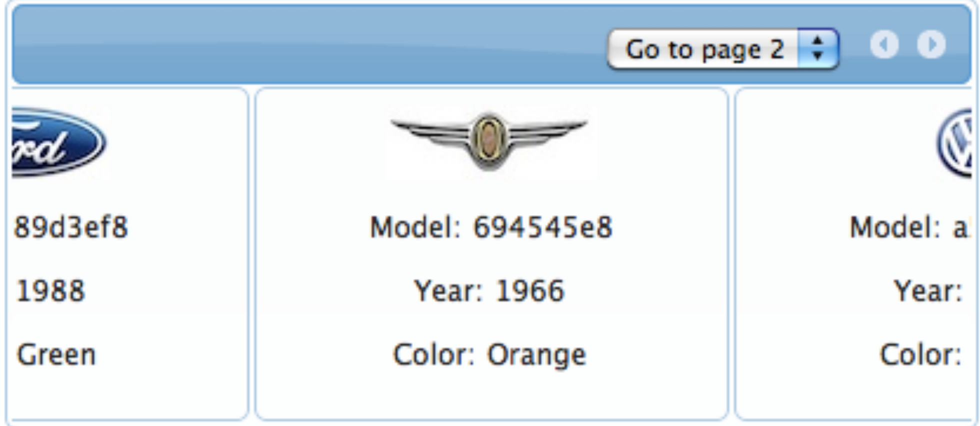
```
<p:carousel value="#{carBean.cars}" var="car" rows="1">
	...
</p:carousel>
```



Reveal Amount

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

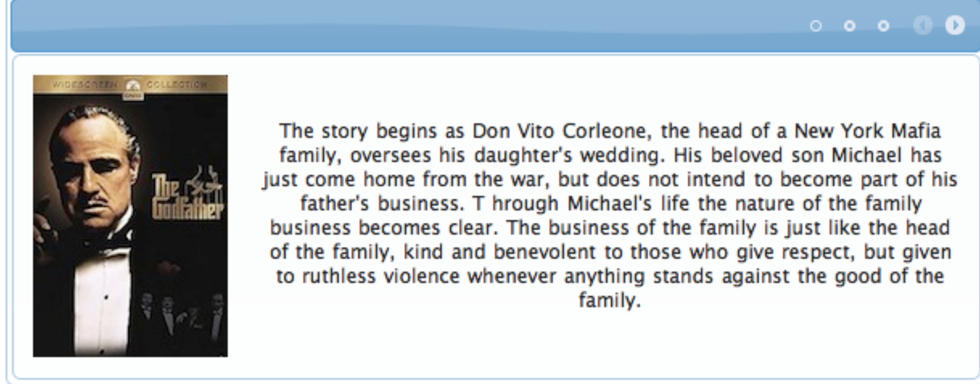
Carousel can display the contents in a slideshow, for this purpose *autoPlayInterval* and *circular* attributes are used. Following carousel displays a collection of images as a slideshow.

```
<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>
```



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.

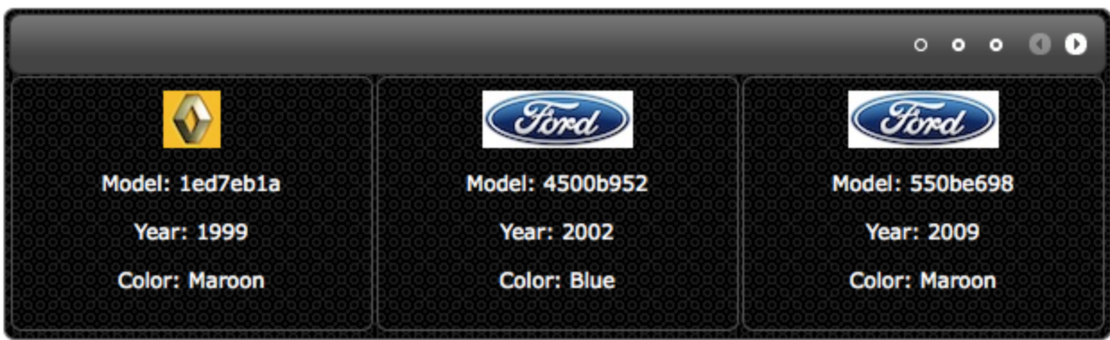
Skinning

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;



3.8 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 also can fire server side events as response to user interaction.

3.8.1 Pie Chart

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

Info

Tag	**pieChart**
Tag Class	**org.primefaces.component.chart.pie.PieChartTag**
Component Class	**org.primefaces.component.chart.pie.PieChart**
Component Type	**org.primefaces.component.chart.PieChart**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.chart.PieChartRenderer**
Renderer Class	**org.primefaces.component.chart.pie.PieChartRenderer**

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	java.util.List	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

Name	Default	Type	Description
update	null	String	Client side id of the component(s) to be updated after async partial submit request
oncomplete	null	String	Javascript event to be called when ajax request for item select event is completed.
itemSelectListener	null	MethodExpression	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
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
widgetVar	null	String	Name of the client side widget

Getting started with PieChart

Chart needs a collection 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. We begin with creating the sale class to represent this model.

```
public class Sale {

	private String brand;
	private int amount;

	public Sale() {}

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

	//getters and setters for brand and amount
}
```

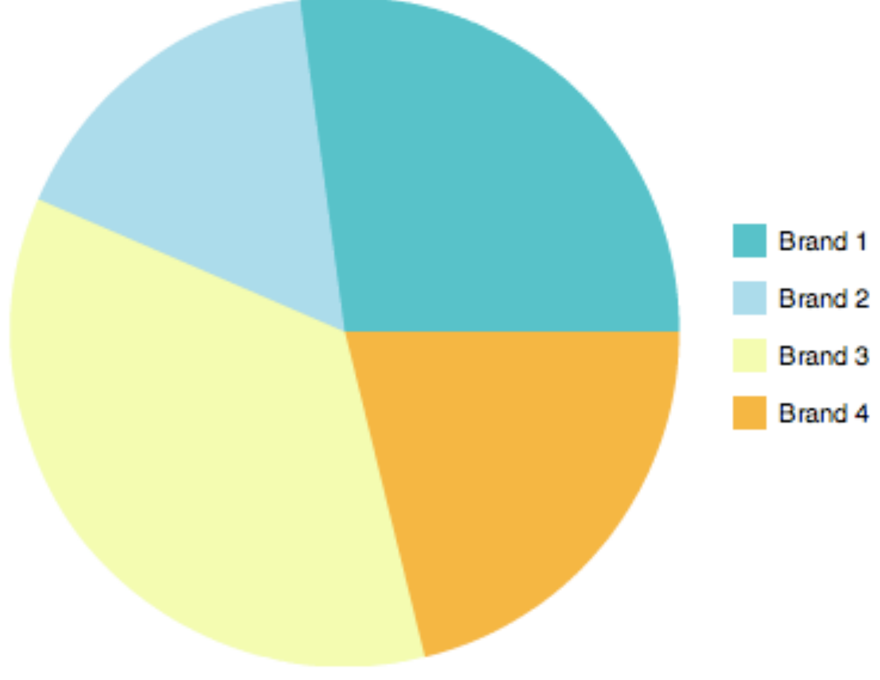
In SaleDisplay bean, a java.util.List holds sale data of the 4 brands.

```
public class SaleDisplayBean {

    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}”  />
```

Output would be;



3.8.2 Line Chart

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

Info

Tag	lineChart
Tag Class	org.primefaces.component.chart.line.LineChartTag
Component Class	org.primefaces.component.chart.line.LineChart
Component Type	org.primefaces.component.chart.LineChart
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.chart.LineChartRenderer
Renderer Class	org.primefaces.component.chart.line.LineChartRenderer

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	java.util.List	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	Client side id of the component(s) to be updated after async partial submit request
oncomplete	null	String	Javascript event to be called when ajax request for item select event is completed.
itemSelectListener	null	MethodExpression	Method expression to listen chart series item select events

Name	Default	Type	Description
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.
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
widgetVar	null	String	Name of the client side widget

Getting started with LineChart

LineChart mainly needs a collection as the value, the xField data for the x-axis and one or more series data each corresponding to a line on the graph. To give an example, we'd display and compare the number of boys and girls year by year who was 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() {}

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

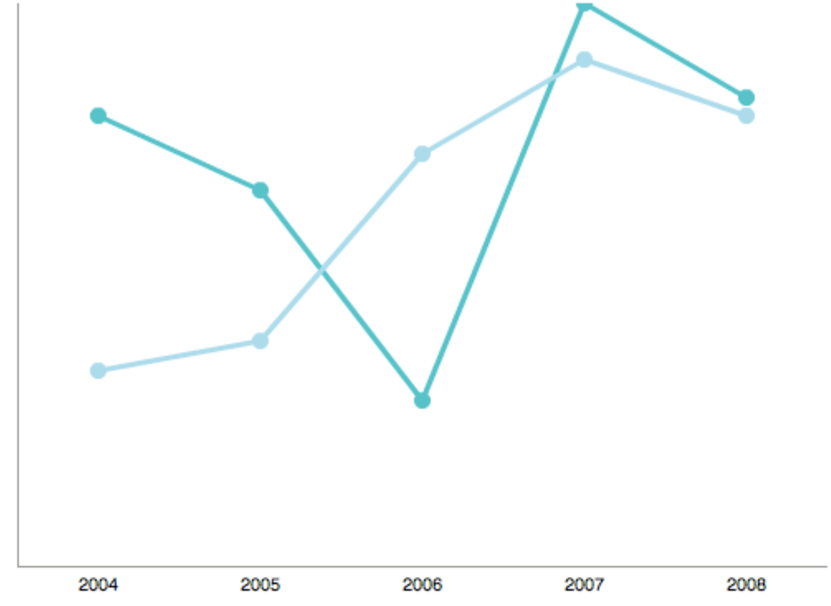
Next thing to do is to prepare the data year by year in BirthDisplayBean.

```
public class BirthDisplayBean {

	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;
	}
}
```

Given this birth p:chartSeriescollection, a linechart can visualize this data as follows;

```
<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>
```

Output of this lineChart would be;



3.8.3 Column Chart

Column chart visualizes one or more series of data using a column graph.

Info

Tag	**columnChart**
Tag Class	**org.primefaces.component.chart.column.ColumnChartTag**
Component Class	**org.primefaces.component.chart.column.ColumnChart**
Component Type	**org.primefaces.component.chart.ColumnChart**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.chart.ColumnChartRenderer**
Renderer Class	**org.primefaces.component.chart.column.ColumnChartRenderer**

Attributes

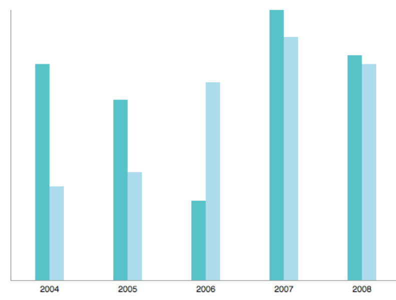
Name	Default	Type	Description
id	Assigned by JSF	String	Unique identifier of the component
rendered	TRUE	boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	java.util.List	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	Client side id of the component(s) to be updated after async partial submit request
oncomplete	null	String	Javascript event to be called when ajax request for item select event is completed.
itemSelectListener	null	MethodExpression	Method expression to listen chart series item select events

Name	Default	Type	Description
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.
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
widgetVar	null	String	Name of the client side widget

Getting started with Column Chart

Column chart usage is very similar to line chart, as an example following column chart displays the birth rate data in the lineChart example. Please see the lineChart section to get more information about the structure of the birth data.

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



3.8.4 Stacked Column Chart

Stacked Column chart is similar to column chart but the columns are stacked per each xField data.

Info

Tag	**stackedColumnChart**
Tag Class	**org.primefaces.component.chart.stackedcolumn. StackedColumnChartTag**
Component Class	**org.primefaces.component.chart.stackedcolumn. StackedColumnChart**
Component Type	**org.primefaces.component.chart.StackedColumnChart**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.chart.StackedColumnChartRenderer**
Renderer Class	**org.primefaces.component.chart.stackedcolumn. StackedColumnChartRenderer**

Attributes

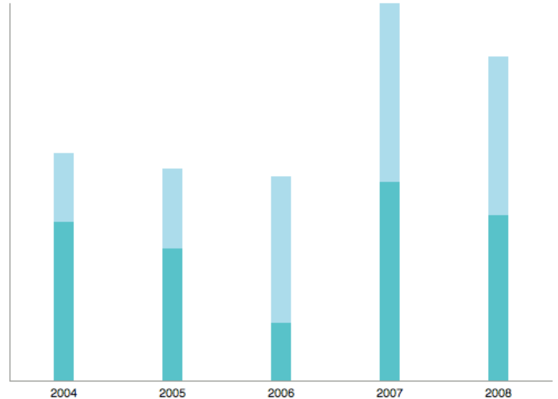
Name	Default	Type	Description
id	Assigned by JSF	String	Unique identifier of the component
rendered	TRUE	boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	java.util.List	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	Client side id of the component(s) to be updated after async partial submit request
oncomplete	null	String	Javascript event to be called when ajax request for item select event is completed.

Name	Default	Type	Description
itemSelectListener	null	MethodExpression	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.
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.
wmode	null	String	wmode property of the flash object
widgetVar	null	String	Name of the client side widget

Getting started with Stacked Column Chart

Stacked column chart usage is very similar to line chart, as an example following stacked column chart displays the birth rate data in the lineChart example. Please see the lineChart section to get more information about the structure of the birth data.

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



3.8.5 Bar Chart

Bar Chart is the horizontal version of the column chart where columns are aligned on x axis as bars.

Info

Tag	barChart
Tag Class	org.primefaces.component.chart.bar.BarChartTag
Component Class	org.primefaces.component.chart.bar.BarChart
Component Type	org.primefaces.component.chart.BarChart
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.chart.BarChartRenderer
Renderer Class	org.primefaces.component.chart.bar.BarChartRenderer

Attributes

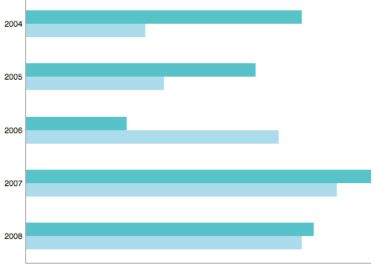
Name	Default	Type	Description
id	Assigned by JSF	String	Unique identifier of the component
rendered	TRUE	boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	java.util.List	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	Client side id of the component(s) to be updated after async partial submit request
oncomplete	null	String	Javascript event to be called when ajax request for item select event is completed.
itemSelectListener	null	MethodExpression	Method expression to listen chart series item select events

Name	Default	Type	Description
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.
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
widgetVar	null	String	Name of the client side widget

Getting started with Bar Chart

Bar chart usage is very similar to line chart, as an example following bar chart displays the birth rate data in the lineChart example. Important difference is that barchart uses yfield attribute instead of the xfield attribute. Please see the lineChart section to get more information about the structure of the birth data.

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



3.8.6 StackedBar Chart

Stacked Bar chart is similar to bar chart but the bar are stacked per each yField data.

Info

Tag	stackedBarChart
Tag Class	org.primefaces.component.chart.stackedbar.StackedBarChartTag
Component Class	org.primefaces.component.chart.stackedbar.StackedBarChart
Component Type	org.primefaces.component.chart.StackedBarChart
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.chart.StackedBarChartRenderer
Renderer Class	org.primefaces.component.chart.stackedbar.StackedBarChartRenderer

Attributes

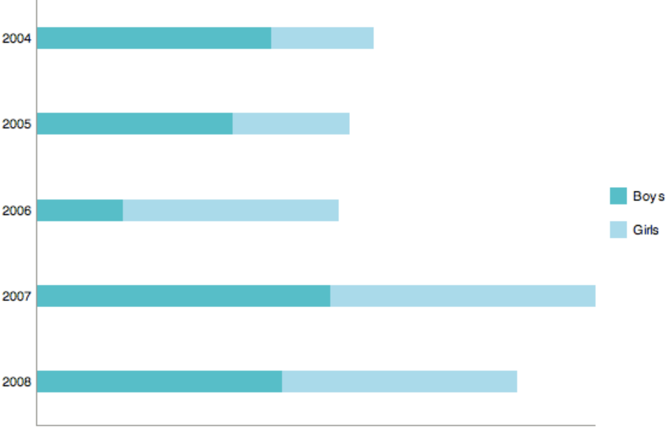
Name	Default	Type	Description
id	Assigned by JSF	String	Unique identifier of the component
rendered	TRUE	boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
value	null	java.util.List	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	Client side id of the component(s) to be updated after async partial submit request
oncomplete	null	String	Javascript event to be called when ajax request for item select event is completed.

Name	Default	Type	Description
itemSelectListener	null	MethodExpression	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.
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.
wmode	null	String	wmode property of the flash object
widgetVar	null	String	Name of the client side widget

Getting started with StackedBar Chart

StackedBar chart usage is very similar to line chart, as an example following stacked bar chart displays the birth rate data in the lineChart example. Important difference is that stackedbarchart uses yfield attribute instead of the xfield attribute. Please see the lineChart section to get more information about the structure of the birth data.

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



3.8.7 Chart Series

A chart can have one or more series and a chartSeries component represents each series in a chart.

Info

Tag	chartSeries
Tag Class	org.primefaces.component.chart.series.ChartSeriesTag
Component Class	org.primefaces.component.chart.series.ChartSeries
Component Type	org.primefaces.component.ChartSeries
Component Family	org.primefaces.component

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 to be displayed on the series
converter	null	Converter	Output converter to be used if any.
label	null	java.lang.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. Please see the other chart component documentations to see the usage of chartSeries.

3.8.8 Skinning Charts

Charts are highly customizable in terms of skinning however they are flash based, as a result 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 the javascript object variable name used in styling, 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

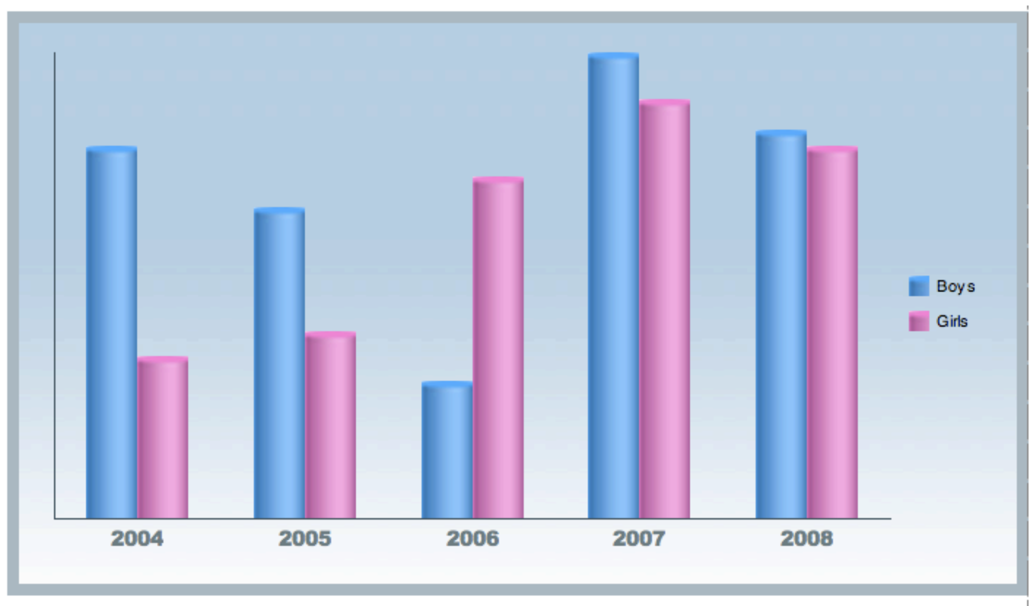
ChartSeries can be styled individually using the style attribute. Styling is same as charts and done via javascript.

```
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}
            }
      };
```

```
 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 Style Selectors

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

3.8.9 Real-Time Charts

Charts have built-in support for ajax polling and live data display. As an example suppose there’s an ongoing vote between two candidates. To start with, create 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 String getCandidate() {
            return candidate;
      }

      public void setCandidate(String candidate) {
            this.candidate = candidate;
      }

      public int getCount() {
            return count;
      }

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

      public void add(int count) {
            this.count = this.count + count;
      }
}
```

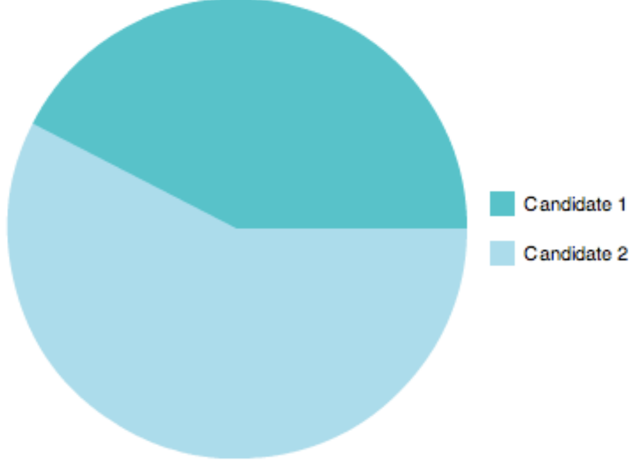
Next step is to provide the data;

```
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 continuosly in 5 seconds interval. Polling interval is tuned using the refreshInterval attribute which is set to 3000 milliseconds.



3.8.10 Interactive Charts

Charts are interactive components and they can respond to events like series item selection. When a series item is clicked an ajax request is sent to the server and an itemSelectListener is notified 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="msg"
                            categoryField="#{vote.candidate}"
                            dataField="#{vote.count}" />

<h:message value="#{chartBean.message}" />
```

```
public class ChartBean implements Serializable {

     //Data creation omitted

     public void itemSelect(ItemSelectEvent event) {
        message = "Item Index: " + event.getItemIndex() + ", Series Index:" +
             event.getSeriesIndex();
    }

}
```

Please note that interactive charts must be nested inside a form.

3.8.11 Charting FAQ

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.

JFreeChart Integration

If you like to use static image charts instead of flash based charts, see the JFreeChart integration example at graphicImage section.

3.9 Collector

Collector is a simple utility component to manage collections without writing java code on backing beans.

Info

Tag	collector
Tag Class	org.primefaces.component.collector.CollectorTag
ActionListener Class	org.primefaces.component.collector.Collector

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 overrive 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
}
```

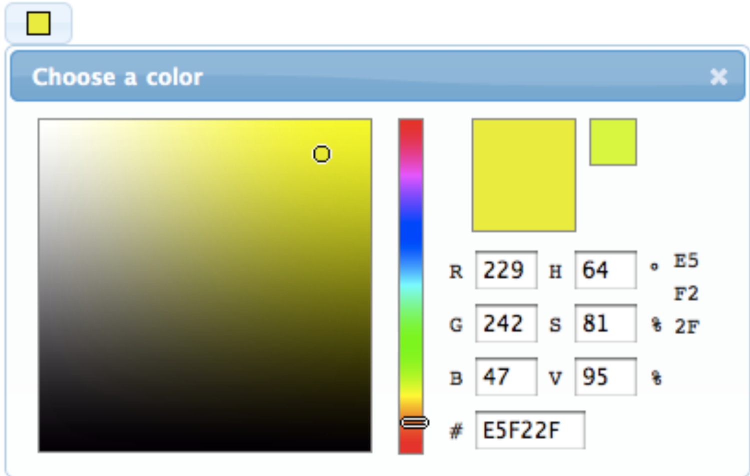
Value attribute is required and sets the object to be added or removed to/from a collection.

```
<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>
```

3.10 Color Picker

ColorPicker component enables color selection with visualization.



Info

Tag	colorPicker
Tag Class	org.primefaces.component.colorpicker.ColorPickerTag
Component Class	org.primefaces.component.colorpicker.ColorPicker
Component Type	org.primefaces.component.ColorPicker
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.ColorPickerRenderer
Renderer Class	org.primefaces.component.colorpicker.ColorPickerRenderer

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

Name	Default	Type	Description
value	null	java.util.Date	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	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	MethodExpression	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 as it's value by default.

```
import java.awt.Color;

public class ColorPickerController {

	private Color selectedColor;

	public Color getSelectedColor(){
		return selectedColor;
	}
	public void setSelectedColor(Color color){
		selectedColor = color;
	}
}
```

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

Converter

In case you don't prefer to use java.awt.Color, you can plug your custom converter.

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

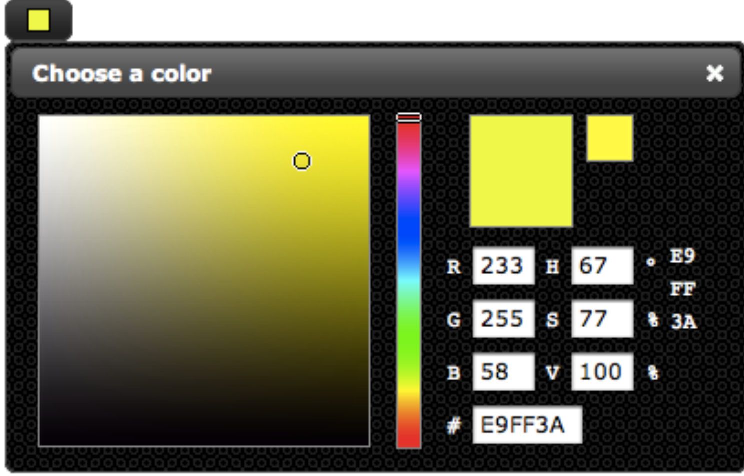
```
public class ColorPickerConverter implements Converter {

	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 seperated string in “R,G,B” format.
		return value == null ? null : value.toString();
	}
}
```

```
public class ColorPickerController {

	private String selectedColor;

	public String getSelectedColor(){
		return selectedColor;
	}
	public void setSelectedColor(String color){
		selectedColor = color;
	}
}
```

This way selected color will not be converted to java.awt.Color but used as a simple rgb string such as ‘250, 214, 255’.

Skinning

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



3.11 Column

Column is an extended version of the standard column providing features like sorting, selection, resizing, filtering and more.

Info

Tag	**column**
Tag Class	**org.primefaces.component.column.ColumnTag**
Component Class	**org.primefaces.component.column.Column**
Component Type	**org.primefaces.component.Column**
Component Family	**org.primefaces.component**

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
sortBy	null	Object	Property to be used when sorting this column.
sortFunction	null	String/ MethodExpression	Custom pluggable sortFunciton
resizable	FALSE	boolean	Boolean value to make the column width resizable
filterBy	FALSE	boolean	Specifies the data filter.
filterEvent	keyup	boolean	Event to trigger a filter request
filterStyle	null	String	Style of the filter component
filterStyleClass	null	String	Style class of the filter component
parser	null	String	Client side column parser.
width	null	Integer	Width in pixels.
styleClass	null	String	Style class of the column.

3.12 CommandButton

CommandButton extends standard JSF commandButton with ajax and skinning features.

Submit

Info

Tag	commandButton
Tag Class	org.primefaces.component.commandbutton.CommandButtonTag
Component Class	org.primefaces.component.commandbutton.CommandButton
Component Type	org.primefaces.component.CommandButton
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.CommandButtonRenderer
Renderer Class	org.primefaces.component.commandbutton.CommandButtonRenderer

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	String	Label for the button
action	null	javax.el.MethodExpression	A method expression that'd be processed when button is clicked.
actionListener	null	javax.faces.event.ActionListener	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.

Name	Default	Type	Description
type	submit	String	Sets the behavior of the button. Possible values are “submit” and “reset”.
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.
ajax	TRUE	Boolean	Specifies the submit mode, when set to true (default), submit would be made with Ajax.
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 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.
style	null	String	Style to be applied on the button element
styleClass	null	String	StyleClass to be applied on the button element
onblur	null	String	onblur dom event handler
onchange	null	String	onchange dom event handler
onclick	null	String	onclick dom event handler
ondblclick	null	String	ondblclick dom event handler
onfocus	null	String	onfocus 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

Name	Default	Type	Description
onmouseout	null	String	onmouseout dom event handler
onmouseover	null	String	onmouseover dom event handler
onmouseup	null	String	onmouseup dom event handler
onselect	null	String	onselect dom event handler
accesskey	null	String	Html accesskey attribute.
alt	null	String	Html alt attribute.
dir	null	String	Html dir attribute.
disabled	FALSE	Boolean	Disables the button.
image	null	String	Html image attribute.
label	null	String	Html label attribute.
lang	null	String	Html lang attribute.
tabindex	null	String	Html tabindex attribute.
title	null	String	Html title attribute.
readonly	FALSE	String	Html readonly attribute.

Getting started with CommandButton

CommandButton component submits it’s enclosed form with ajax by default.

```
public class BookBean {
   public String saveBook() {
      //button action is called
      return null;
   }
}
```

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

Reset Buttons

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

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

Push Buttons

Push buttons are used to execute custom javascript. 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 seperated 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 update with the ajax response.

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

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

Button Icons

An icon on a button is displayed 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;
}
```

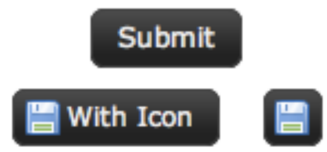
Skinning

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;



3.13 CommandLink

CommandLink extends standard JSF commandLink with Ajax capabilities.

Info

Tag	commandLink
Tag Class	org.primefaces.component.commandlink.CommandLinkTag
Component Class	org.primefaces.component.commandlink.CommandLink
Component Type	org.primefaces.component.CommandLink
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.CommandLinkRenderer
Renderer Class	org.primefaces.component. commandlink.CommandLinkRenderer

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	String	Href value of the rendered anchor.
action	null	MethodExpression	A method expression that'd be processed when button is clicked.
actionListener	null	ActionListener	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.
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.

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	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 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.
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	onblur dom event handler
onchange	null	String	onchange dom event handler
onclick	null	String	onclick dom event handler
ondblclick	null	String	ondblclick dom event handler
onfocus	null	String	onfocus 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
onselect	null	String	onselect dom event handler
accesskey	null	String	HTML accesskey attribute

Name	Default	Type	Description
charset	null	String	HTML charset attribute
coords	null	String	HTML coords attribute
dir	null	String	HTML dir attribute
disabled	null	Boolean	Disables the link
hreflang	null	String	HTML hreflang attribute
rel	null	String	HTML rel attribute
rev	null	String	HTML rev attribute
shape	null	String	HTML shape attribute
tabindex	null	String	HTML tabindex attribute
target	null	String	HTML target attribute
title	null	String	HTML title attribute
type	null	String	HTML type attribute

Getting started with commandLink

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

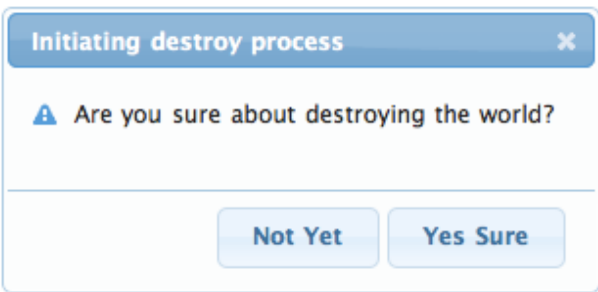
```
<h:form>
     <p:commandLink actionListener=”#{bean.action}” update=”text”>
            <h:outputText value=”Ajax Submit” />
     </p:commandLink>

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

Skinning

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

3.14 ConfirmDialog

ConfirmDialog is a replacement to the legacy javascript confirmation box. Its main use is to have the user do a decision. Skinning, customization and avoiding popup blockers are notabled advantages over classic javascript confirmation.



Info

Tag	confirmDialog
Tag Class	org.primefaces.component.confirmdialog.ConfirmDialogTag
Component Class	org.primefaces.component.confirmdialog.ConfirmDialog
Component Type	org.primefaces.component.ConfirmDialog
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.ConfirmDialogRenderer
Renderer Class	org.primefaces.component.confirmdialog.ConfirmDialogRenderer

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
widgetVar	null	String	Variable name of the client side widget.
message	null	String	Text to be displayed in body.
header	null	String	Text for the header.
severity	null	String	Message severity for the dislayed icon.

Name	Default	Type	Description
draggable	TRUE	Boolean	Controls draggability
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

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>
```

Effects

There are various effect options to be used when displaying and closing the dialog.

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

Severity

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

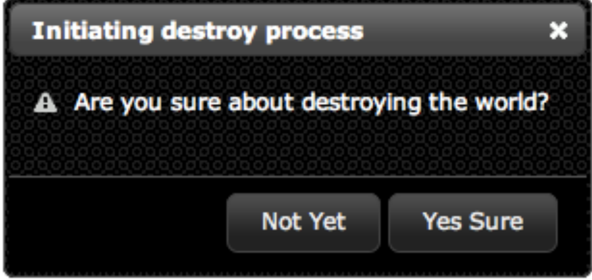
Skinning

ConfirmDialog resides in a main container element which the *styleClass* option apply.

Following is the list of structural style classes;

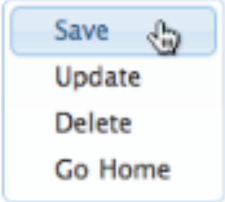
Style Class	Applies
.ui-dialog	Container element of dialog
.ui-dialog-titlebar	Title bar
.ui-dialog-title-dialog	Header text
.ui-dialog-titlebar-close	Close icon
.ui-dialog-content	Dialog body
.ui-dialog-buttonpane	Footer button panel

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



3.15 ContextMenu

ContextMenu provides a popup menu that is displayed on mouse right-click event.



Info

Tag	contextMenu
Tag Class	org.primefaces.component.contextmenu.ContextMenuTag
Component Class	org.primefaces.component.contextmenu.ContextMenu
Component Type	org.primefaces.component.ContextMenu
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.ContextMenuRenderer
Renderer Class	org.primefaces.component.contextmenu.ContextMenuRenderer

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
widgetVar	null	String	Variable name of the client side widget
for	null	String	Id of the component to attach to
style	null	String	Style of the main container element
styleClass	null	String	Style class of the main container element

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
```

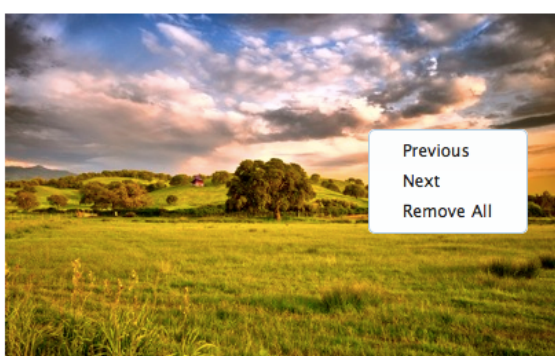
ContextMenu example above is attached to the whole page and consists of three different menuitems with different use cases. First menuitem triggers an ajax action, second one triggers a non-ajax action and third one is used for navigation.

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;



Skinning

ContextMenu resides in a main container element which the *style* and *styleClass* option apply.

Following is the list of structural style classes;

Style Class	Applies
.ui-menu	Container element of menu
.ui-menu-item	Each menu item
.ui-menu-item-label	Each menu item label

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

3.16 Dashboard

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

Info

Tag	dashboard
Tag Class	org.primefaces.component.dashboard.DashboardTag
Component Class	org.primefaces.component.dashboard.Dashboard
Component Type	org.primefaces.component.Dashboard
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.DashboardRenderer
Renderer Class	org.primefaces.component.dashboard.DashboardRenderer

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
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.

Name	Default	Type	Description
reorderListener	null	MethodExpr ession	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 applications 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.

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 structual 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
vod 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 identifed 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

3.17 DataExporter

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

Info

Tag	**dataExporter**
Tag Class	**org.primefaces.component.export.DataExporterTag**
ActionListener Class	**org.primefaces.component.export.DataExporter**

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 seperated 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	MethodExpression	PreProcessor for the exported document.
postProcessor	null	MethodExpression	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.

Excel export

```
<h:commandButton value=”Export as Excel”>
     <p:dataExporter type=”xls” target=”tableId” fileName=”cars”/>
</h:commandButton>
```

PDF export

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

CSV export

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

XML export

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

PageOnly

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

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

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 defined 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 seperated string (excludeColumns="0,2,6")

```
<h:commandButton value="Export as Excel">
    <opt:exportActionListener type="xls" target="tableId" fileName="cars"
    excludeColumns="0"/>
</h:commandButton>
```

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);
    HSSFCellStyle cellStyle = wb.createCellStyle();
    cellStyle.setFillForegroundColor(HSSFColor.GREEN.index);
    cellStyle.setFillPattern(HSSFCellStyle.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=”xls” 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));
}
```

3.18 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	dataGrid
Tag Class	org.primefaces.component.datagrid.DataGridTag
Component Class	org.primefaces.component.datagrid.DataGrid
Component Type	org.primefaces.component.DataGrid
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.DataGridRenderer
Renderer Class	org.primefaces.component.datagrid.DataGridRenderer

Attributes

Name	Default	Type	Description
id	Assigne d 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	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 CarListController() {
        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" style="width:100%">
                <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;

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

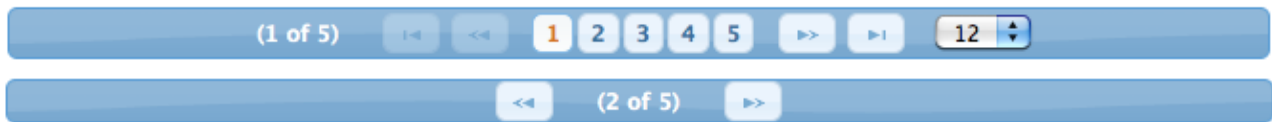
Note that {RowsPerPageDropDown} has it’s own template, options to display is provided via rowsPerPageTemplate attribute.

Default UI is;

which corresponds to the following template.

“{FirstPageLink} {PreviousPageLink} {PageLinks} {NextPageLink} {LastPageLink}”

Here are more examples based on different templates;



Paginator Position

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

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. Online showcase example of datagrid has an example that demonstrates a simple case of selecting a data with an ajax commandLink and displaying the details with a dialog. Important point is to place datagrid contents in a p:column which is a child of datagrid.

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>
```

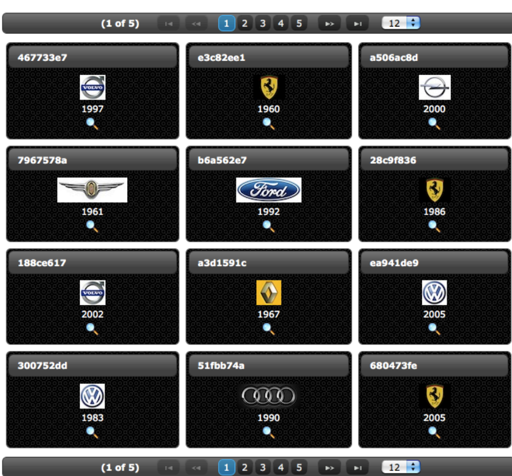
Skinning

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

Following is the list of structural style classes;

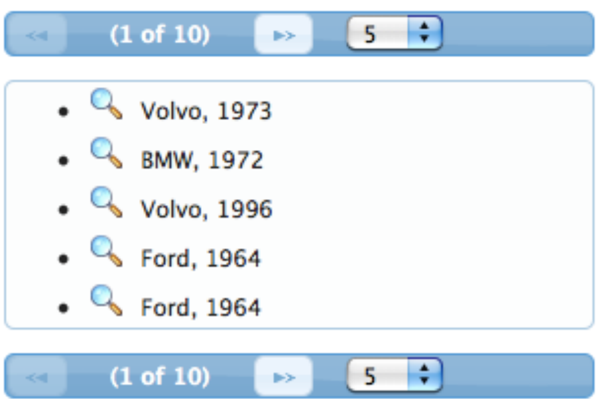
Class	Applies
.ui-datagrid	Main container element
.ui-datagrid-data	Table element containing data
.ui-datagrid-row	A row in grid
.ui-datagrid-data	A column in grid

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



3.19 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	**dataList**
Tag Class	**org.primefaces.component.datalist.DataListTag**
Component Class	**org.primefaces.component.datalist.DataList**
Component Type	**org.primefaces.component.DataList.DataListTag**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.DataListRenderer**
Renderer Class	**org.primefaces.component.datalist.DataListRenderer**

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	Data to display.

Name	Default	Type	Description
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.
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 a list of cars to display throughout the datalist 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 CarListController() {
		cars = new ArrayList<Car>();
		cars.add(new Car("myModel",2005,"ManufacturerX","blue"));
		//add more cars
	}

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

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

Since DataList is a data iteration component, it renders it's children for each data represented with var option. ItemType defines the bullet type of each item, some examples for an unordered list are disc, circle and square;

Disc	Circle	Square
• Opel, 1980 • Chrysler, 1966 • Volvo, 1962 • Audi, 1990 • Ford, 1972 • Mercedes, 2003 • BMW, 1984 • Audi, 1975 • Volvo, 1973	○ Opel, 1980 ○ Chrysler, 1966 ○ Volvo, 1962 ○ Audi, 1990 ○ Ford, 1972 ○ Mercedes, 2003 ○ BMW, 1984 ○ Audi, 1975 ○ Volvo, 1973	▪ Opel, 1980 ▪ Chrysler, 1966 ▪ Volvo, 1962 ▪ Audi, 1990 ▪ Ford, 1972 ▪ Mercedes, 2003 ▪ BMW, 1984 ▪ Audi, 1975 ▪ Volvo, 1973

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 as follows;

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

For ordered lists, various item types are available as well;

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

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>
```

Paginator Template

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

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

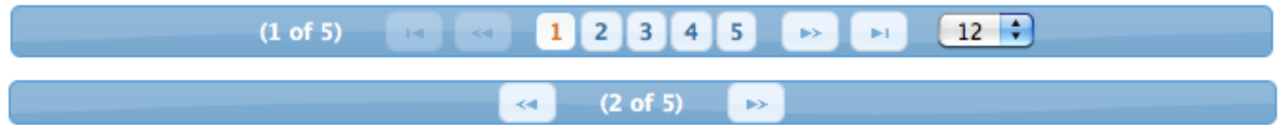
Note that {RowsPerPageDropDown} has it’s own template, options to display is provided via rowsPerPageTemplate attribute.

Default UI is;

which corresponds to the following template.

“{FirstPageLink} {PreviousPageLink} {PageLinks} {NextPageLink} {LastPageLink}”

Here are more examples based on different templates;



Paginator Position

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

Selecting Data

Selection of data displayed in datalist is very similar to row selection in datatable, you can access the current data using the var reference. Online showcase example of datalist has an example that demonstrates a simple case of selecting a data with an ajax commandLink and displaying the details with a dialog. Important point is to place datalist contents in a p:column which is a child of dataList.

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

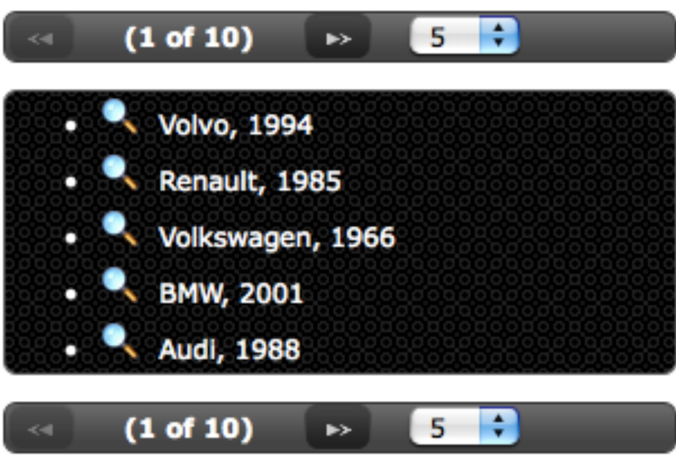
Skinning

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-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;



3.20 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.

Info

Tag	dataTable
Tag Class	org.primefaces.component.datatable.DataTableTag
Component Class	org.primefaces.component.datatable.DataTable
Component Type	org.primefaces.component.DataTable
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.DataTableRenderer
Renderer Class	org.primefaces.component.datatable.DataTableRenderer

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.

Name	Default	Type	Description
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
var	null	String	Name of the request-scoped variable that'll be used as the holder of each rowdata before processing a row
rows	null	int	Number of rows to display per page
first	0	int	Index of the first row to be displayed
widgetVar	null	String	Name of the client side widget
paginator	FALSE	Boolean	Sets paginator
paginatorTemplate	null	String	Template for the paginator layout, default value is “{FirstPageLink} {PreviousPageLink} {PageLinks} {NextPageLink} {LastPageLink}”
rowsPerPageTemplate	null	String	Template for the rowsPerPage dropdown.
currentPageReportTemplate	null	String	Template of the currentPageReport UI.
firstPageLinkLabel	null	String	Label of the first link in paginator
previousPageLinkLabel	null	String	Label of the previous link in paginator
nextPageLinkLabel	null	String	Label of the next link in paginator
lastPageLinkLabel	null	String	Label of the last link in paginator
selection	null	Object	An object to populate the selected row data.
selectionMode	null	String	Type of data selection, valid values are ‘multiple’, ‘single’, ‘cellblock’, ‘cellrange’ and ‘singlecell’.
dynamic	FALSE	boolean	Specifies sorting/paging mode, when set to true sorting and paging is handled with ajax.
lazy	FALSE	boolean	Enables lazy loading feature.
rowIndexVar	null	String	Variable name referring to the rowIndex being processed.
paginatorPosition	both	String	Position of paginator, valid values are ‘both’, ‘top’ or ‘bottom’.

Name	Default	Type	Description
emptyMessage	null	String	Message to be shown when there're records to display.
errorMessage	null	String	Message to be shown when an error occurs during data loading.
loadingMessage	null	String	Message to be shown when loading data with ajax.
sortAscMessage	null	String	Tooltip to be shown to sort a column data in ascending order.
sortDescMessage	null	String	Tooltip to be shown to sort a column data in descending order.
update	null	String	Client side id of the component(s) to be updated after ajax row selection.
style	null	String	Style of the main container element of table.
styleClass	null	String	Style class of the main container element of table.
onselectStart	null	String	Javascript event handler to be called before ajax request for instant ajax row selection request begins.
onselectComplete	null	String	Javascript event handler to be called after ajax request for instant ajax row selection request is completed.
dblClickSelect	FALSE	boolean	Enabled row selection on double click instead of single click(default)
page	1	Integer	Index of the current page, first page is 1.
pageLinks	10	Integer	Number of page links to display

Getting started with the DataTable

We will be using a list of cars to display throughout the datatable 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 datatable to the car list.

```
public class CarBean {

    private List<Car> cars;

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

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

```
<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>
```

Dynamic vs Non-Dynamic

DataTable has two main modes, when it is non-dynamic(default) it works as a pure client side component, on the other hand dynamic datatables fetch their data from backing bean model with ajax. Features including paging, sorting and filtering are both implemented on client side and server side to handle both cases of dynamic seting. For small datasets non-dynamic datatables is much faster and have the advantage of avoiding roundtrips to server with 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>
```

Paginator Template

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

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

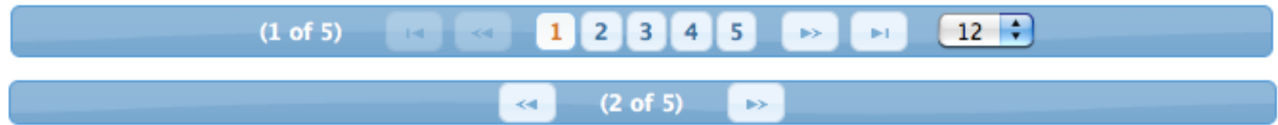
Note that {RowsPerPageDropDown} has it’s own template, options to display is provided via rowsPerPageTemplate attribute.

Default UI is;

which corresponds to the following template.

“{FirstPageLink} {PreviousPageLink} {PageLinks} {NextPageLink} {LastPageLink}”

Here are more examples based on different templates;



Paginator Position

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

Sorting

Sorting is controlled at column level, defining sortBy attribute enables sorting on that particular column. If datatable is dynamic, sorting is handled on server side with ajax, if not sorting happens on client side.

```
<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>
```

When sorting is handled on client side, you need to define built-in client side data parsers for proper sorting. Possible values for parser attribute of column are ‘string’(default), ‘number’ and ‘date’.

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

     <p:column sortBy="#{car.year}" parser="number">
          <f:facet name="header">
               <h:outputText value="Year" />
          </f:facet>
          <h:outputText value="#{car.year}" />
     </p:column>

     ...more columns

</p:dataTable>
```

Custom Sorting

Instead of using the default sorting algorithm, you can plug-in your own sort function.

When ajax sorting enabled sorting must refer to a java method that takes two parameters and return an integer value.

```
<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
}
```

In case datatable is not dynamic then sortFunction must refer to a javascript function.

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

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

    ...more columns

</p:dataTable>
```

```
<script type="text/javascript">
sortByYear = function(a, b, desc, field) {
    var val1 = a.getData(field);
    var val2 = b.getData(field);

    //return an integer
}
</script>
```

Client side javascript function gets four parameters;

Parameter	Description
a	Javascript object representing a row data
b	Javascript object representing a row data
desc	Boolean value of order, returns true if order is descending
field	Column key to be used to retrieve column value

Row Selection

There are several built-in solutions that make row selection a piece of cake. One way is to directly click on table rows and second way is to use a selection column.

Single Row Selection

To select a single row when a row is clicked use the selectionMode attribute of the datatable.

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

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

selectedCar is a simple member of type Car that will be set with the selected data once the form is submitted. Note that when a row is clicked, row is highlighted.

Additionaly if you'd like allow row selection with double clicking to a row instead of single click set dblClickSelect option to true.

```
public class CarBean {

     private List<Car> cars;

     private Car selectedCar;

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

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

     public Car getSelectedCar() {
           return selectedCar;
     }

     public void setSelectedCar(Car selectedCar) {
           this.selectedCar = selectedCar;
     }
}
```

Model	Year	Manufacturer	Color
8c875f4a	2004	Chrysler	Brown
c4a45109	1974	BMW	Silver
33fa48a5	1975	Volvo	Blue
ee239b6d	1996	Audi	Maroon
65c662f4	1994	Chrysler	Blue
236412bb	2000	Ferrari	White
9ee96f78	1963	Volvo	Silver
647e5cbe	2001	Ford	Silver
c020c63c	1974	Ford	Black
5d7a3123	1983	Audi	Blue

Multiple Row Selection

If you require selecting multiple rows, set the selectionMode to multiple and define a Car array. This way using modifier keys like ctrl or shift, multiple rows can be selected.

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

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

```
public class CarBean {

     private List<Car> cars;

     private Car[] selectedCars;

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

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

     public Car[] getSelectedCars() {
          return selectedCars;
     }

     public void setSelectedCar(Car[] selectedCar) {
          this.selectedCars = selectedCars;
     }
}
```

Model	Year	Manufacturer	Color
b6a283f0	1965	BMW	Maroon
f99f119c	1979	Chrysler	Green
c812f494	1974	Audi	Orange
3793e6f5	1975	Ferrari	Brown
189a5d38	1965	BMW	Brown
e15a8609	1972	Chrysler	Yellow
4fe12042	1973	Volvo	Brown
3f47a9aa	2004	Renault	Green
706f1533	2001	Mercedes	Black
2ad40bcd	1977	Renault	Silver

Instant Ajax Row Selection

Two methods describe above requires the form to be submitted before the row selection can happen. If you need instant row selection with ajax define an update attribute pointing to the component to be updated. This way when a row is clicked ajax request is triggered and selected row(s) are assigned to the selection model instantly without a need for an implicit form submit.

```
<p:dataTable var="car" value="#{carListController.cars}"
     selection="#{carListController.selectedCars}" selectionMode="single"
     update="display" onselectComplete="dialog.show()">

     ...columns

</p:dataTable>

<p:dialog widgetVar="dialog">
  <p:outputPanel id="display">
     <ui:repeat value="#{carListController.selectedCars}" var="selectedCar">
           <h:outputText value="#{selectedCar.model}" />
     </ui:repeat>
  </p:dialog>
</p:outputPanel>
```

When a row is selected on the datatable above, ajax request updates the display panel and shows a dialog with the selected cars information. Callbacks like onselectStart and onselectComplete allows creating flexible UIs. This is quite useful if you need to display detailed information about selected data instantly.

Cell Selection

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

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

     ...columns

</p:dataTable>
```

```
public class CarBean {

     private List<Car> cars;

     private Cell selectedCell;

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

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

     public Cell getSelectedCell() {
          return selectedCell;
     }
}
```

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	Client id of the cell column
value	Object	Value displayed in cell.

For multiple cell selection use “cellblock” and “cellrange” selection mode options, in this case selection should be a Cell[] reference instead of a single Cell.

Data Filtering

Setting a column filter is enabled by using *filterBy* option..

```
<p:dataTable var="car" value="#{carListController.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>
```

Model	Year	Manufacturer	Color
1fc20225	1988	Ferrari	Blue
548d3adf	2000	Renault	Red
62fc760d	1966	Opel	Blue
02248c66	1974	Ford	White
a57dbcb7	1984	Audi	Silver
39a96cf2	1999	Volkswagen	Black
4485acb3	1960	BMW	Orange
1724f9b4	2008	Mercedes	Orange
83445f95	1965	Volkswagen	Silver
c4dca202	1974	Ford	White

Similar to paging and sorting, dynamic datatables use ajax to filter data whereas non-dynamic datatables handle filtering on client side.

By default filtering is triggered with keyup event, this is configurable using *filterEvent* attribute, in addition filter inputs can be styled using *filterStyle* and *filterStyleClass* attributes.

Lazy Loading

Dealing with huge sets of data like thousand and even millions is not a trivial task, good news is datatable provides a built-in feature that can even handle billions of data in an efficient way. The idea behind lazy loading is to load only the rows of the datatable page being displayed.

In order to enable lazy loading you just need to set lazy attribute to true and provide a LazyDataModel as the value of the datatable.

```
<p:dataTable var=”car” value=”#{carListController.lazyModel}”
     dynamic=”true” lazy=”true”>
     //columns
</p:dataTable>
```

```
public class CarListController {

     private LazyDataModel<Car> lazyModel;

     public CarListController() {
     /**
     * Test with one hundred million records.
     * In a real application use a count query to get the rowcount.
     */
     lazyModel = new LazyDataModel<Car>(100000000) {

     /**
     * Dummy implementation of loading a certain segment of data.
     * In a real applicaiton, this method should access db and do a limit
     * based query
      */
     @Override
     public List<Car> fetchLazyData(int first, int pageSize) {
           //Query a list of cars starting with offset first and max size
           //pagesize
           }
       };
     }


     public LazyDataModel getLazyModel() {
           return lazyModel;
     }
}
```

When lazy loading is enabled, datamodel will executed your LazyModel’s fetchLazyData method with the first and pagesize variables. It’s your responsibility to load a chunk of data that starts from the first offset and with size pageSize.

For example if you’re using jpa you can use setFirstResult(first) and setMaxResults (pageSize) api to load a certain amount of data. With lazy loading you never load the whole dataset but only the necessary portion. LazyLoading feature is also enhanced with ajax paging for rich user experience.

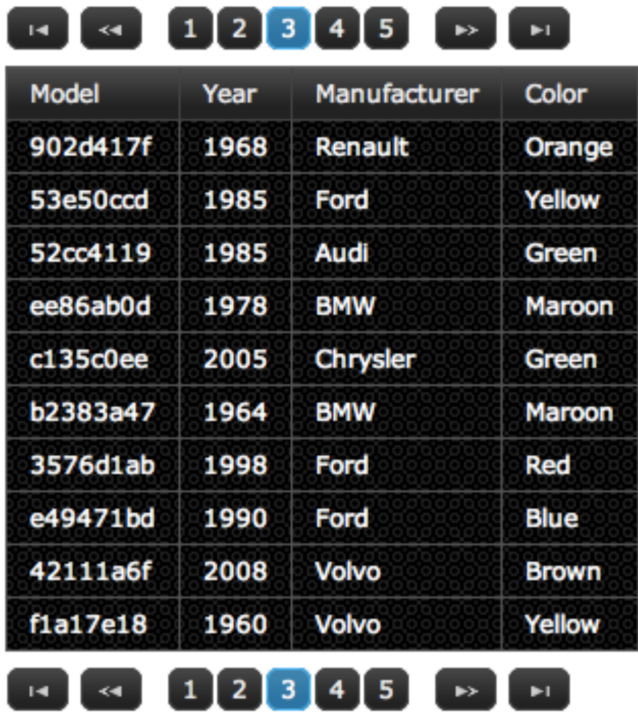
Skinning

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	Data container
.ui-datatable .ui-datatable-data td	Each cell in datatable
.ui-datatable-liner	Content displayed in datatable cells

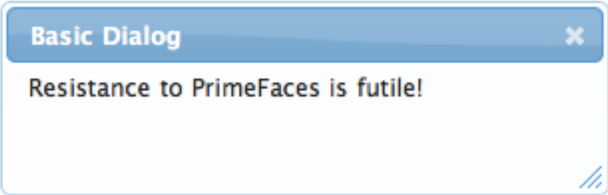
As skinning style classes are global, see the main Skinning section for more information. Here is an example based on a different theme;



Model	Year	Manufacturer	Color
902d417f	1968	Renault	Orange
53e50ccd	1985	Ford	Yellow
52cc4119	1985	Audi	Green
ee86ab0d	1978	BMW	Maroon
c135c0ee	2005	Chrysler	Green
b2383a47	1964	BMW	Maroon
3576d1ab	1998	Ford	Red
e49471bd	1990	Ford	Blue
42111a6f	2008	Volvo	Brown
f1a17e18	1960	Volvo	Yellow

3.21 Dialog

Dialog is a panel component that overlays other elements. Dialog avoids popup blockers, provides customization, resizing and ajax interactions.



Info

Tag	**dialog**
Tag Class	**org.primefaces.component.dialog.DialogTag**
Component Class	**org.primefaces.component.dialog.Dialog**
Component Type	**org.primefaces.component.Dialog**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.DialogRenderer**
Renderer Class	**org.primefaces.component.dialog.DialogRenderer**

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
header	null	String	Text of the header
draggable	TRUE	Boolean	Specifies draggability
resizable	TRUE	Boolean	Specifies resizability

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.
visible	FALSE	Boolean	Set's dialogs visibility
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	MethodExpression	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
widgetVar	null	String	Name of the client side widget

Getting started with the Dialog

Since dialog is a container component it needs children components to display. 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 via widgetVar.

- show() : Displays the dialog
- hide() : Makes dialog invisible

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

<a href="#" onclick="dialog.show()">Show</a>
<a href="#" onclick="dialog.hide()">Hide</a>
```

Positioning

Dialog can be positioned anywhere on screen, default position is the center of the window. [x,y] based absolute positioning is defined using a comma seperated value.

```
<p:dialog header="Header Text" widgetVar="dialog" position="10,50">
    //Content
</p:dialog>
```

Ajax Interaction

A dialog can also be used for ajax interaction. Following example demonstrates an example powered by PrimeFaces PPR.

```
<h:outputText id="name" value="Name #{pprBean.firstname}"/>

<h:outputLink value="#" onclick="dlg.show()">Enter FirstName</h:outputLink>

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

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

When the dialog is shown, it displays a form to enter the firstname, once Ajax Submit button is clicked, dialog is hidden and outputText with id=”name” is partially updated.

Ajax CloseListener and onCloseUpdate

A server side *closeListener* can be be invoked by passing an *org.primefaces.event.CloseEvent as a* parameter when the dialog is closed. Optionally components to update after dialog is closed can be defined with onCloseUpdate attribute. 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
     }
}
```

Effects

There are various effect options to be used when displaying(*showEffect*) and closing (*hideEffect*) the dialog.

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

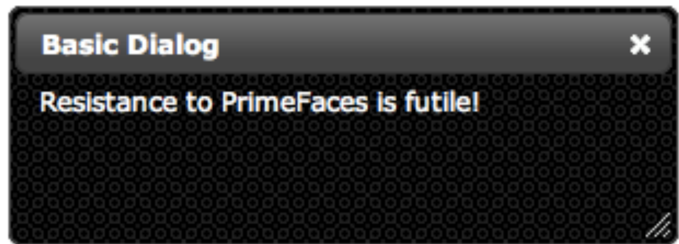
Skinning

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. Here is an example based on a different theme;



3.22 Drag&Drop

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

Draggable

Info

Tag	**draggable**
Tag Class	**org.primefaces.component.dnd.DraggableTag**
Component Class	**org.primefaces.component.dnd.Draggable**
Component Type	**org.primefaces.component.Draggable**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.DraggableRenderer**
Renderer Class	**org.primefaces.component.dnd.DraggableRenderer**

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
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

Name	Default	Type	Description
snapMode	null	String	Specifies the snap mode. Valid values are ‘both’, ‘inner’ and ‘outer’.
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 p: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 it's 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.pf-panel-hd"/>
```

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 seperated 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.

```
<p:outputPanel layout="block" style="width:400px;height:200px;">
     <p:panel id="conpnl" header="Restricted">
        <h:outputText value="I am restricted to my parent's boundaries" />
     </p:panel>
</p:outputPanel>

<p:draggable for="conpnl" containment="parent" />
```

Droppable

Info

Tag	droppable
Tag Class	org.primefaces.component.dnd.DroppableTag
Component Class	org.primefaces.component.dnd.Droppable
Component Type	org.primefaces.component.Droppable
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.DroppableRenderer
Renderer Class	org.primefaces.component.dnd.DroppableRenderer

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
widgetVar	null	String	Variable name of the client side widget
dropListener	null	javax.el.MethodExpression	A server side listener to process a DragDrop event.
for	null	String	Id of the component to add droppable behavior
disabled	FALSE	Boolean	Disables of 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 dropped.
onDropUpdate	null	String	Component(s) to update with ajax after a draggable is dropped.
onDrop	null	String	Javascript event handler to execute when a draggable is dropped.

Name	Default	Type	Description
accept	null	String	Selector to define the accepted draggables.
scope	null	String	Scope key to match draggables and droppables.
tolerance	null	String	Specifies the intersection mode to accept a draggable.

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:outputPanel>

<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 the droppable.

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

Drop Listener

A dropListener is a simple java method that’s executed when a draggable item is dropped onto a droppable component. A DragDrop event is passed as a parameter holding information about the dragged and dropped components. Using the previous example just add a dropListener to the droppable.

```
<p:graphicImage id=”messi” value=”barca/messi_thumb.jpg” />
<p:draggable for=”messi”/>

<p:outputPanel id=”zone” styleClass=”slot” />
<p:droppable for=”zone” dropListener=”#{ddController.onDrop}”/>
```

```
public void onDrop(DragDropEvent ddEvent) {
     logger.info(”Dragged Id: {}”, ddEvent.getDragId());
     logger.info(”Droppped Id: {}”, ddEvent.getDropId());
}
```

The method above just logs the item being dragged and dropped. Output of this method would be;

```
Dragged Id: messi
Dropped Id: zone
```

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. Also on the client side you can use onDrop callback to execute custom javascript.

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 handy to change the style of the droppable when the state changes.

3.23 Dock

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

Info

Tag	**dock**
Tag Class	**org.primefaces.component.dock.DockTag**
Component Class	**org.primefaces.component.dock.Dock**
Component Type	**org.primefaces.component.Dock**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.DockRenderer**
Renderer Class	**org.primefaces.component.dock.DockRenderer**

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
position	bottom	String	Position of the dock, *bottom* or *top*.
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

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

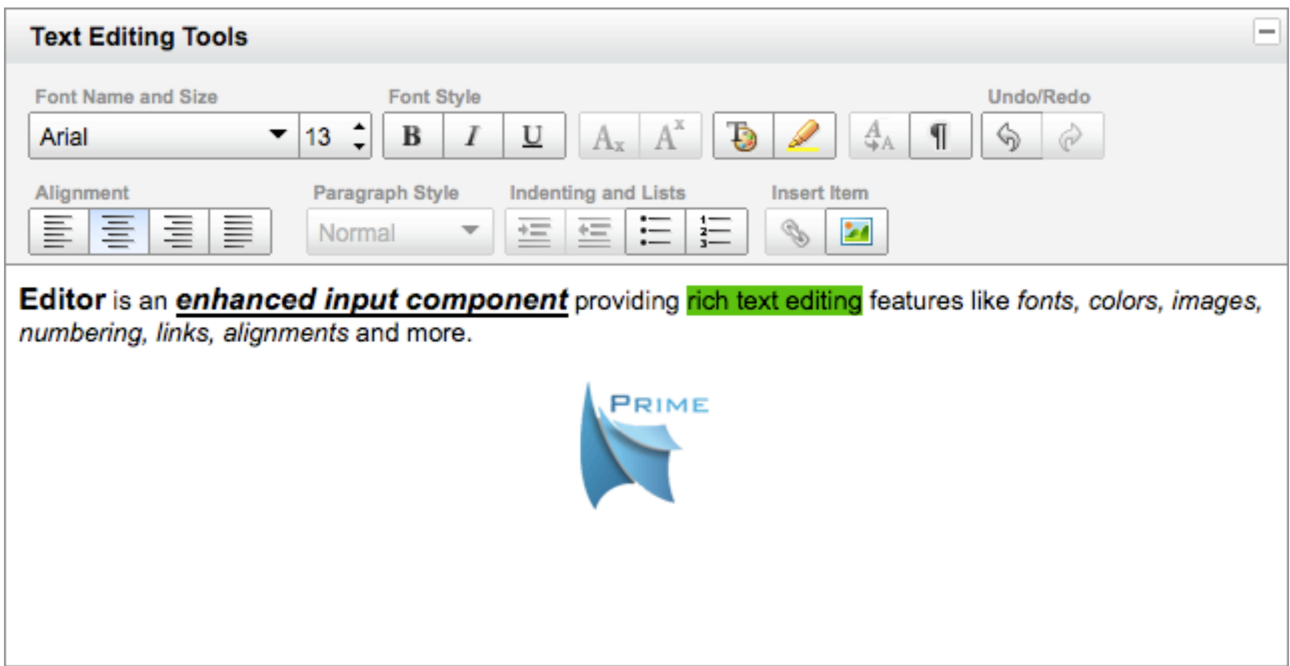
When mouse is over the dock items, icons enlarge. The configuration of this effect is done via the maxWidth and proximity attributes.

Dynamic Menus

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

3.24 Editor

Editor is an enhanced input component providing rich text editing features. Editor supports advanced text editing features like fonts, colors, images, alignment and more.



Info

Tag	editor
Tag Class	org.primefaces.component.editor.EditorTag
Component Class	org.primefaces.component.editor.Editor
Component Type	org.primefaces.component.Editor
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.EditorRenderer
Renderer Class	org.primefaces.component.editor.EditorRenderer

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.

Name	Default	Type	Description
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	MethodExpression	A method binding expression that refers to a method validationg the input
valueChangeListener	null	ValueChangeListe ner	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.
width	500px	String	Width of the editor
height	300px	String	Height of the editor
widgetVar	null	String	Javascript variable name of the wrapped widget
resizable	FALSE	Boolean	Makes editor resizable when set to true

Name	Default	Type	Description
language	null	String	Language of editor labels, default is en
title	null	String	Title text of editor
disabled	FALSE	Boolean	Disabled editing.

Getting started with the Editor

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

```
public class MyController {
      private String text;

      public String getText() {
            return text;
      }
      public void setText(String text) {
            this.text = text
      }
}
```

```
<p:editor value=”#{myController.text}” />
```

Editor and I18N

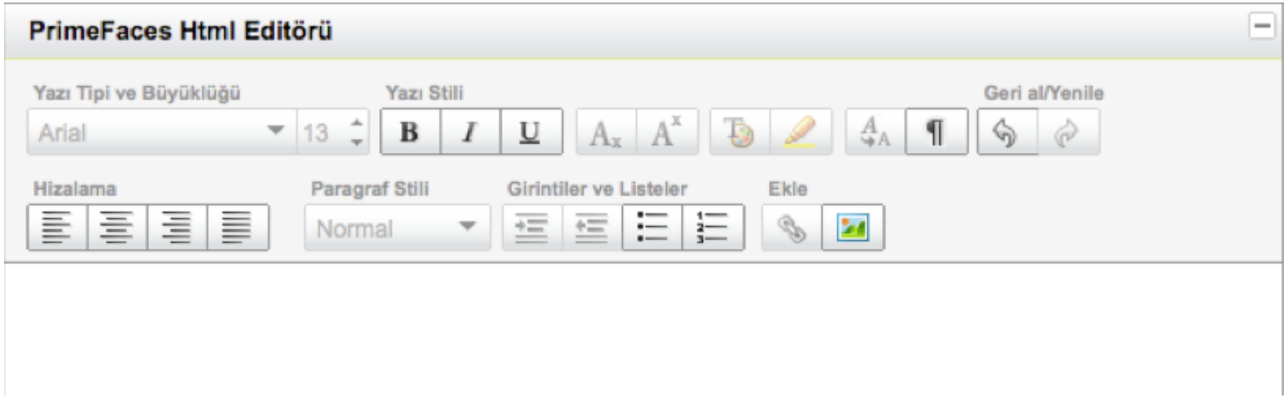
Labels like Font Style, Paragraph Style can be localized using the language attribute. Default language is English and following languages are supported out of the box.

- *“tr” : Turkish*
- *“pt” : Portuguese*

Please contact PrimeFaces team using support forum if you’re willing to provide a new translation.

Header text of the editor can also be changed via the title attribute. Following is a Turkish editor.

```
<p:editor value=”#{myController.text}” title=”PrimeFaces Html Editörü”
          language=”tr”/>
```



Ajax and Editor

Ajax is a special case for editor, when you submit the form with ajax editor value will not be passed to the value that is bound because editor can't attach it's value to the form automatically in ajax case. Solution is to do it manually;

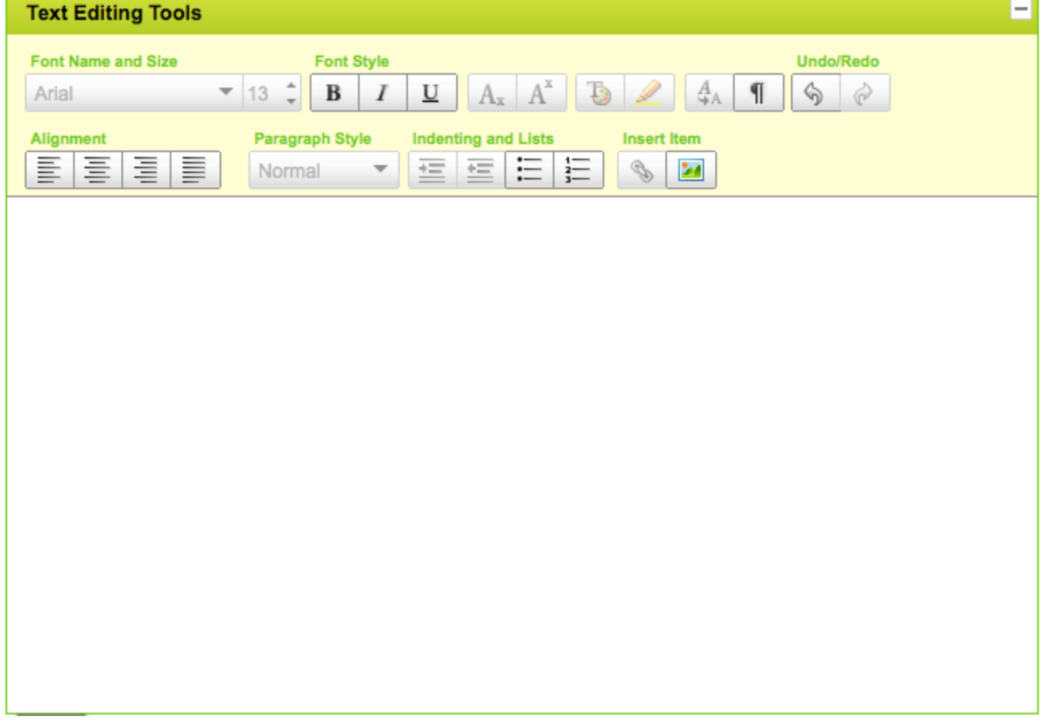
```
<h:form>
	<p:editor value=”#{myController.text}” widgetVar=”editor” />
	<p:commandButton value=”Submit” onclick=”editor.saveHTML()” />
</h:form>
```

When you are not using ajax, this won't be necessary as editor attaches saveHTML() called to the form's submit handler automatically behind the scenes. So following non-ajax case would work;

```
<h:form>
	<p:editor value=”#{myController.text}” />
	<p:commandButton value=”Submit” ajax=”false” />
	<h:commandButton value=”Submit” />
</h:form>
```

Skinning Editor

```
.yui-skin-sam .yui-editor-container {
            border-color: #33CC00;
}

.yui-skin-sam .yui-toolbar-container .yui-toolbar-titlebar h2 {
      background: url(../design/nav.gif);
}

.yui-skin-sam .yui-toolbar-container {
    background-color: #FFFFCC ;
}

.yui-skin-sam .yui-toolbar-container .yui-toolbar-group h3 {
    color: #33CC00 ;
}
```



Full list of CSS Selectors is available at;

http://developer.yahoo.com/yui/examples/editor/skinning_editor.html

3.25 Effect

Effect component is based on the jQuery effects library.

Info

Tag	effect
Tag Class	org.primefaces.component.effect.EffectTag
Component Class	org.primefaces.component.effect.Effect
Component Type	org.primefaces.component.Effect
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.EffectRenderer
Renderer Class	org.primefaces.component.effect.EffectRenderer

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
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	int	Speed of the animation in ms

Getting started with Effect

Effect component needs to be nested inside another component. If for attribute is not provided, by default parent would be animated.

```
<h:outputText value=”#{bean.value}”>
    <p:effect type=”pulsate” event=”click” />
</h:outputText>
```

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

Animation Target

By default, effect is attached to it's parent on the specified event. There may be cases where you want to display an effect on another component on the same page while keeping the parent as the trigger. For attribute is added for this purpose.

```
<h:outputLink id="lnk" value="#">
    <h:outputText value="Show the Barca Temple" />
    <p:effect type="appear" event="click" for="img" />
</h:outputLink>

<h: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.

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"/>
```

3.26 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	fileDownload
Tag Class	org.primefaces.component.filedownload.FileDownloadTag
ActionListener Class	org.primefaces.component.filedownload.FileDownloadActionListener

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 ready 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 FileDownloadController {

   private StreamedContent file;

   public FileDownloadController() {
       InputStream stream = this.getClass().getResourceAsStream(”file.pdf”);
       file = new DefaultStreamedContent(stream, “application/pdf”,
                       “downloaded_file.pdf”);
     }
     //getters and setters
}
```

This streamed content should be bound to the value of the fileDownload.

```
<h:commandButton value="Download">
      <p:fileDownload value="#{fileDownloadController.file}" />
</h:commandButton>
```

Similarly a more graphical presentation would be to use a commandlink with an image.

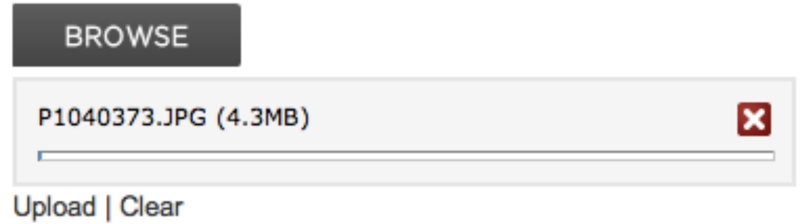
```
<h:commandLink value="Download">
      <p:fileDownload value="#{fileDownloadController.file}"/>
      <h:graphicImage value="pdficon.gif" />
</h:commandLink>
```

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.

3.27 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. Additionally in case the user agent does not support flash or javascript, fileUpload will fallback to the legacy input type="file" and still work.



Info

Tag	**fileUpload**
Tag Class	**org.primefaces.component.fileupload.FileUploadTag**
Component Class	**org.primefaces.component.fileupload.FileUpload**
Component Type	**org.primefaces.component.FileUpload**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.FileUploadRenderer**
Renderer Class	**org.primefaces.component.fileupload.FileUploadRenderer**

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.
fileUploadListener	null	MethodExpression	Method expression to listen file upload events.
multiple	FALSE	boolean	Allows multi file uploads, turned off by default.

Name	Default	Type	Description
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.
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 seperated 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=”#{backingBean.handleFileUpload}” />
```

BROWSE
Upload | Clear

FileUploadListener is the way to access the uploaded files, when a file is uploaded defined fileUploadListener is processed with a FileUploadEvent as the parameter.

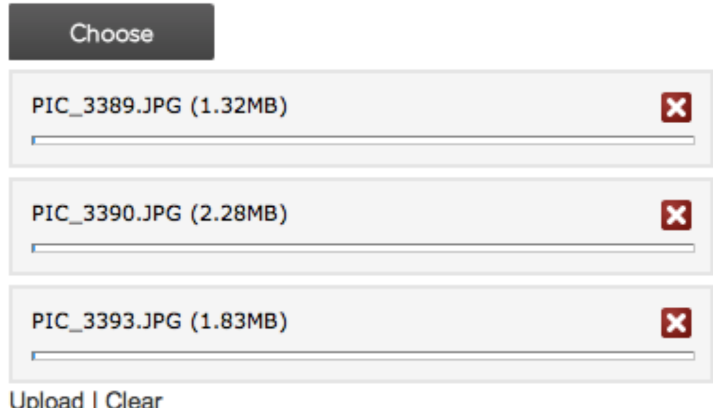
```
public class Controller {

    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=”#{controller.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=”#{controller.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=”#{controller.handleFileUpload}”
     multiple=”true” update=”messages”>
</p:fileUpload>

<p:growl id=”messages” />
```

```
public class Controller {

     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=”#{controller.handleFileUpload}”
```

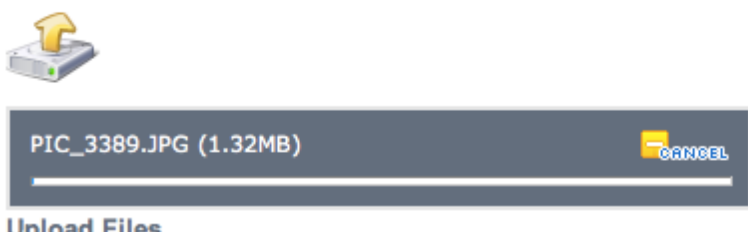
Size Limit

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 10000 bytes for each file.

```
<p:fileUpload fileUploadListener="#{controller.handleFileUpload}"
     sizeLimit="10000" />
```

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="#{fileUploadController.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.

Parameter Name	Description
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>
```

3.28 Focus

Focus is a clever component that makes it easy for page authors to manage the element focus on a JSF page.

Info

Tag	focus
Tag Class	org.primefaces.component.focus.FocusTag
Component Class	org.primefaces.component.focus.Focus
Component Type	org.primefaces.component.Focus.FocusTag
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.FocusRenderer
Renderer Class	org.primefaces.component.focus.FocusRenderer

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
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.

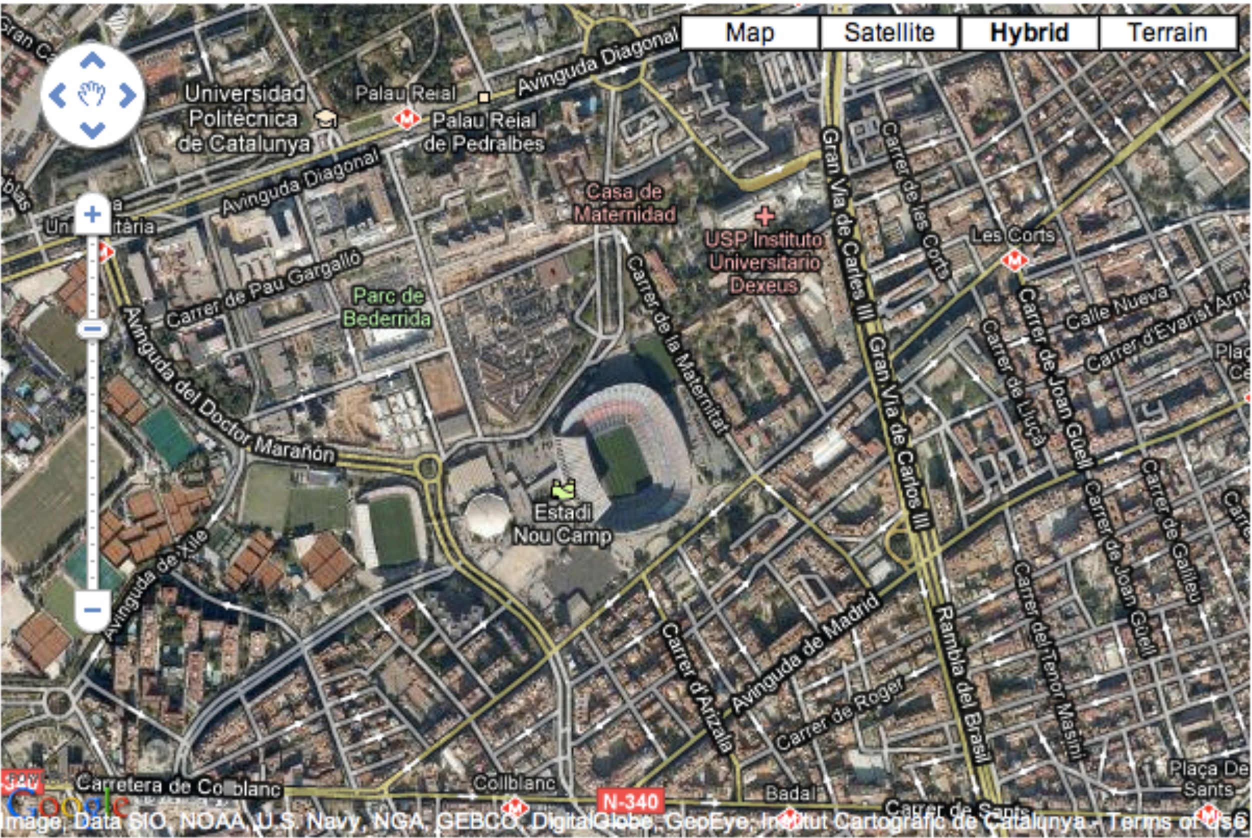
Explicit Focus

Additionally, using for attribute focus can be set explicitly on an input component.

```
<p:focus for="text"/>

<h:inputText id="text" value="{bean.value}" />
```

3.29 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	gmap
Tag Class	org.primefaces.component.gmap.GMapTag
Component Class	org.primefaces.component.gmap.GMap
Component Type	org.primefaces.component.Gmap
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.GmapRenderer
Renderer Class	org.primefaces.component.gmap.GmapRenderer

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.
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	MethodExpression	Server side listener to invoke when an overlay is selected with ajax.
stateChangeListener	null	MethodExpression	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	MethodExpression	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.

Name	Default	Type	Description
markerDragListener	null	MethodExpression	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

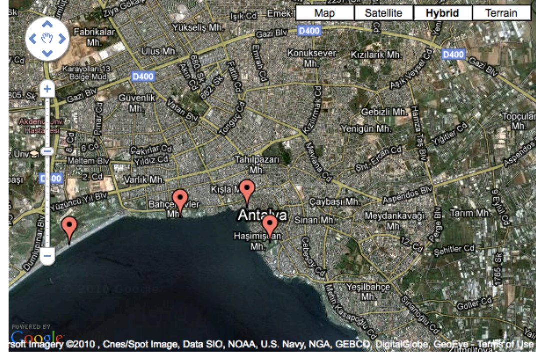
GMap is backed by an *org.primefaces.model.map.MapModel* instance, PrimeFaces provides *org.primefaces.model.map.DefaultMapModel* as the default implementation. API Docs of all GMap related model classes are available at the end of GMap section and also at javadocs of PrimeFaces.

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" value="#{mapBean.model}"/>
```

```
import org.primefaces.model.map.MapModel;
import org.primefaces.model.map.DefaultMapModel;
import org.primefaces.model.map.Marker;
import org.primefaces.model.map.LatLng;

public class MapBean {

   private MapModel model;

   public MapBean() {
      model = new DefaultMapModel();
      model.addOverlay(new Marker(new LatLng(36.879466, 30.667648), "M1"));
      model.addOverlay(new Marker(new LatLng(36.883707, 30.689216), "M2"));
      model.addOverlay(new Marker(new LatLng(36.879703, 30.706707), "M3"));
      model.addOverlay(new Marker(new LatLng(36.885233, 37.702323), "M4"));
   }

   public Model getModel() {
      return model
   }
 }
```

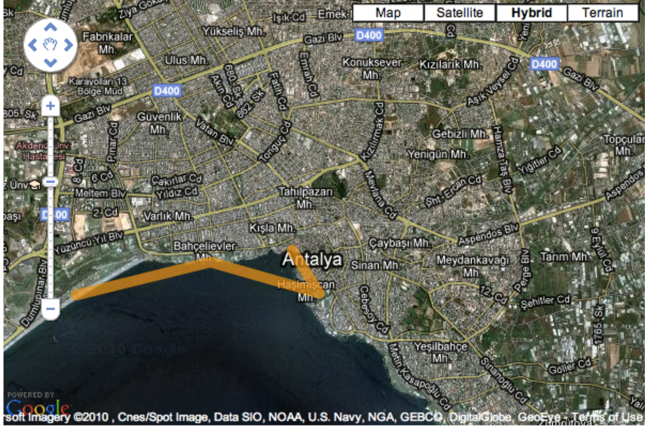


Polylines

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" value=”#{mapBean.model}”/>
```

```
import org.primefaces.model.map.MapModel;
import org.primefaces.model.map.DefaultMapModel;
import org.primefaces.model.map.Polyline;
import org.primefaces.model.map.LatLng;

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 Model getModel() {
      return model
   }
 }
```

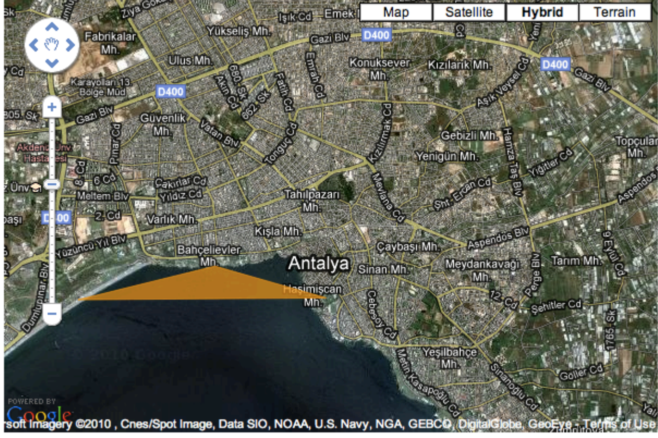


Polygons

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" value=”#{mapBean.model}”/>
```

```
import org.primefaces.model.map.MapModel;
import org.primefaces.model.map.DefaultMapModel;
import org.primefaces.model.map.Polygon;
import org.primefaces.model.map.LatLng;

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 Model getModel() {
      return 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 javascript 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" value=”#{mapBean.model}”
           overlaySelectListener=”#{mapBean.onMarkerSelect}”
           onOverlaySelectUpdate=”growl”/>
</h:form>
```

```
import org.primefaces.model.map.MapModel;
import org.primefaces.model.map.DefaultMapModel;
import org.primefaces.model.map.Marker;
import org.primefaces.model.map.LatLng;
import org.primefaces.event.map.OverlaySelectEvent;

public class MapBean {

   private MapModel model;

   public MapBean() {
      model = new DefaultMapModel();
      //add markers
   }

   public Model 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" value=”#{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 Model getModel() { return model; }

  public Model 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" value=”#{mapBean.model}”
           markerDragListener=”#{mapBean.onMarkerDrag}”
           onMarkerDragUpdate=”growl”/>
</h:form>
```

```
import org.primefaces.model.map.MapModel;
import org.primefaces.model.map.DefaultMapModel;
import org.primefaces.model.map.Marker;
import org.primefaces.model.map.LatLng;
import org.primefaces.event.map.MarkerDragEvent;

public class MapBean {

   private MapModel model;

   public MapBean() {
      model = new DefaultMapModel();
      //create model with draggable markers
   }

   public Model getModel() {
      return model;
   }

   public void onMarkerDrag(MarkerDragEvent event) {
      Marker marker = (Marker) event.getMarker();
      //add facesmessage
   }
 }
```

Map Events

GMap 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" value=”#{mapBean.model}”
            stateChangeListener=”#{mapBean.onStateChange}”
            onStateChangeUpdate=”growl”/>
</h:form>
```

```
import org.primefaces.model.map.MapModel;
import org.primefaces.model.map.DefaultMapModel;
import org.primefaces.model.map.LatLng;
import org.primefaces.model.map.LatLngBounds;
import org.primefaces.event.map.StateChangeEvent;

public class MapBean {

    private MapModel model;

    public MapBean() {
        model = new DefaultMapModel();
        //create model with draggable markers
    }

    public Model 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" value="#{mapBean.model}"
           pointSelectListener="#{mapBean.onPointSelect}"
           onPointSelectUpdate="growl"/>
</h:form>
```

```
import org.primefaces.model.map.MapModel;
import org.primefaces.model.map.DefaultMapModel;
import org.primefaces.model.map.LatLng;
import org.primefaces.event.map.PointSelectEvent;

public class MapBean {

   private MapModel model;

   public MapBean() {
      model = new DefaultMapModel();
      //create model
   }

   public Model getModel() {
      return model;
   }

   public void onPointSelect(PointSelectEvent event) {
      LatLng location = event.getLatLng();

      //add facesmessage
   }
}
```

Street View

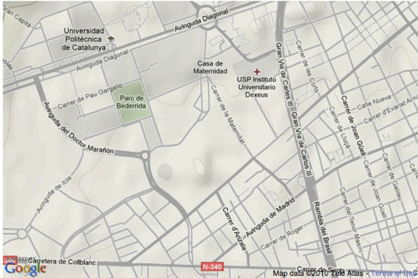
StreeView is enabled simply by setting *streetView* option to true.

```
<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” />
```



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, generted 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

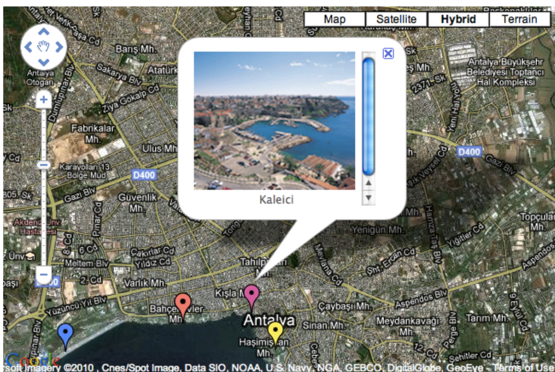
Property	Default	Type	Description
latLng	null	LatLng	Coordionates of the selected point

org.primefaces.event.map.StateChangeEvent

Property	Default	Type	Description
bounds	null	LatLngBounds	Boundaries of the map
zoomLevel	0	Integer	Zoom level of the map

3.30 GMapInfoWindow

GMapInfoWindow is used with GMap component to open a window on map when an overlay is selected.



Info

Tag	gmapInfoWindow
Tag Class	org.primefaces.component.gmap.GMapInfoWindowTag
Component Class	org.primefaces.component.gmap.GMapInfoWindow
Component Type	org.primefaces.component.GMapInfoWindow
Component Family	org.primefaces.component

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
maxWidth	null	Integer	Maximum width of the info window

Getting started with GMapInfoWindow

See GMap section for more information about how gmapInfoWindow is used.

3.31 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	**graphicImage**
Tag Class	**org.primefaces.component.graphicimage.GraphicImageTag**
Component Class	**org.primefaces.component.graphicimage.GraphicImage**
Component Type	**org.primefaces.component.GraphicImage**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.GraphicImageRenderer**
Renderer Class	**org.primefaces.component.graphicimage.GraphicImageRenderer**

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	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

Name	Default	Type	Description
ismap	FALSE	Boolean	Specifies to use a server-side image map
usemap	null	String	Name of the client side map
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=”#{dynamicImageController.image}” />
```

```
public class DynamicImageController {

  private StreamedContent image;
  //getters&setters

  public DynamicImageController() {
  InputStream stream = this.getClass().getResourceAsStream(”barcalogo.jpg”);
  image = new DefaultStreamedContent(stream, “image/jpeg”);
 }
}
```

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.ResultsSet* API has the getBinaryStream() method to read blob files stored in database.

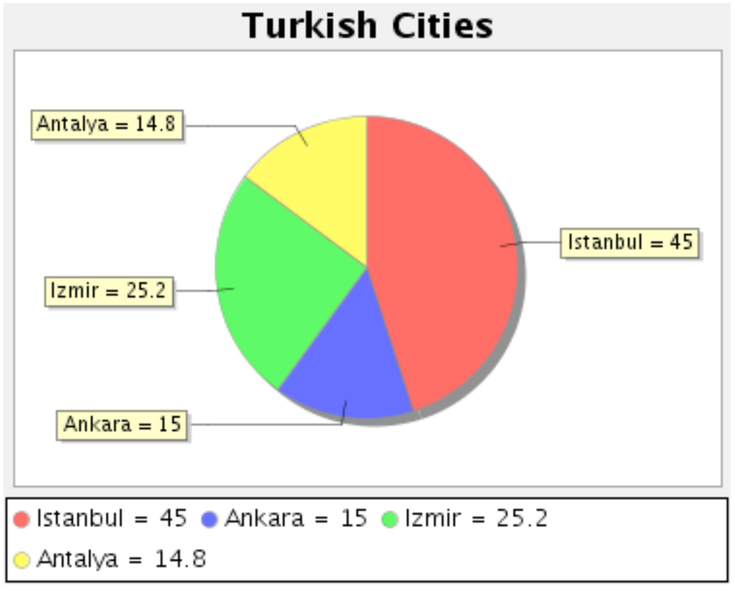
Displaying Charts with JFreeChart

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.

```
public class BackingBean {

	private StreamedContent chartImage;

	public BackingBean() {
		try {
			JFreeChart jfreechart = ChartFactory.createPieChart
("Turkish Cities", createDataset(), true, true, false);
			File chartFile = new File("dynamichart");
			ChartUtilities.saveChartAsPNG(chartFile, jfreechart, 375,
300);
			chartImage = 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;
	}

	//getters and setters
}
```

```
<p:graphicImage value="#{backingBean.chartImage}" />
```

Basically p:graphicImage makes any JSF chart component using JFreechart obselete and lets you to avoid wrappers 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.

```
public class BackingBean {

      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();
            }
      }
      //getters and setters
}
```

```
<p:graphicImage value=”#{backingBean.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" />
```

3.32 GraphicText

GraphicText can convert any text to an image format in runtime.

Info

Tag	**graphicText**
Tag Class	**org.primefaces.component.graphictext.GraphicTextTag**
Component Class	**org.primefaces.component.graphictext.GraphicText**
Component Type	**org.primefaces.component.GraphicText**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.GraphicTextRenderer**
Renderer Class	**org.primefaces.component.graphictext.GraphicTextRenderer**

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	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

Name	Default	Type	Description
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

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”/>
```

3.33 Growl

Growl is based on the Mac's growl notification widget and used to display FacesMessages similar to h:messages.

Info

Tag	growl
Tag Class	org.primefaces.component.growl.GrowlTag
Component Class	org.primefaces.component.growl.Growl
Component Type	org.primefaces.component.Growl
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.GrowlRenderer
Renderer Class	org.primefaces.component.growl.GrowlRenderer

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
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.

Name	Default	Type	Description
globalOnly	FALSE	boolean	When true, only facesmessages without clientids are displayed.
life	6000	integer	Duration in milliseconds to display non-sticky messages.
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 ca 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

If you need to display messages with growl after an ajax request you just need to update it just like a regular component.

```
<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;
}
```

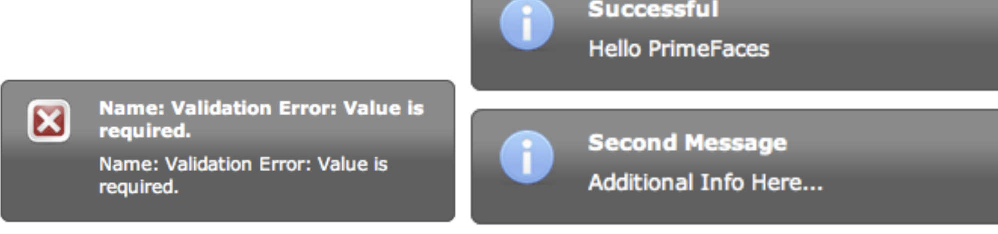
With this setting growl will be located at top left corner.

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;



3.34 HotKey

HotKey is a generic key binding component that can bind any formation of keys to javascript event handlers or ajax calls.

Info

Tag	**hotkey**
Tag Class	**org.primefaces.component.hotkey.HotKeyTag**
Component Class	**org.primefaces.component.hotkey.HotKey**
Component Type	**org.primefaces.component.HotKey**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.HotKeyRenderer**
Renderer Class	**org.primefaces.component.hotkey.HotKeyRenderer**

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
bind	null	String	The Key binding.
handler	null	String	Javascript event handler to be executed when the key binding is pressed.
action	null	javax.el.MethodExpression	A method expression that'd be processed in the partial request caused by uiajax.
actionListener	null	javax.faces.event.ActionListener	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.

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 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. Simples 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

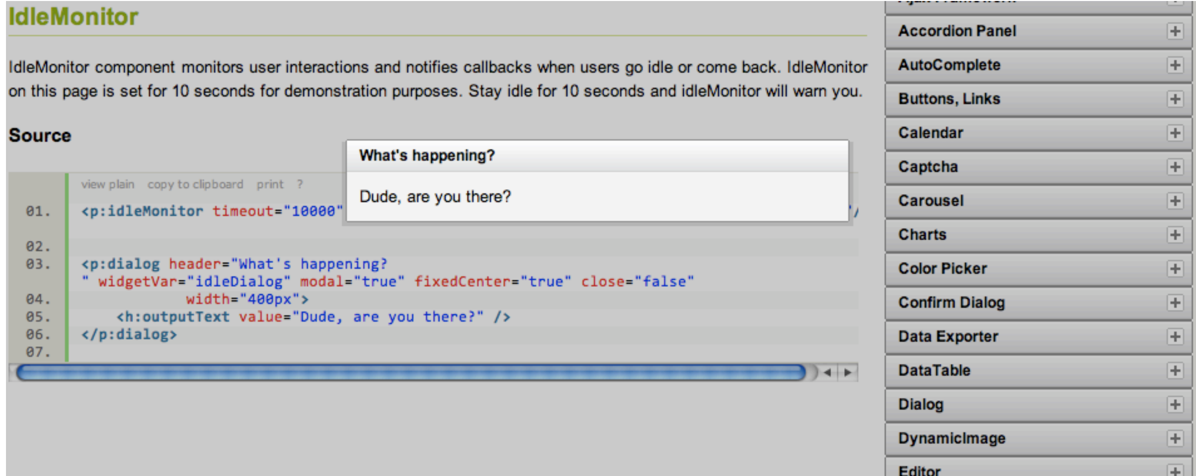
Ajax is a built-in feature of hotKeys meaning you can do ajax calls with key combinations. Following form can be submitted with the *ctrl+shift+s* combination.

```
 <h:form prependId="false">

  <p:hotkey bind="ctrl+shift+s" update="display"
            actionListener="# {hotkeyController.action}"/>

  <h:panelGrid columns="2" style="margin-bottom:10px">
      <h:outputLabel for="firstname" value="Firstname:" />
      <h:inputText id="firstname" value="#{pprBean.firstname}" />
  </h:panelGrid>

  <h:outputText id="dsplay" value="Hello: #{pprBean.firstname}"
            rendered="#{not empty pprBean.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.

3.35 IdleMonitor

IdleMonitor watches users' actions on a page and notify several callbacks in case they go idle or active again.



Tag	**idleMonitor**
Tag Class	**org.primefaces.component.idlemonitor.IdleMonitorTag**
Component Class	**org.primefaces.component.idlemonitor.IdleMonitor**
Component Type	**org.primefaces.component.IdleMonitor**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.IdleMonitorRenderer**
Renderer Class	**org.primefaces.component.idlemonitor.IdleMonitor**

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
timeout	300000	int	Time to wait in milliseconds until deciding if the user is idle. Default is 5 minutes.
onidle	null	String	Javascript event to execute when user goes idle

Name	Default	Type	Description
onactive	null	String	Javascript event to execute when user goes active
idleListener	null	javax.el.Method Expression	Server side event to be called in case user goes idle
update	null	String	Client side id of the component(s) to be updated after async partial submit request

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 prependId=”false”>
  <p:idleMonitor idleListener=”#{idleMonitorController.handleIdle}”
           update=”message”/>

  <h:outputText id=”message” value=”#{idleMonitorController.msg}” />
</h:form>
```

```
public class IdleMonitorController {

     private String msg;

     public String getMsg() {
           return msg;
     }

     public void setMsg(String msg) {
           this.msg = msg;
     }

     public void idleListener(IdleEvent event) {
           msg = “Message from server: Your session is closed”;

           //invalidate session
     }
}
```

Note: An idleMonitor must be enclosed in a form if an idleListener is defined.

3.36 ImageCompare

ImageCompare provides a rich user interface to compare two images.

Info

Tag	**imageCompare**
Tag Class	**org.primefaces.component.imagecompare.ImageCompareTag**
Component Class	**org.primefaces.component.imagecompare.ImageCompare**
Component Type	**org.primefaces.component.ImageCompare**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.ImageCompareRenderer**
Renderer Class	**org.primefaces.component.imagecompare.ImageCompareRenderer**

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.

Name	Default	Type	Description
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
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

Two images are placed inside a div container element, *style* and *styleClass* attributes apply to this element.

3.37 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	imageCropper
Tag Class	org.primefaces.component.imagecropper.ImageCropperTag
Component Class	org.primefaces.component. imagecropper.ImageCropper
Component Type	org.primefaces.component.ImageCropper
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.ImageCropperRenderer
Renderer Class	org.primefaces.component.imagecropper.ImageCropperRenderer

Attributes

Name	Default	Type	Description
id	Assigned by JSF	String	Unique identifier of the component

Name	Default	Type	Description
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	MethodBindi ng	A method binding expression that refers to a method validationg the input
valueChangeListener	null	ValueChang eListener	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	Javascript variable name of the wrapped widget

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="#{myBean.croppedImage}" image="/campnou.jpg" />
```

```
public class MyBean {
	private CroppedImage croppedImage;

	public CroppedImage getCroppedImage() {
		return croppedImage;
	}

	public void setCroppedImage(CroppedImage croppedImage) {
		this.croppedImage = croppedImage;
	}
}
```

CroppedImage is 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

Probably most important property is the bytes since it contains the byte[] representation of the cropped area, an example that saves the cropped part to a folder in web server is described below.

```
<p:imageCropper value="#{myBean.croppedImage}"
                image="/campnou.jpg">
</p:imageCropper>

<h:commandButton value="Crop" action="#{myBean.crop}" />
```

```
public class ImageCropperBean {

	private CroppedImage croppedImage;

	public CroppedImage getCroppedImage() {
		return croppedImage;
	}
	public void setCroppedImage(CroppedImage croppedImage) {
		this.croppedImage = croppedImage;
	}

	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 (FileNotFoundException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}
		return null;
	}
}
```

External Images

ImageCropper has the ability to crop external images as well.

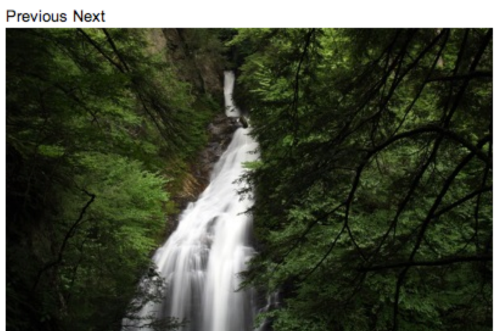
```
<p:imageCropper value=”#{myBean.croppedImage}”
		image=”http://primefaces.prime.com.tr/en/images/schema.png”>
</p:imageCropper>
```

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.

3.38 ImageSwitch

Imageswitch component is used to enable switching between a set of images with some nice effects. ImageSwitch also provides a simple client side api for flexibility.



Info

Tag	imageSwitch
Tag Class	org.primefaces.component.imageswitch.ImageSwitchTag
Component Class	org.primefaces.component.imageswitch.ImageSwitch
Component Type	org.primefaces.component.ImageSwitch
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.ImageSwitchRenderer
Renderer Class	org.primefaces.component.imageswitch.ImageSwitchRenderer

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
effect	null	String	Name of the effect for transition.
speed	500	int	Speed of the effect in milliseconds.
slideshowSpeed	3000	int	Slideshow speed in milliseconds.
slideshowAuto	TRUE	boolean	Starts slideshow automatically on page load.

Name	Default	Type	Description
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>
```

You need to use the ImageSwitch client side api to trigger the transitions. Example below uses two span elements to navigate between the images.

```
<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();	Manually starts a slideshow.
void stopSlideshow();	Manually stops a slidehow.

Use the widgetVar to get the variable name of the client side widget.

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.

- *FadeIn*
- *FlyIn*
- *FlyOut*
- *FlipIn*
- *FlipOut*
- *ScrollIn*
- *ScrollOut*
- *SingleDoor*
- *DoubleDoor*

3.39 Inplace

Inplace provides easy inplace editing and inline content display. Inplace consists of two members, display element is the inital clickable label and inline element is the hidden content that'll be displayed when display element is toggled.

Basic Input: Edit Me

Basic Input: Edit Me

Info

Tag	**inplace**
Tag Class	**org.primefaces.component.inplace.InplaceTag**
Component Class	**org.primefaces.component.inplace.Inplace**
Component Type	**org.primefaces.component.Inplace**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.InplaceRenderer**
Renderer Class	**org.primefaces.component.inplace.InplaceRenderer**

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
label	null	String	Label to be shown in display mode.
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.
widgetVar	null	String	Javascript variable name of the client side object.

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 childs 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 fadeIn and fadeOut meaning display element will fadeOut and inline content will be shown with fadeOut effect. 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>
```

Skinning Inplace

Style Class	Applies
.pf-inplace-highlight	Display element when hovered.
.pf-inplace-display	Display element.
.pf-inplace-display-disabled	Disabled display element.
.pf-inplace-content	Inline content.

3.40 InputMask

InputMask forces an input to fit in a defined mask template.

Date: 10/01/2009
Phone: (213) 421-3423
Phone with Ext: (645) 645-7447 x65474
taxId: 21-3214231
SSN: 534-53-4264
Product Key: nk-231-h432

Info

Tag	**inputMask**
Tag Class	**org.primefaces.component.inputmask.InputMaskTag**
Component Class	**org.primefaces.component.inputmask.InputMask**
Component Type	**org.primefaces.component.InputMask**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.InputMaskRenderer**
Renderer Class	**org.primefaces.component.inputmask.InputMask**

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
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	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	MethodBinding	A method binding expression that refers to a method validationg 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.
accesskey	null	String	Html accesskey attribute
alt	null	String	Html alt attribute
dir	null	String	Html dir attribute
disabled	FALSE	Boolean	Html disabled attribute
lang	null	String	Html lang attribute
maxlength	null	Integer	Html maxlength attribute
onblur	null	String	Html onblur attribute
onchange	null	String	Html onchange attribute
onclick	null	String	Html onclick attribute
ondblclick	null	String	Html ondblclick attribute
onfocus	null	String	Html onfocus attribute
onkeydown	null	String	Html onkeydown attribute
onkeypress	null	String	Html onkeypress attribute

Name	Default	Type	Description
onkeyup	null	String	Html onkeyup attribute
onmousedown	null	String	Html onmousedown attribute
onmousemove	null	String	Html onmousemove attribute
onmouseout	null	String	Html onmouseout attribute
onmouseover	null	String	Html onmouseover attribute
onmouseup	null	String	Html onmouseup attribute
readonly	FALSE	Boolean	Html readonly attribute
size	null	Integer	Html size attribute
style	null	String	Html style attribute
styleClass	null	String	Html styleClass attribute
tabindex	null	Integer	Html tabindex attribute
title	null	String	Html title attribute

Getting Started with InputMask

InputMask is actually an extended h:inputText and usage is very similar. InputMask below enforces input to be in 99/99/9999 date format.

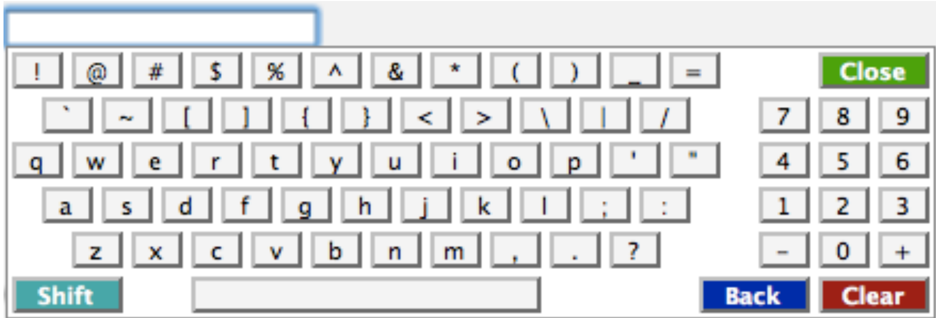
```
<p:inputMask value=”#{bean.field}” mask=”99/99/9999” />
```

Mask Examples

```
<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”/>
```

3.41 Keyboard

Keyboard is an input component that uses a virtual keyboard to provide the input. Important features are the customizable layouts and skinning capabilities.



Info

Tag	**keyboard**
Tag Class	**org.primefaces.component.keyboard.KeyboardTag**
Component Class	**org.primefaces.component.keyboard.Keyboard**
Component Type	**org.primefaces.component.Keyboard**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.KeyboardRenderer**
Renderer Class	**org.primefaces.component.keyboard.KeyboardRenderer**

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 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	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	MethodBi nding	A method binding expression that refers to a method validationg the input
valueChangeListe ner	null	ValueCha ngeListen er	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.
accesskey	null	String	Html accesskey attribute
alt	null	String	Html alt attribute
dir	null	String	Html dir attribute
disabled	FALSE	Boolean	Html disabled attribute
lang	null	String	Html lang attribute
maxlength	null	Integer	Html maxlength attribute
onblur	null	String	Html onblur attribute
onchange	null	String	Html onchange attribute
onclick	null	String	Html onclick attribute
ondblclick	null	String	Html ondblclick attribute
onfocus	null	String	Html onfocus attribute
onkeydown	null	String	Html onkeydown attribute
onkeypress	null	String	Html onkeypress attribute
onkeyup	null	String	Html onkeyup attribute

Name	Default	Type	Description
onmousedown	null	String	Html onmousedown attribute
onmousemove	null	String	Html onmousemove attribute
onmouseout	null	String	Html onmouseout attribute
onmouseover	null	String	Html onmouseover attribute
onmouseup	null	String	Html onmouseup attribute
readonly	FALSE	Boolean	Html readonly attribute
size	null	Integer	Html size attribute
style	null	String	Html style attribute
styleClass	null	String	Html styleClass attribute
tabindex	null	Integer	Html tabindex attribute
title	null	String	Html title attribute
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.

Getting Started with Keyboard

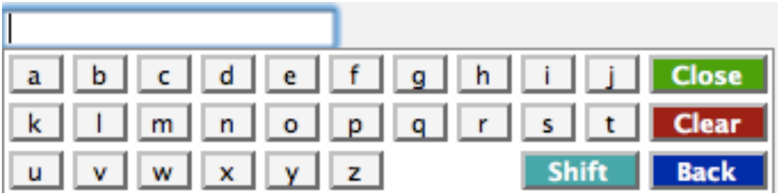
Keyboard is used just like a simple inputText, by default when the input gets the focus a keyboard is displayed.

```
<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.

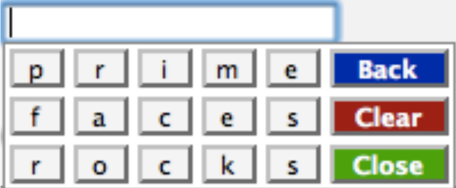
```
<p:keyboard value="#{bean.value}" layout="alphabetic"/>
```



Custom Layouts

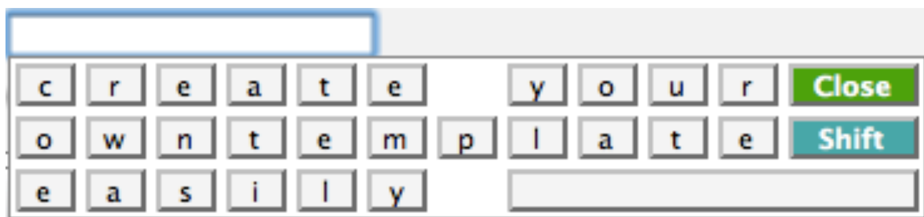
Keyboard has a very flexible layout mechanism allowing you to come up with your own layout.

```
<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"/>
```



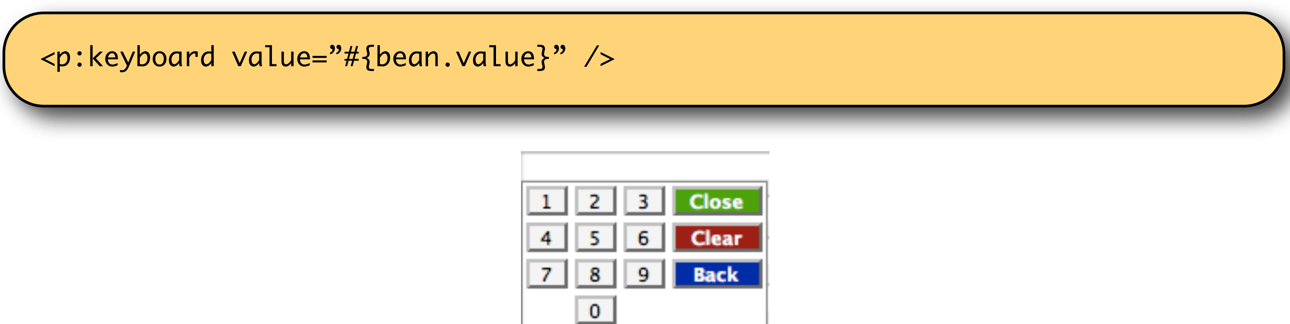
A layout template consists of built-in keys and your own keys. Following is the list of all built-in keys.

- back
- clear
- close
- shift
- spacebar
- space
- halfspace

All other text in a layout is realized as seperate keys so “prime” would create 5 keys as “p” “r” “i” “m” “e”. Use dash to seperate each member in layout and use commas to create a new row.

Keypad

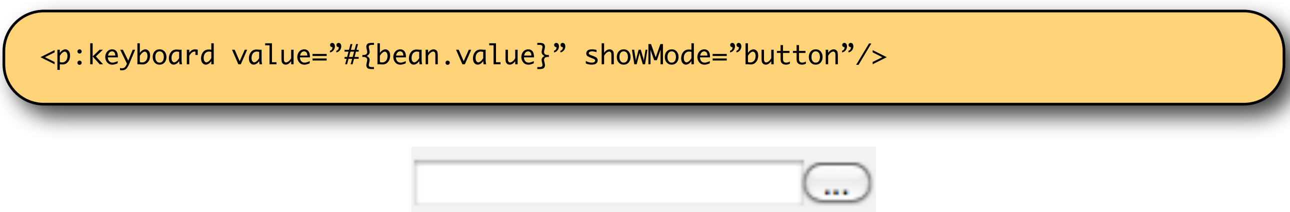
By default keyboard displays whole keys, if you only need the numbers use the keypad mode.



```
<p:keyboard value=”#{bean.value}” />
```


ShowMode

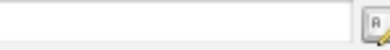
There’re a couple of different ways to display the keyboard, by default keyboard is shown once input field receives the focus. This is customized using the showMode feature which accept values ‘focus’, ‘button’, ‘both’. Keyboard below displays a button next to the input field, when the button is clicked the keyboard is shown.



```
<p:keyboard value=”#{bean.value}” showMode=”button”/>
```

Button can also be customized using the buttonImage and buttonImageOnly attributes.

```
<p:keyboard value=”#{bean.value}” buttonImage=”key.png”
     buttonImageOnly=”true”/>
```



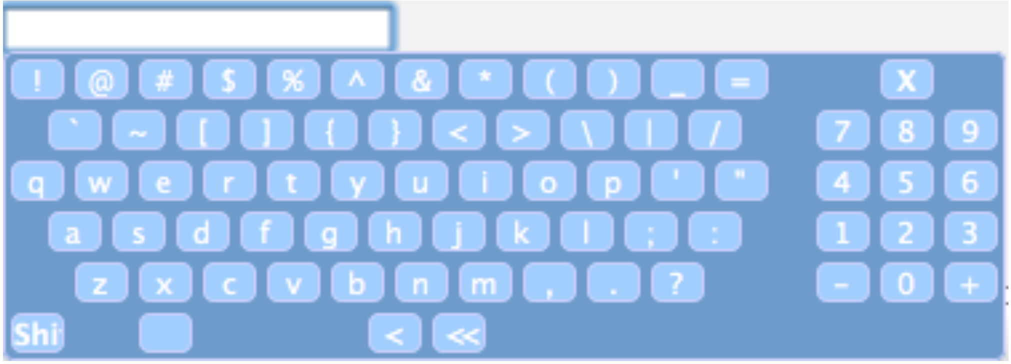
Skinning Keyboard

Skinning keyboard is achieved with CSS. Following are three different skinning examples.

Aqua

```
<p:keyboard value=”#{bean.value}” styleClass=”aqua”/>
```

```
#keypad-div.aqua {
      background: #6699CC;
      border: 1px solid #CCCCFF;
      -moz-border-radius: 4px;
      -webkit-border-radius: 4px;
}

.aqua .keypad-key {
      width: 20px;
      border: 1px solid #CCCCFF;
      -moz-border-radius: 4px;
      -webkit-border-radius: 4px;
      color: #FFFFFF;
      background: #99CCFF;
}

.aqua .keypad-key-down {
      background: #8c8;
}

.aqua .keypad-clear {
      letter-spacing: -3px;
}

.aqua .keypad-space {
      width: 20px;
}
```



Homebrew

```
<p:keyboard value="#{bean.value}" styleClass="homebrew"/>
```

```
#keypad-div.homebrew {
      background: #333333;
      border: 1px solid #CCCCFF;
      -moz-border-radius: 4px;
      -webkit-border-radius: 4px;
}
.homebrew .keypad-key {
      width: 20px;
      border: 1px solid #CCCCFF;
      -moz-border-radius: 4px;
      -webkit-border-radius: 4px;
      color: #000;
      background: #33CC00;
}
.homebrew .keypad-key-down {
      background: #8c8;
}
.homebrew .keypad-clear {
      letter-spacing: -3px;
}
.homebrew .keypad-space {
      width: 20px;
}
```

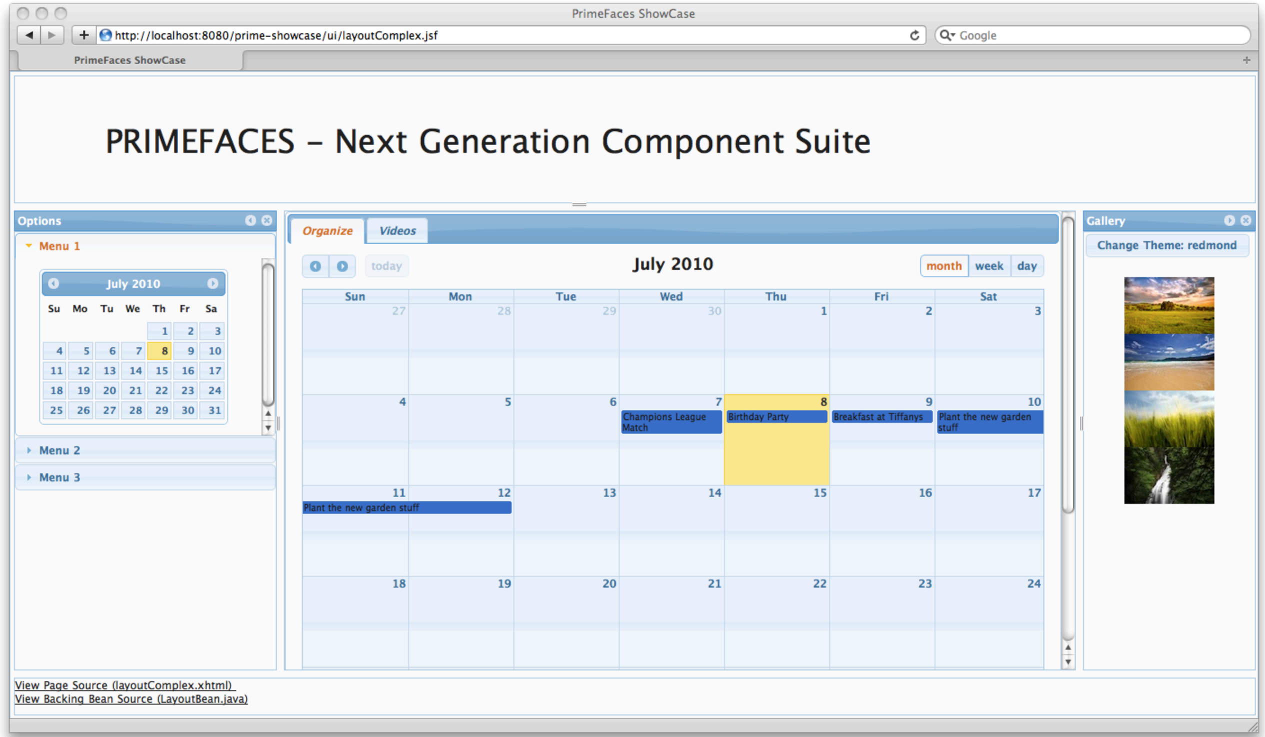
Brownie

```
<p:keyboard value="#{bean.value}" styleClass="brownie"/>
```

```
#keypad-div.brownie {
      background: #330000;
      border: 1px solid #CCCCFF;
      -moz-border-radius: 4px;
      -webkit-border-radius: 4px;
}
.brownie .keypad-key {
      width: 20px;
      border: 1px solid #CCCCFF;
      -moz-border-radius: 4px;
      -webkit-border-radius: 4px;
      color: #000;
      background: #FFFFFF;
}
.brownie .keypad-key-down {
      background: #8c8;
}
.brownie .keypad-clear {
      letter-spacing: -3px;
}
.brownie .keypad-space {
      width: 20px;
}
```

3.42 Layout

Layout component features a highly customizable borderLayout model making it very easy to create complex layouts even if you're not familar with web design.



Info

Tag	layout
Tag Class	org.primefaces.component.layout.LayoutTag
Component Class	org.primefaces.component.layout.Layout
Component Type	org.primefaces.component.Layout
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.LayoutRenderer
Renderer Class	org.primefaces.component.layout.LayoutRenderer

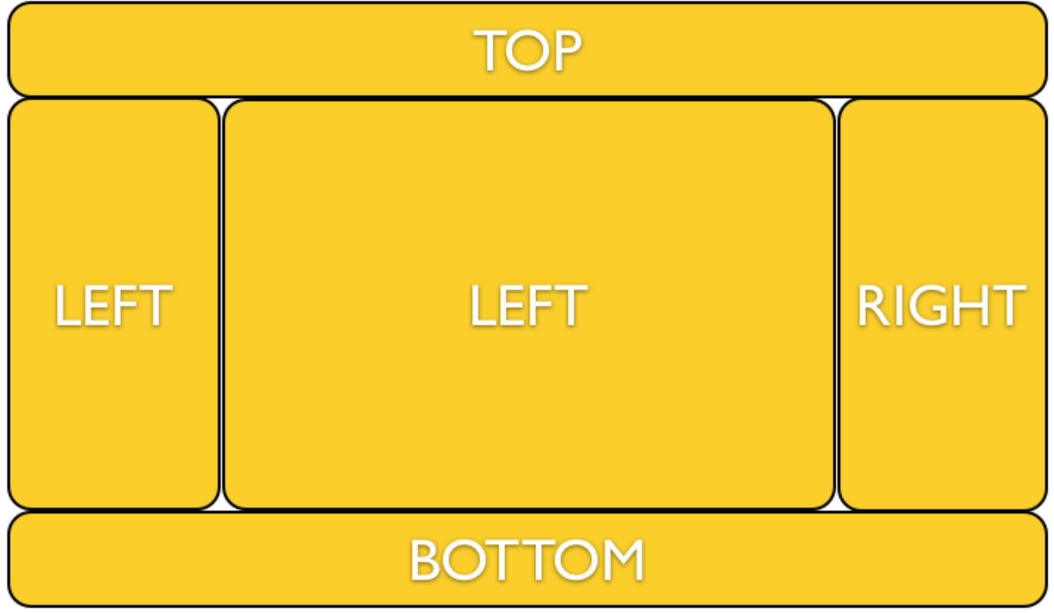
Attributes

Name	Default	Type	Description
id	Assigned by JSF	String	Unique identifier of the component

Name	Default	Type	Description
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.
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	MethodExpression	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	MethodExpression	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	MethodExpression	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;

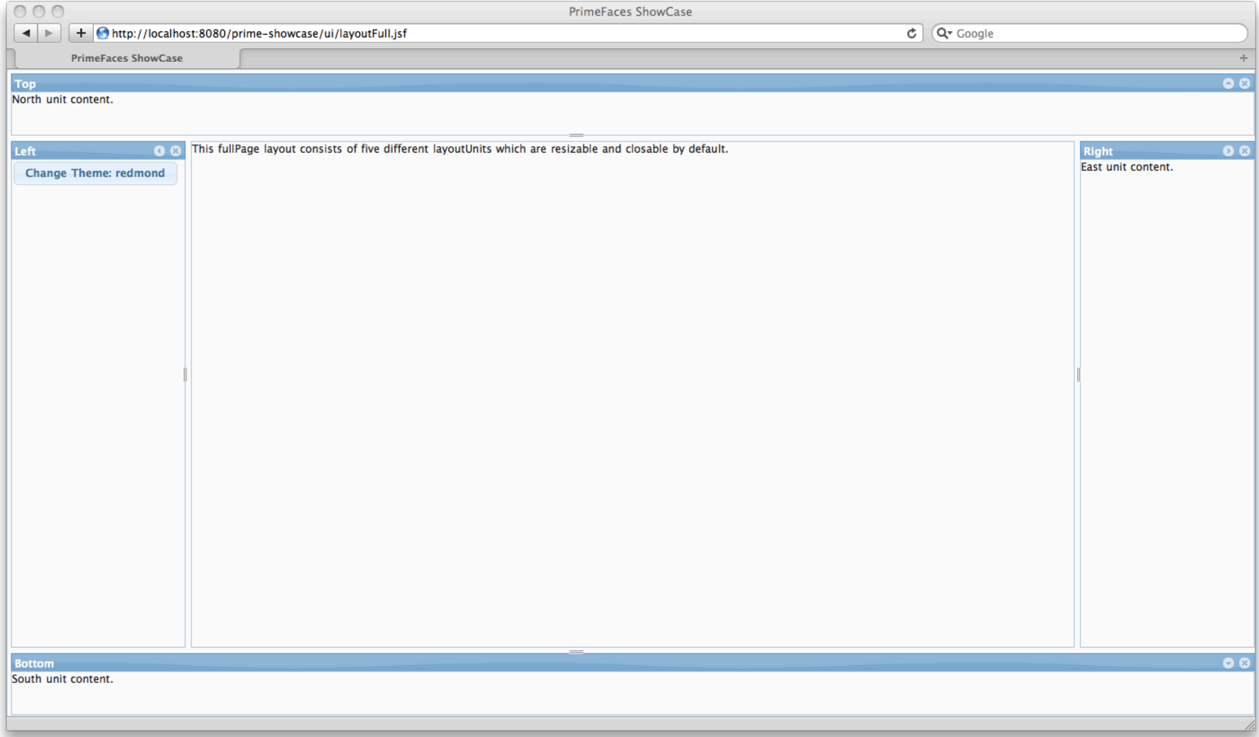


Full Page Layout

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.

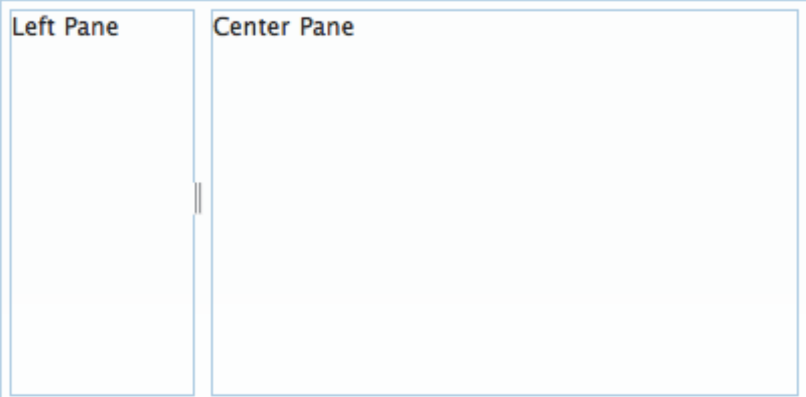
Dimensions

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 API

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

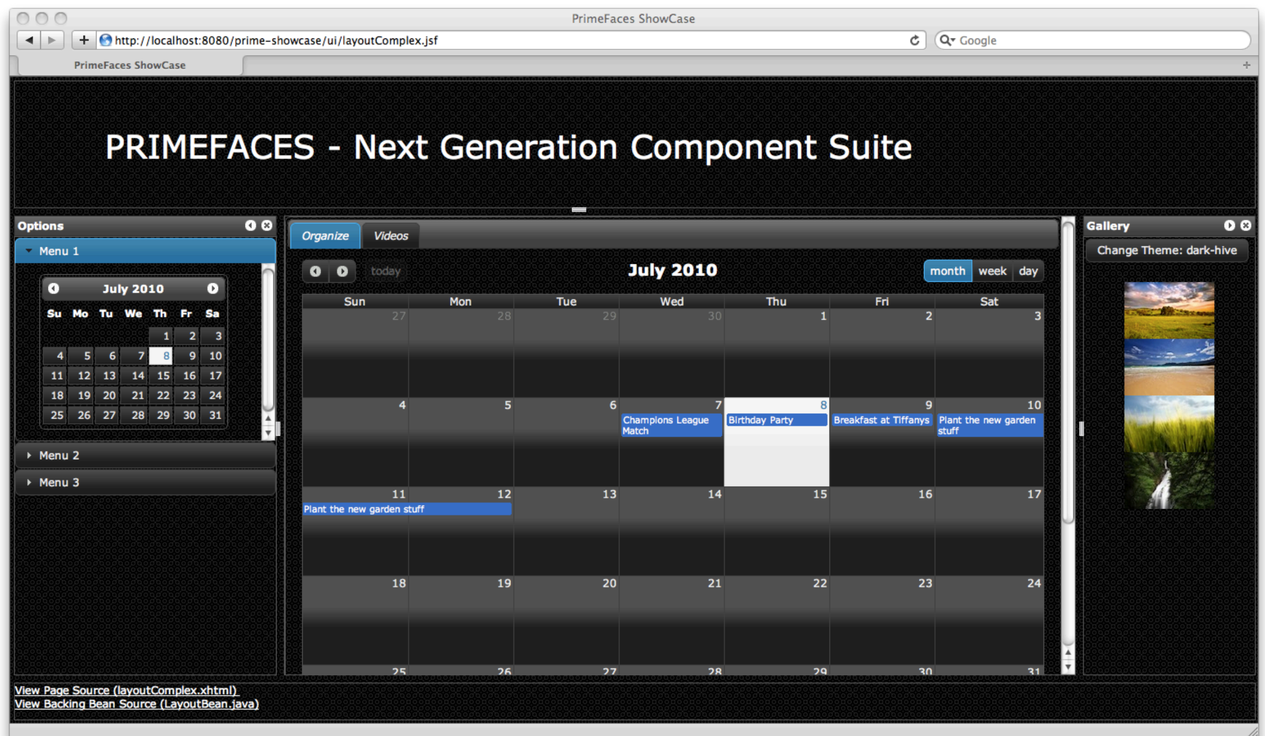
Making layout stateful would be easy, once you create your data to store the user preference, you can update this data using ajax event listeners provided by layout. For example if a layout unit is collapsed, you can save and persist this information. By binding this persisted information to the collapsed 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. Here is an example based on a different theme;



3.43 LayoutUnit

LayoutUnit represents a region in the border layout model of the Layout component.

Info

Tag	layoutUnit
Tag Class	org.primefaces.component.layout.LayoutUnitTag
Component Class	org.primefaces.component.layout.LayoutUnit
Component Type	org.primefaces.component.LayoutUnit
Component Family	org.primefaces.component

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
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

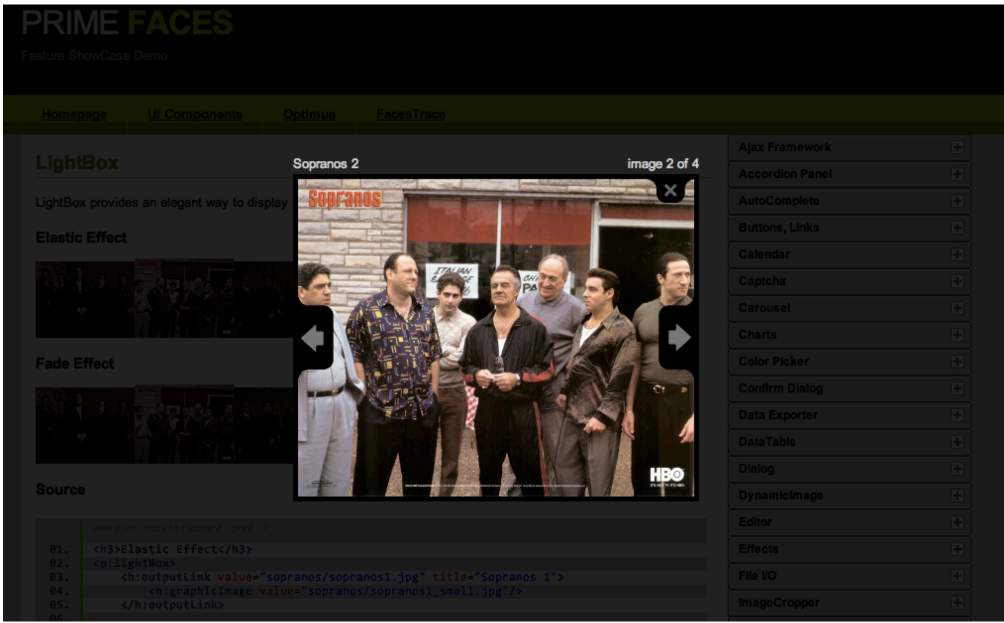
Name	Default	Type	Description
minHeight	null	Integer	Minimum dimension of height in resizing
maxHeight	null	Integer	Maximum dimension of height in resizing
gutter	4px	String	Gutter size of layout unit.
zindex	null	Integer	zindex property to control overlapping with other components
visible	TRUE	Boolean	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.

3.44 LightBox

Lightbox features a powerful overlay that can display images, multimedia content, other JSF components and external urls.



Info

Tag	lightBox
Tag Class	org.primefaces.component.lightbox.LightBoxTag
Component Class	org.primefaces.component lightbox.LightBox
Component Type	org.primefaces.component.LightBox
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.LightBoxRenderer
Renderer Class	org.primefaces.component.lightbox.LightBoxRenderer

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.

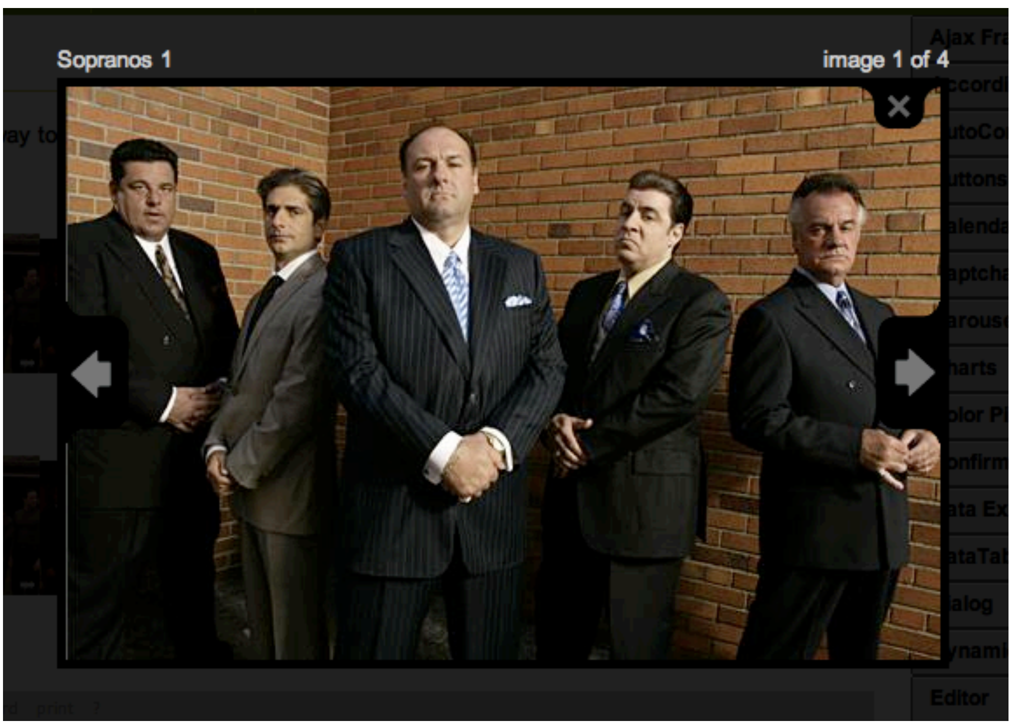
Name	Default	Type	Description
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean
style	null	String	Style of the container element not the overlay element.
styleClass	null	String	Style class of the container element not the overlay element.
widgetVar	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	int	Speed of the transition effect in milliseconds.
width	null	String	Width of the overlay.
heigth	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	int	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;

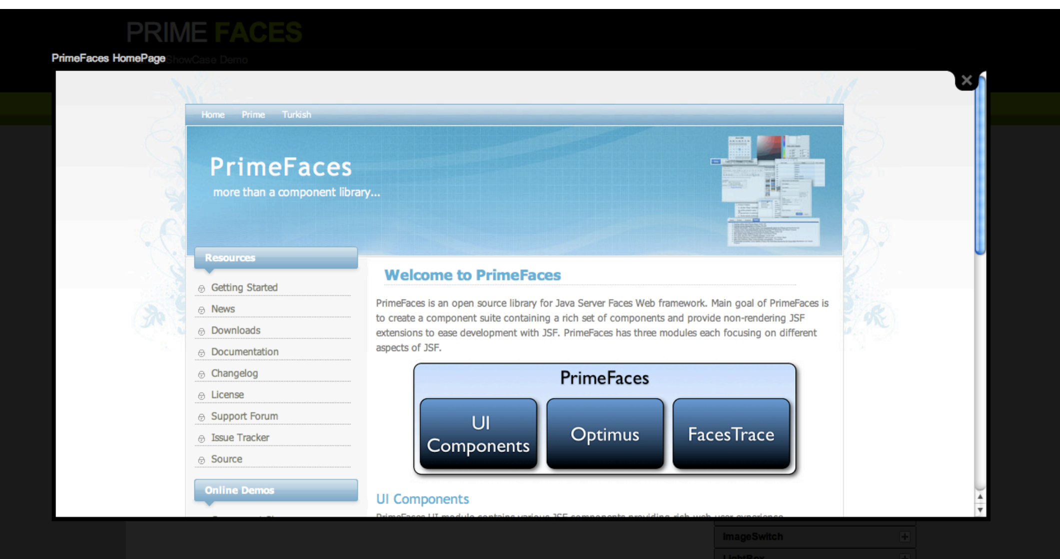


IFrame Mode

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>
```

Skinning LightBox

style and styleClass attributes effect the parent dom element containing the outputLink components. These classes do not effect the overlay. There'll be more customization options for skinning the overlay and built-in themes in future releases.

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>
```

DOCTYPE

If lightbox is not working, it may be due to lack of DOCTYPE declaration.

3.45 LinkButton

LinkButton is a goButton implementation that is used to redirect to a URL.

Info

Tag	linkButton
Tag Class	org.primefaces.component.linkbutton.LinkButtonTag
Component Class	org.primefaces.component.linkbutton.LinkButton
Component Type	org.primefaces.component.LinkButton
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.LinkButtonRenderer
Renderer Class	org.primefaces.component.linkbutton.LinkButtonRenderer

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	String	Label for the link button
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	Id for the button object defined as a YUI button for to be accessed outside.
href	null	String	Href value for the link button used for navigating.
target	null	String	Html anchor target attribute, valid values are "_blank","_top","_parent","_self"

Name	Default	Type	Description
style	null	String	Style to be applied on the button element
styleClass	null	String	StyleClass to be applied on the button element
onblur	null	String	onblur dom event handler
onchange	null	String	onchange dom event handler
onclick	null	String	onclick dom event handler
ondblclick	null	String	ondblclick dom event handler
onfocus	null	String	onfocus 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
onselect	null	String	onselect dom event handler

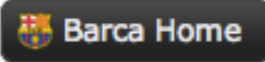
Getting started with LinkButton

LinkButton component requires a link(href) to navigate.

```
<p:linkButton value=”Barca” href=”http://www.fcbarcelona.com” />
```

Skinning LinkButton

Please check commandButton component for styling link button. Here is an example based on a different theme;



3.46 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	**media**
Tag Class	**org.primefaces.component.media.MediaTag**
Component Class	**org.primefaces.component.media.Media**
Component Type	**org.primefaces.component.Media**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.MediaRenderer**
Renderer Class	**org.primefaces.component.media.MediaRenderer**

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	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

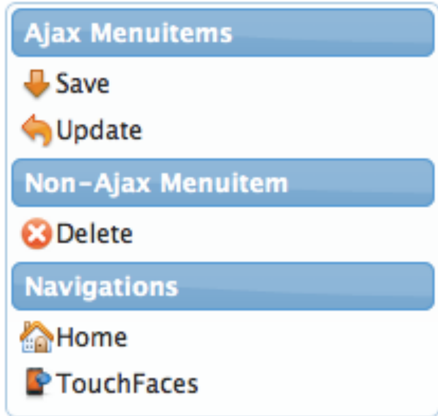
Media component can also play binary media content, example for this use case is storing media files in datatabase using binary format. In order to implement this, create a StreamedContent and set it as the value of media.

```
<p:media value=”#{mediaBean.media}” width=”250” height=”225”
     player=”quicktime”/>
```

```
import java.io.InputStream;

import org.primefaces.model.DefaultStreamedContent;
import org.primefaces.model.StreamedContent;

public class MediaController {

      private StreamedContent media;

      public MediaController() {
            InputStream stream = //Create binary stream from database
            media = new DefaultStreamedContent(stream, “video/quicktime”);
      }

      public StreamedContent getMedia() {
            return media;
      }
}
```

3.47 Menu

Menu is a navigation component with various customized modes like multi tiers, overlay and nested menus.



Info

Tag	**menu**
Tag Class	**org.primefaces.component.menu.MenuTag**
Component Class	**org.primefaces.component.menu.Menu**
Component Type	**org.primefaces.component.Menu**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.MenuRenderer**
Renderer Class	**org.primefaces.component.menu.MenuRenderer**

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.
visible	FALSE	Boolean	Sets menu's visibility. Only *applicable* to dynamic positioned menus.

Name	Default	Type	Description
x	null	Integer	Sets the menu’s left absolute coordinate. Only *applicable* to dynamic positioned menus.
y	null	Integer	Sets the menu’s top absolute coordinate. Only *applicable* to dynamic positioned menus.
fixedCenter	FALSE	Boolean	Boolean value that specifies whether the component should be automatically centered in the viewport on window scroll and resize. Only *applicable* to dynamic positioned menus.
constraintToViewPort	TRUE	FALSE	Boolean indicating if the Menu will try to remain inside the boundaries of the size of viewport. Only applicable to *dynamic* positioned menus.
position	static	String	Sets the way menu is placed on the page, when “static” menu is displayed in the normal flow, when set to “dynamic” menu is not on the normal flow allowing overlaying. Default value is “static”.
clickToHide	TRUE	Boolean	Sets the behavior when outside of the menu is clicked. Only applicable to *dynamic* positioned menus.
keepOpen	FALSE	Boolean	Sets the behavior when the menu is clicked. Only applicable to *dynamic* positioned menus.
tiered	FALSE	Boolean	Sets the tiered mode, when set to true menu will be rendered in different tiers.
effect	FADE	String	Sets the effect for the menu display, default value is FADE. Possible values are “FADE”, “SLIDE”, “NONE”. Use “NONE” to disable animation at all.
effectDuration	0.25	Double	Sets the effect duration in seconds.
autoSubmenuDisplay	TRUE	Boolean	When set to true, submenus are displayed on mouseover of a menuitem.
showDelay	250	Integer	Sets the duration in milliseconds before a submenu is displayed.
hideDelay	0	Integer	Sets the duration in milliseconds before a menu is hidden.
submenuHideDelay	250	Integer	Sets the duration in milliseconds before a submenu is hidden.
context	null	String	Position of the menu.
style	null	String	Style of the main container element.

Name	Default	Type	Description
styleClass	null	String	Style class of the main container element.
zindex	null	Integer	zindex property to control overlapping with other elements.
widgetVar	null	String	Javascript variable name of the wrapped widget.
model	null	MenuModel	MenuModel instance to create menus programmatically

Getting started with the Menu

A menu is composed of submenus and 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>
```

Dynamic Positioning

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 overlaying of other elements. A dynamic menu can be positioned on the page using the *x*,*y*, *fixedCenter* or *context* attributes. X and Y attributes defined the top and left coordinates of the menu. Another alternative is by setting fixedCenter to true, this way menu would be positioned at the center of page.

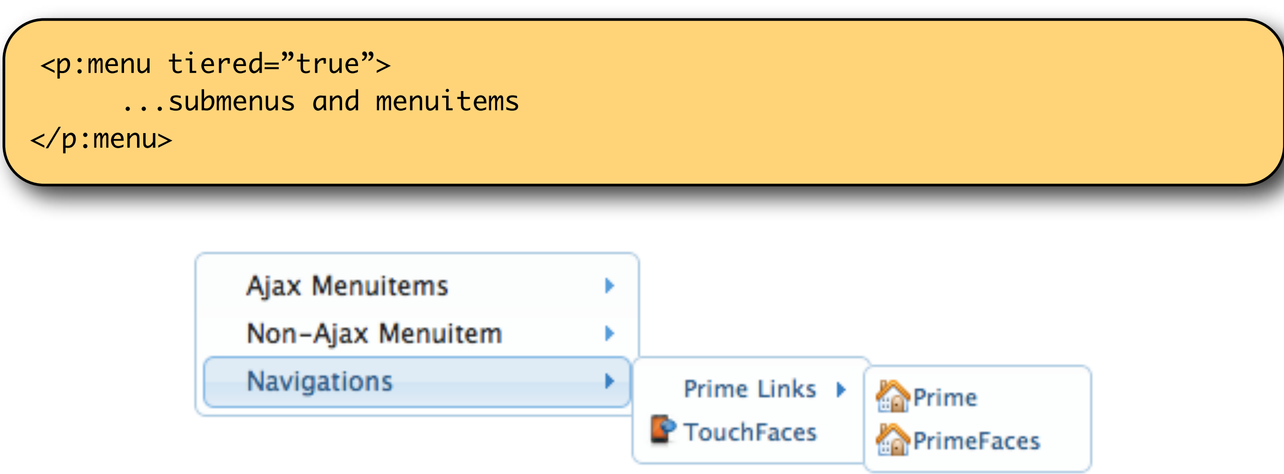
A dynamic menu is not visible by default and show()/hide() client side api is used to change the visibility of menu.

```
<p:menu position="dynamic" widgetVar="myMenu">
      ...submenus and menuitems
</p:menu>

<a href="#" onclick="myMenu.show()">Show</a>
<a href="#" onclick="myMenu.hide()">Hide</a>
```

MultiTiered Menus

By default each submenu is displayed at a single tier, menu also supports nested submenus, *tiered* attribute needs to set to true to enable this features.

```
<p:menu tiered="true">
      ...submenus and menuitems
</p:menu>
```



Ajax and Non-Ajax Actions

As menu uses menuitems, it is easy to invoke actions with or without ajax. 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: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*, *effectDuration*, *showDelay* and more. Full list is described at the attributes table.

Dynamic Menus

Menus can be created programmatically as well, this is much 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}" />
```

```
import org.primefaces.component.menuitem.MenuItem;
import org.primefaces.component.submenu.Submenu;
import org.primefaces.model.DefaultMenuModel;
import org.primefaces.model.MenuModel;

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;
	}
}
```

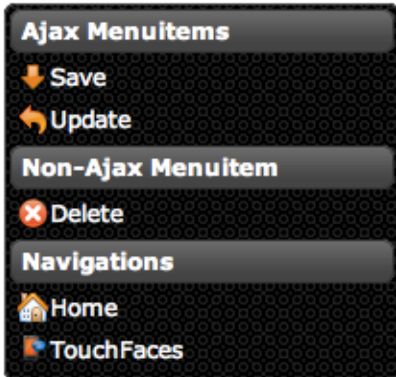
Skinning

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

Following is the list of structural style classes;

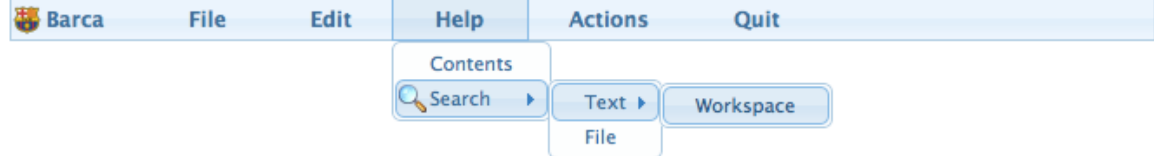
Style Class	Applies
.ui-menu	Container element of menu
.ui-menu-item	Each menu item
.ui-menu-item-label	Each menu item label

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



3.48 Menubar

Menubar is a horizontal navigation component.



Info

Tag	menubar
Tag Class	org.primefaces.component.menubar.MenubarTag
Component Class	org.primefaces.component.menubar.Menubar
Component Type	org.primefaces.component.Menubar
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.MenubarRenderer
Renderer Class	org.primefaces.component.menubar.MenubarRenderer

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.
effect	FADE	String	Sets the effect for the menu display.
effectDuration	0.25	Double	Sets the effect duration in seconds.
autoSubmenuDisplay	FALSE	Boolean	When set to true, submenus are displayed on mouseover of a menuitem.
zindex	null	Integer	zindex property to control overlapping with other elements.
widgetVar	null	String	Name of the client side widget
model	null	MenuModel	MenuModel instance to create menus programmatically

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

Animation to use when showing and hiding menubar contents is customizable using *effect* and *effectDuration* attributes.

```
<p:menubar effect=”FADE|SLIDE|NONE” effectDuration=”1”>
     ...
</p:menubar>
```

Submenu Label Facet

For more powerful customization, submenu supports a facet named “label”. By placing a menuitem as this facet, you can take full advantage of menuitem api.

Following example allows a root menubar item to execute an action to logout the user.

```
<p:menubar>
     <p:submenu>
           <f:facet name=”label”>
                 <p:menuitem label=”Logout” action=”#{bean.logout}”/>
           </f:facet>
     </p:submenu>
</p:menubar>
```

Ajax and Non-Ajax Actions

As menu uses menuitems, it is easy to invoke actions with or without ajax. 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: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.

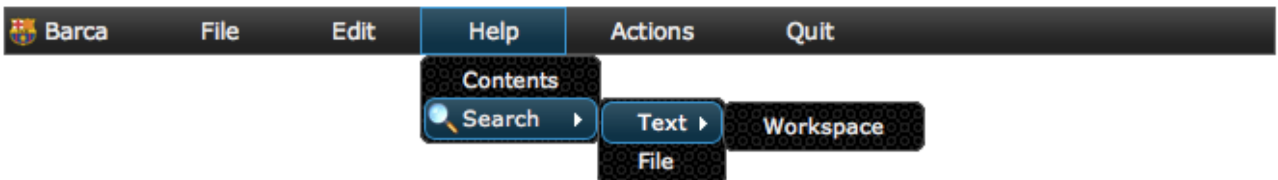
Skinning

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

Following is the list of structural style classes;

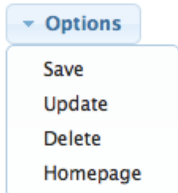
Style Class	Applies
.ui-menubar	Main container element of menubar
.ui-menu	Container element of each submenu
.ui-menubar-item	Each menubar item
.ui-menubar-item-label	Each menubar item label
.ui-menu-item	Each menu item of a submenu
.ui-menu-item-label	Each menu item label of a submenu

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



3.49 MenuButton

MenuButton displays different commands in a popup menu.



Info

Tag	**menuButton**
Tag Class	**org.primefaces.component.menubutton.MenuButtonTag**
Component Class	**org.primefaces.component.menubutton.MenuButton**
Component Type	**org.primefaces.component.MenuButton**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.MenuButtonRenderer**
Renderer Class	**org.primefaces.component.menubutton.MenuButtonRenderer**

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	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.

Getting started with the MenuButton

MenuButton consists of one ore 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>
```

Dynamic Menus

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

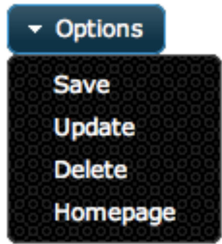
Skinning

MenuButton resides in a main container which *style* and *styleClass* attributes apply.

Following is the list of structural style classes;

Style Class	Applies
.ui-button	Button element
.ui-button-text	Label of button
.ui-menu	Container element of menu
.ui-menu-item	Each menu item
.ui-menu-item-label	Each menu item label

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



3.50 MenuItem

MenuItem is used by various menu components of PrimeFaces.

Info

Tag	**menuItem**
Tag Class	**org.primefaces.component.menuitem.MenuItemTag**
Component Class	**org.primefaces.component.menuitem.MenuItem**
Component Type	**org.primefaces.component.MenuItem**
Component Family	**org.primefaces.component**

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	String	Label of the menuitem
actionListener	null	javax.el.Method Expression	Action listener to be invoked when menuitem is clicked.
action	null	javax.el.Method Expression	Action to be invoked when menuitem is clicked.
immediate	FALSE	Boolean	Specifies the lifecycle phase to process the action event fired by menuitem.
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

Name	Default	Type	Description
onclick	null	String	Javascript event handler for click event
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.
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.
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.

3.51 Message

Message is a pre-skinned extended version of the standard JSF message component.

Text: * text: Validation Error: Value is required.

Info

Tag	message
Tag Class	org.primefaces.component.message.MessageTag
Component Class	org.primefaces.component.message.Message
Component Type	org.primefaces.component.Message
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.MessageRenderer
Renderer Class	org.primefaces.component.message.MessageRenderer

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.
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

Getting started with Message

Message usage is exactly same as standard message.

```
 <h:inputText id=”txt” value=”#{bean.text}” />
<p:message for=”txt” />
```

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.

3.52 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	messages
Tag Class	org.primefaces.component.messages.MessagesTag
Component Class	org.primefaces.component.messages.Messages
Component Type	org.primefaces.component.Messages
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.MessagesRenderer
Renderer Class	org.primefaces.component.messages.MessagesRenderer

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.
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.

Name	Default	Type	Description
globalOnly	FALSE	String	When true, only facesmessages with no clientIds are displayed.
redisplay	TRUE	Boolean	Defines if already rendered messages should be displayed

Getting started with Message

Message usage is exactly same as standard messages.

```
<p:messages />
```

Skinning Message

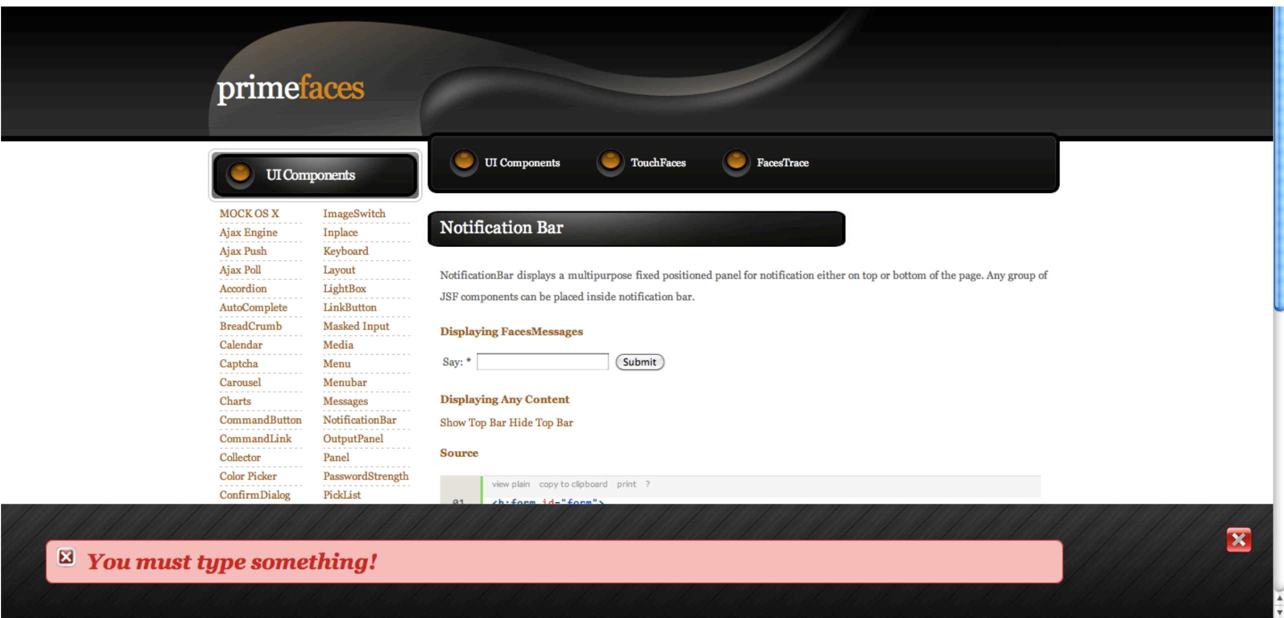
Full list of CSS selectors of message is as follows;

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.

3.53 NotificationBar

NotificationBar displays a multipurpose fixed positioned panel for notification. Any group of JSF content can be placed inside notificationbar.



Info

Tag	notificationBar
Tag Class	org.primefaces.component.notificationbar.NotificationBarTag
Component Class	org.primefaces.component.notificationbar.NotificationBar
Component Type	org.primefaces.component.NotificatonBar
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.NotificationBarRenderer
Renderer Class	org.primefaces.component.notificationbar.NotificationBarRenderer

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.

Name	Default	Type	Description
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
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.show()”>Show</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 it's 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

3.54 OutputPanel

OutputPanel is a display only element that's useful in various cases such as adding placeholders to a page.

Info

Tag	outputPanel
Tag Class	org.primefaces.component.outputpanel.OutputPanelTag
Component Class	org.primefaces.component.outputpanel.OutputPanel
Component Type	org.primefaces.component.OutputPanel
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.OutputPanelRenderer
Renderer Class	org.primefaces.component.output.OutputPanelRenderer

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
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 inline (span) or block(div).

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 if 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 obselete.

Skinning OutputPanel

style and styleClass attributes are used to skin the outputPanel which in turn renders a simple span element. Following outputPanel displays a block container which is also used in the drag&drop example to specify a droppable area.

```
.slot {
    background:#FF9900;
    width:64px;
    height:96px;
}
```

```
<p:outputPanel styleClass="slot" layout="block"><p:outputPanel>
```

3.55 Panel

Panel is a grouping component for other components, notable features are toggling, closing, built-in popup menu and ajax event listeners.

FC Barcelona is one of only three clubs never to have been relegated from La Liga and is the most successful club in Spanish football along with Real Madrid, having won twenty La Liga titles, a record twenty-five Spanish Cups, eight Spanish Super Cups, four Eva Duarte Cups and two League Cups. They are also one of the most successful clubs in European football having won fourteen official major trophies in total, including ten UEFA competitions. They have won three UEFA Champions League titles, a record four UEFA Cup Winners' Cups, a record three Inter-Cities Fairs Cups (the forerunner to the UEFA Europa League), three UEFA Super Cups and one FIFA Club World Cup. The club is also the only European side to have played continental football in every season since its inception in 1955.

Info

Tag	**panel**
Tag Class	**org.primefaces.component.panel.PanelTag**
Component Class	**org.primefaces.component.panel.Panel**
Component Type	**org.primefaces.component.Panel**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.PanelRenderer**
Renderer Class	**org.primefaces.component.panel.PanelRenderer**

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
header	null	String	Header text
footer	null	String	Footer text
toggleable	FALSE	Boolean	Makes panel toggleable.
toggleSpeed	1000	Integer	Speed of toggling in milliseconds

Name	Default	Type	Description
onToggleUpdate	null	String	Component(s) to update after ajax toggleListener is invoked and panel is toggled
toggleListener	null	javax.el MethodExpression	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	javax.el MethodExpression	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 toggles, 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;

About Barca

FC Barcelona is one of only three clubs never to have been relegated from La Liga and is the most successful club in Spanish football along with Real Madrid, having won twenty La Liga titles, a record twenty-five Spanish Cups, eight Spanish Super Cups, four Eva Duarte Cups and two League Cups. They are also one of the most successful clubs in European football having won fourteen official major trophies in total, including ten UEFA competitions. They have won three UEFA Champions League titles, a record four UEFA Cup Winners' Cups, a record three Inter-Cities Fairs Cups (the forerunner to the UEFA Europa League), three UEFA Super Cups and one FIFA Club World Cup. The club is also the only European side to have played continental football in every season since its inception in 1955.

3.56 Password Strength

Password Strength provides a visual feedback regarding password complexity.

Info

Tag	password
Tag Class	org.primefaces.component.password.PasswordTag
Component Class	org.primefaces.component.password.Password
Component Type	org.primefaces.component.Password
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.PasswordRenderer
Renderer Class	org.primefaces.component.password.PasswordRenderer

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 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	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	MethodBinding	A method binding expression that refers to a method validationg 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.
accesskey	null	String	Html accesskey attribute
alt	null	String	Html alt attribute
dir	null	String	Html dir attribute
disabled	FALSE	Boolean	Html disabled attribute
lang	null	String	Html lang attribute
maxlength	null	Integer	Html maxlength attribute
onblur	null	String	Html onblur attribute
onchange	null	String	Html onchange attribute
onclick	null	String	Html onclick attribute
ondblclick	null	String	Html ondblclick attribute
onfocus	null	String	Html onfocus attribute
onkeydown	null	String	Html onkeydown attribute
onkeypress	null	String	Html onkeypress attribute
onkeyup	null	String	Html onkeyup attribute
onmousedown	null	String	Html onmousedown attribute
onmousemove	null	String	Html onmousemove attribute

Name	Default	Type	Description
onmouseout	null	String	Html onmouseout attribute
onmouseover	null	String	Html onmouseover attribute
onmouseup	null	String	Html onmouseup attribute
readonly	FALSE	Boolean	Html readonly attribute
size	null	Integer	Html size attribute
style	null	String	Html style attribute
styleClass	null	String	Html styleClass attribute
tabindex	null	Integer	Html tabindex attribute
title	null	String	Html title attribute
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.
autocomplete	FALSE	String	Controls native autocomplete support of browser.
widgetVar	null	String	Javascript variable name of the client side Password strength object.

Getting Started with Password

Password is an input component and used just like a standard input text.

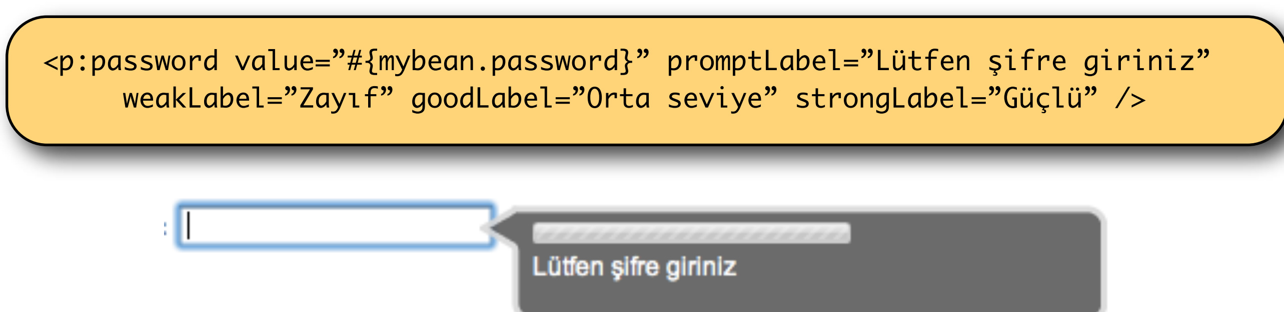
```
<p:password value=”#{mybean.password}” />
```

```
public class MyBean {

    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="#{mybean.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.

3.57 PickList

PickList is used for transferring data between two different collections.

Info

Tag	**pickList**
Tag Class	**org.primefaces.component.picklist.PanelTag**
Component Class	**org.primefaces.component.picklist.Panel**
Component Type	**org.primefaces.component.PickList**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.PickListRenderer**
Renderer Class	**org.primefaces.component.picklist.PickListRenderer**

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 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	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	MethodBinding	A method binding expression that refers to a method validationg 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.
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	Javascript variable name of the client side PickList object.

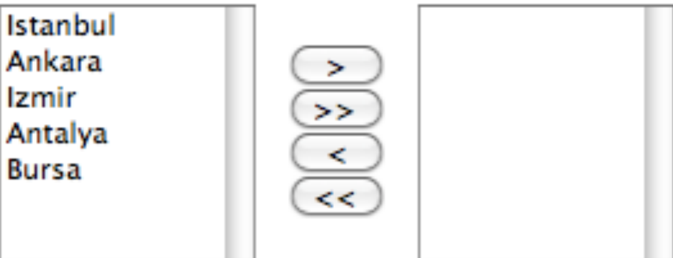
Getting started with PickList

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 DuaListModel 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 you submit the form containing the pickList, the data model will be populated with the new values and you can access these values with DualListModel.getSource() and DualListModel.getTarget() api.

Complex Pojos

Most of the time you would deal with complex pojos rather than primitive types like String. This use case is no different excepty 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;
    }
}
```

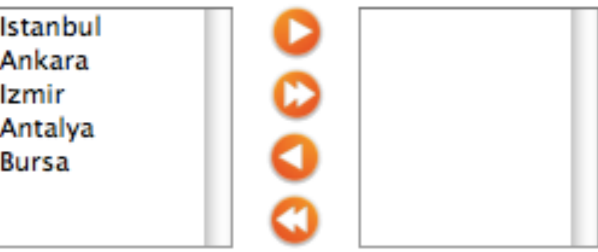
```
<p:pickList value=”#{pickListBean.players}” var=”player”
    itemLabel=”#{player.name}” itemValue=”#{player}” converter=”player”>
```

Customizing Controls

PickList is a composite component and as other PrimeFaces composite components, pickList provides a customizable UI. Using the facet based approach you can customize which controls would be displayed and how.

```
<p:pickList value=”#{pickListBean.players}” var=”player”
    itemLabel=”#{player.name}” itemValue=”#{player}” converter=”player”>

    <f:facet name=”add”>
        <p:graphicImage value=”/images/picklist/add.png”/>
    </f:facet>
    <f:facet name=”addAll”>
        <p:graphicImage value=”/images/picklist/addall.png”/>
    </f:facet>
```

```
        <f:facet name="remove">
              <p:graphicImage value="/images/picklist/remove.png"/>
        </f:facet>
        <f:facet name="removeAll">
              <p:graphicImage value="/images/picklist/removeall.png"/>
        </f:facet>
</p:pickList>
```



Skinning PickList

In addition to the customized controls, there're a couple of css selectors applying to picklist.

Class	Applies
.ui-picklist-source	Source listbox
.ui-picklist-target	Target listbox
.ui-picklist-control	Container for a picklist control (add, remove eg.)

PickList is located inside an container element which can be styled using *style* and *styleClass* attributes.

3.58 Poll

Poll is an ajax component that has the ability to send periodical ajax requests and execute actionlisteners on JSF backing beans.

Info

Tag	**poll**
Tag Class	**org.primefaces.component.poll.PollTag**
Component Class	**org.primefaces.component.poll.Poll**
Component Type	**org.primefaces.component.Poll**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.PollRenderer**
Renderer Class	**org.primefaces.component.poll.PollRenderer**

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.
interval	2	Integer	Interval in seconds to do periodic ajax requests.
action	null	javax.el.MethodExpression	A method expression that'd be processed in the partial request caused by uiajax.
actionListener	null	javax.faces.event.ActionListener	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.
widgetVar	null	String	Javascript variable name of the poll object.

Name	Default	Type	Description
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.
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:form>
    <h:outputText id="txt_count" value="#{counterBean.count}" />
    <p:poll actionListener="#{counterBean.increment}" update="txt_count" />
</h:form>
```

```
public class CounterBean {

    private int count;

    public void increment(ActionEvent actionEvent) {
        count++;
    }
    //getters and setters
}
```

Tuning timing

By default the periodic interval is 2 seconds, this can be changed with the interval attribute. Following poll works every 5 seconds.

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

<p:poll interval="5" actionListener="#{counterBean.increment}"
      update="txt" />
```

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>
```

3.59 Printer

Printer allows sending a specific JSF component to the printer, not the whole page.

Info

Tag	printer
Tag Class	org.primefaces.component.printer.PrinterTag
Component Class	org.primefaces.component.printer.Printer
Component Type	org.primefaces.component.Printer
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.PrinterRenderer
Renderer Class	org.primefaces.component.printer.PrinterRenderer

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
target	null	String	Server side id of a JSF component to print.

Getting started with the Printer

Printer is attached to any action component like a button or a link. Printer below allows printing only the outputText, not the whole page.

```
 <h:commandButton id="btn" value="Print">
      <p:printer target="output" />
</h:commandButton>

<h:outputText id="output" value="PrimeFaces Rocks!" />
```

Following printer prints an image on the page.

```
<h:outputLink id="lnk" value="#">
     <p:printer target="image" />
     <h:outputText value="Print Image" />
</h:outputLink>

<p:graphicImage id="image" value="/images/nature1.jpg" />
```

3.60 ProgressBar

ProgressBar is a process status indicator that can either work purely on client side or interact with server side using ajax.

Info

Tag	propressBar
Tag Class	org.primefaces.component.progressbar.ProgressBarTag
Component Class	org.primefaces.component.progressbar.ProgressBar
Component Type	org.primefaces.component.Progressbar
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.ProgressBarRenderer
Renderer Class	org.primefaces.component.progressbar.ProgressBarRenderer

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
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.

Name	Default	Type	Description
style	null	String	Inline style of the main container element.
styleClass	null	String	Style class of the main container element.
onCompleteUpdate	null	String	Specifies component(s) to update with ajax when progress is completed
completeListener	null	MethodExpression	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	MethodExpression	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>
```

Simple client side api is provided to retrieve and set the value of the progressbar.

- setValue(value) : Updates the progress value
- getValue() : Returns the current progress value

Interval

ProgressBar is based on polling and 3000 milliseconds is the default interval 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 Progress

Ajax mode is enabled by setting ajax attribute to true, in this case the value defined on a JSF backing bean is retrieved peridically and used to update the progress.

```
<p:progressBar ajax="true" value="#{progressBean.progress}"/>
```

```
public class ProgressBean {

	private int progress;
	//getters and setters
}
```

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;

	//getters and setters

	public void handleComplete() {
		//Add a faces message
	}

	public void handleCancel() {
		//Add a faces message
	}

}
```

```
<p:progressBar widgetVar=”pb” value=”#{progressBean.progress}”
           completeListener=”#{progressBean.handleComplete}”
           onCompleteUpdate=”messages”
           cancelListener=”#{progressBean.handleCancel}”
           onCancelUpdate=”messages”
           ajax=”true”/>

<p:growl id=”messages” />
```

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;

3.61 Push

Push component is an agent that creates a channel between the server and the client.

Info

Tag	push
Tag Class	org.primefaces.component.push.PushTag
Component Class	org.primefaces.component.push.Push
Component Type	org.primefaces.component.Push
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.PushRenderer
Renderer Class	org.primefaces.component.push.PushRenderer

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
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.

3.62 Rating

Rating component features a star based rating system. Rating can be used as a plain input component or with ajax RateListeners.

Info

Tag	rating
Tag Class	org.primefaces.component.rating.RatingTag
Component Class	org.primefaces.component.rating.Rating
Component Type	org.primefaces.component.Rating
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.RatingRenderer
Renderer Class	org.primefaces.component.rating.RatingRenderer

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 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	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	MethodBinding	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.
stars	5	int	Number of stars to display
rateListener	null	javax.el.MethodExpression	A server side listener to process a RateEvent
update	null	String	Client side id of the component(s) to be updated after async partial submit request
disabled	FALSE	boolean	Disabled user interaction

Getting Started with Rating

Rating is an input component that takes a double variable as it’s value.

```
public class RatingController {

	private double rating;
	//Getters and Setters
}
```

```
<p:rating value=”#{ratingController.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 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="msg"/>

<h:outputText id="msg" value="#{ratingController.message}" />
```

```
public class RatingController {

	private String message;

	public String getMessage() {
		return message;
	}
	public void setMessage(String message) {
		this.message = message;
	}

	public void handleRate(RateEvent rateEvent) {
		message = "You rated:" + rateEvent.getRating();
	}
}
```

3.63 RemoteCommand

RemoteCommand provides a way to execute JSF backing bean methods directly from javascript.

Info

Tag	remoteCommand
Tag Class	org.primefaces.component.remotecommand.RemoteCommandTag
Component Class	org.primefaces.component.remotecommand.RemoteCommand
Component Type	org.primefaces.component.RemoteCommand
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.RemoteCommandRenderer
Renderer Class	org.primefaces.component.remotecommand.RemoteCommandRenderer

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
action	null	javax.el.MethodExpression	A method expression that'd be processed in the partial request caused by uiajax.
actionListener	null	javax.faces.event.ActionListener	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.

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 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 RemoteCommand

RemoteCommand is used by invoking the command from your javascript function.

```
 <p:remoteCommand name=”increment” actionListener=”#{counter.increment}”
      out=”count” />
<h:outputText id=”count” value=”#{counter.count}” />
```

```
 <script type=”text/javascript”>
function customfunction() {
 //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 and output text is updated.

Note that remoteCommand must be nested inside a form.

3.64 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	**resizable**
Tag Class	**org.primefaces.component.resizable.ResizableTag**
Component Class	**org.primefaces.component.resizable.Resizable**
Component Type	**org.primefaces.component.Resizable**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.ResizableRenderer**
Renderer Class	**org.primefaces.component.resizable.ResizableRenderer**

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
widgetVar	null	String	Javascript variable name of the wrapped widget
proxy	FALSE	boolean	Displays a proxy when resizing
status	FALSE	boolean	Shows the height and width of the resizing component
handles	null	boolean	Handles to use, any combination of 't', 'b', 'r', 'l', 'bl', 'br', 'tl', 'tr' is valid, shortcut "all" enables all handlers.
ghost	FALSE	String	Displays a ghost effect
knobHandles	FALSE	boolean	Displays smaller handles

Name	Default	Type	Description
animate	FALSE	String	Controls animation
effect	null	String	Effect for animation
animateDuration	0.75	double	Duration of animation
minWidth	null	Integer	Minimum width to resize
maxWidth	null	Integer	Maximum width to resize
minHeight	null	Integer	Minimum height to resize
maxHeight	null	Integer	Maximum height to resize

Getting started with resizable

To make a component resizable, just add p:resizable as a child to a parent component that needs to be resized;

```
<h:graphicImage id="img" value="/ui/barca/campnou.jpg">
    <p:resizable  />
</h:graphicImage>
```

That's it now an image can be resized by the user if he/she wants to see more detail :) Another common use case 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 />
</h:inputTextarea>
```

Note: It's important for components that're resized to have an assigned id because some components do not render their clientId's if you don't give them an id explicitly.

Animations

Other than plain resize handling, animations and effects are also supported.

```
<h:inputTextarea id="area" value="Resize me!!!">
    <p:resizable proxy="true" animate="true" effect="bounceOut"/>
</h:inputTextarea>
```

Effect names

- 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

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!!!">
    <p:resizable maxWidth="200" maxHeight="200"/>
</h:inputTextarea>
```

3.65 Resource

Resource component manually adds resources like javascript and css bundled with PrimeFaces to a page.

Info

Tag	**resource**
Tag Class	**org.primefaces.component.resource.ResourceTag**
Component Class	**org.primefaces.component.resource.Resource**
Component Type	**org.primefaces.component.Resource**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.ResourceRenderer**
Renderer Class	**org.primefaces.component.resource.ResourceRenderer**

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
name	null	String	Path of the resource

Getting started with resource

The best place to locate p:resource is inside head tag. Following resource adds jquery bundled with Primefaces to the page.

```
<head>
   <p:resource name="/jquery/jquery.js" />
</head>
```

3.66 Resources

Resources component renders all script and link tags necessary for PrimeFaces component to to work.

Info

Tag	**resources**
Tag Class	**org.primefaces.component.resources.ResourcesTag**
Component Class	**org.primefaces.component.resources.Resources**
Component Type	**org.primefaces.component.Resources**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.ResourcesRenderer**
Renderer Class	**org.primefaces.component.resources.ResourcesRenderer**

Attributes

Name	Default	Type	Description
id	Assigne d 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 seperated list of resources to be excluded.

Getting started with resources

The best place to locate p:resources is inside head tag.

```
<head>
  <p:resources />
</head>
```

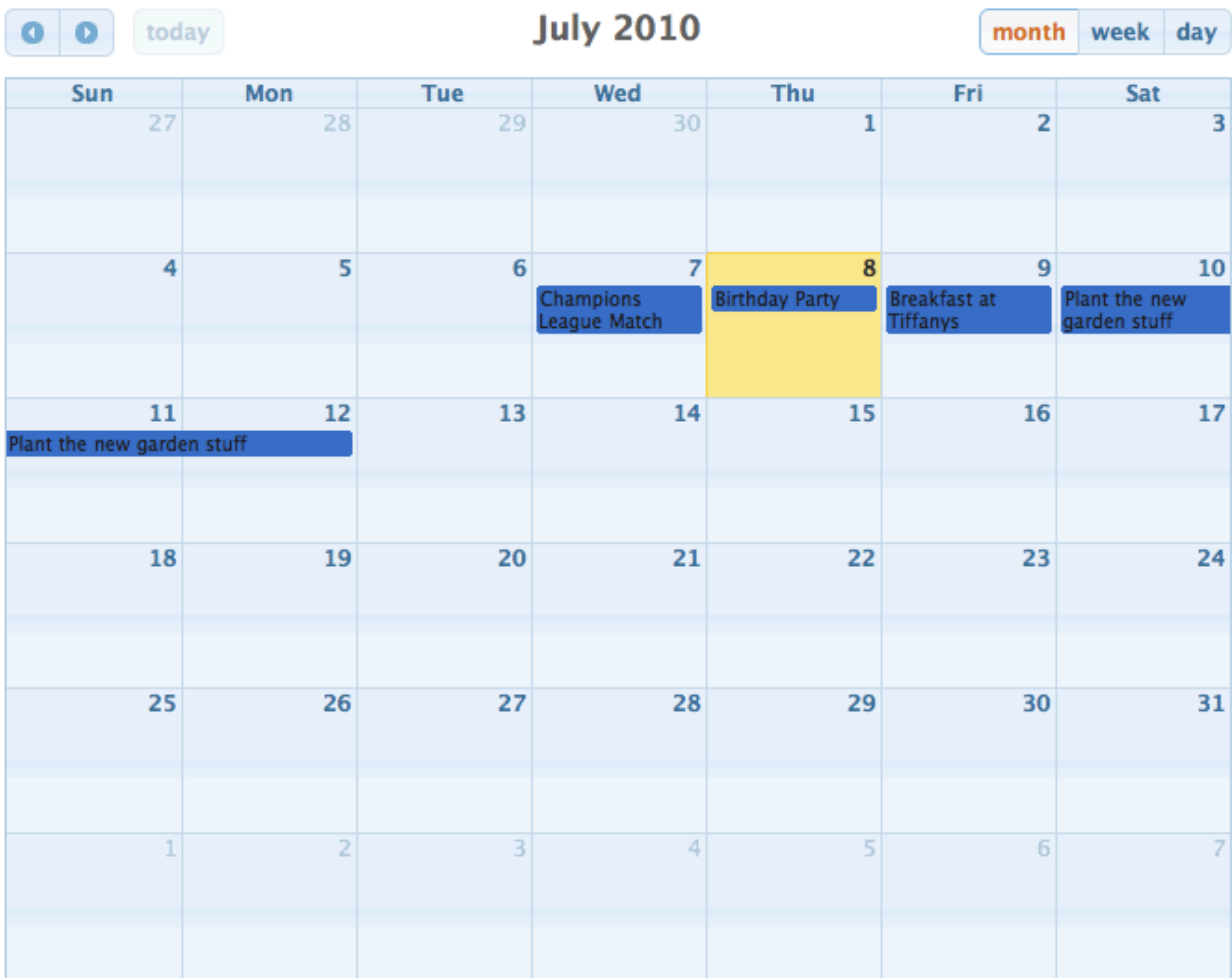
Excluding a resource

In case you'd like to avoid inclusion of a certain resource, use the exlude setting. If there're more than once resources to exclude, provide a comma seperated list.

```
<p:resources exclude="/jquery/jquery.js">
```

3.67 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

Tag	schedule
Tag Class	org.primefaces.component.schedule.ScheduleTag
Component Class	org.primefaces.component.schedule.Schedule
Component Type	org.primefaces.component.Schedule
Component Family	org.primefaces
Renderer Type	org.primefaces.component.ScheduleRenderer
Renderer Class	org.primefaces.component.schedule.ScheduleRenderer

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
widgetVar	null	String	Javascript variable name of the 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
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 ommitted, 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	Method Expression	A server side listener to invoke when an event is selected
dateSelectListener	null	Method Expression	A server side listener to invoke when a date is selected

Name	Default	Type	Description
eventMoveListener	null	Method Expression	A server side listener to invoke when a date is moved
eventResizeListener	null	Method Expression	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.
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.

Editable Schedule

Schedule is read-only by default, to enable event editing, set editable property to true. ScheduleEventDialog is an optional abstact dialog where implementors can customize with the UI they want to display. Here's an example;

```
<p:schedule value="#{scheduleBean.model}" editable="true"
     widgetVar="myschedule">

<p:scheduleEventDialog header="Event Details">
  <h:panelGrid columns="2">
      <h:outputLabel for="title" value="Title:" />
      <h:inputText id="title" value="#{scheduleBean.event.title}" />

      <h:outputLabel for="from" value="From:" />
      <h:inputText id="from" value="#{scheduleBean.event.startDate}" />
            <f:convertDateTime pattern="dd/MM/yyyy" />
      </p:inputMask>

      <h:outputLabel for="to" value="To:" />
      <h:inputText id="to" value="#{scheduleBean.event.endDate}" />
            <f:convertDateTime pattern="dd/MM/yyyy" />
      </h:inputText>

      <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();"/>
    </h:panelGrid>
  </p:scheduleEventDialog>
</p:schedule>
```

As this schedule is editable, a ScheduleEvent needs to be added to the backing bean to bind with the dialog. Additionally simple addEvent method is added that will be processed when save button in dialog is clicked. This simply syncs the event represented in dialog with ScheduleModel.

```
public class ScheduleBean {

    private ScheduleModel<ScheduleEvent> model;

    private ScheduleEventImpl event = new DefaultScheduleEvent();

    public ScheduleBean() {
        eventModel = new ScheduleModel<ScheduleEvent>();
        eventModel.addEvent(new DefaultScheduleEvent(”title”, new Date(),
        new Date()));
    }

    public ScheduleModel<ScheduleEvent> getModel() {
        return model;
    }

    public ScheduleEventImpl getEvent() {
        return event;
    }
    public void setEvent(ScheduleEventImpl event) {
        this.event = event;
    }

    public void addEvent(ActionEvent actionEvent) {
        if(event.getId() == null)
            eventModel.addEvent(event);
        else
            eventModel.updateEvent(event);

        event = new DefaultScheduleEvent();
    }
}
```

Selecting an Event

EventSelectListener is invoked each time an event is clicked.

```
<p:schedule value=”#{scheduleBean.model}” editable=”true”
    eventSelectListener=”#{scheduleBean.onEventSelect}”>
    ...
</p:schedule>
```

```
public void onEventSelect(ScheduleEntrySelectEvent selectEvent) {
      event = (ScheduleEventImpl) selectEvent.getScheduleEvent();
}
```

onSelectEvent 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, if defined scheduleEventDialog is updated by default.

Selecting a Date

DateSelectListener is similar to EventSelectedListener however it 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}">
      ...
</p:schedule>
```

```
public void onDateSelect(DateSelectEvent selectEvent) {
      event = new DefaultScheduleEvent();
      event.setStartDate(selectEvent.getDate());
      event.setEndDate(selectEvent.getDate());
}
```

Optionally schedule has *onDateSelectEventUpdate* option to update any other component (s) on page, if defined scheduleEventDialog is updated by default.

Moving an Event

Events can be dragged and dropped into new dates, to get notified about this user interaction with ajax, define a server side eventMoveListener.

```
<p:schedule value="#{scheduleBean.model}" editable="true"
      eventMoveListener="#{scheduleBean.onEventMove}">
      ...
</p:schedule>
```

```
public void onEventMove(ScheduleEntryMoveEvent selectEvent) {
      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 update moved event's start and end dates, the delta values are provided for information purposes.

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}">
      ...
</p:schedule>
```

```
public void onEventMove(ScheduleEntryResizeEvent selectEvent) {
      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. As a result when you try to update schedule like with a regular PrimeFacess 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 then updating with regular PPR. An example of this is demonstrated at editable schedule example, save button is calling *myschedule.update()* at oncomplete event handler.

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 can 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 everytime 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

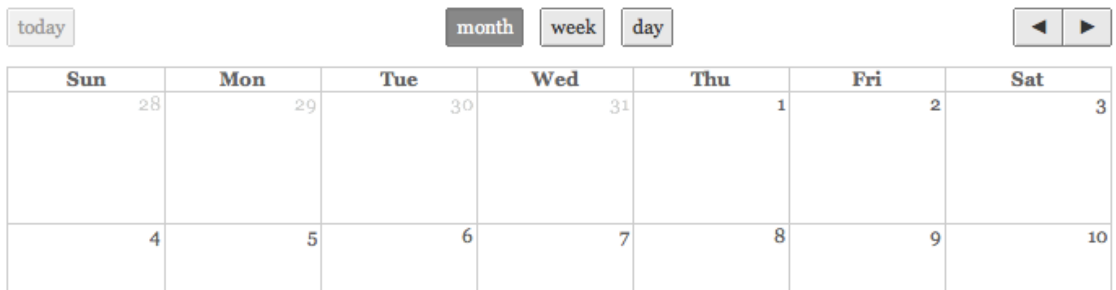
Header controls of Schedule can be customized based on templates, valid values of template options are;

- 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”
</p:schedule>
```

Output of this header will be as follows;

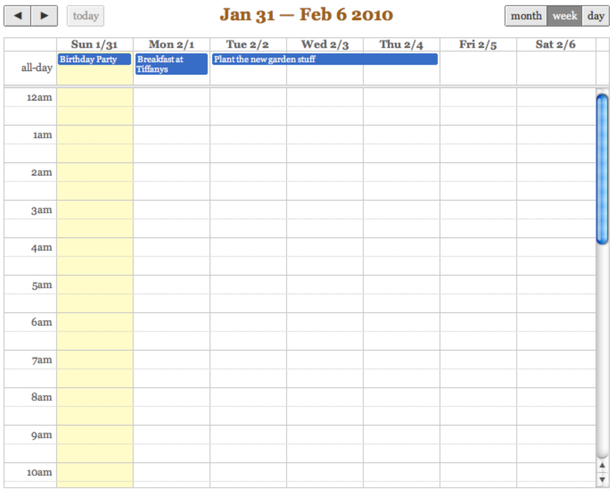


Views

5 different views are supported, these are “month”, “agendaWeek”, “agendaDay”, “basicWeek” and “basicDay”.

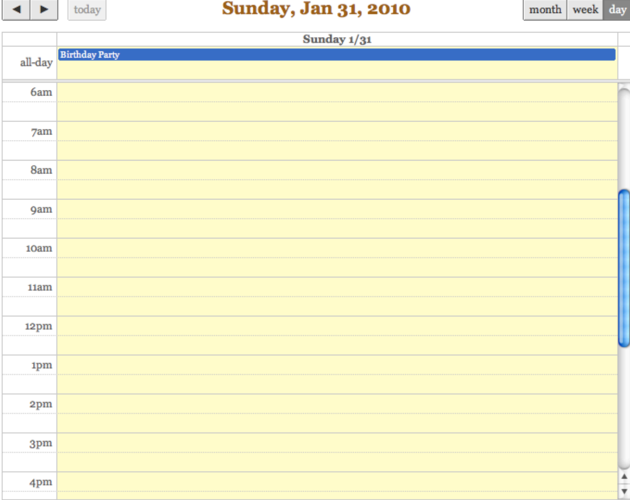
agendaWeek

```
<p:schedule value=”#{scheduleBean.model}” view=”agendaWeek”/>
```



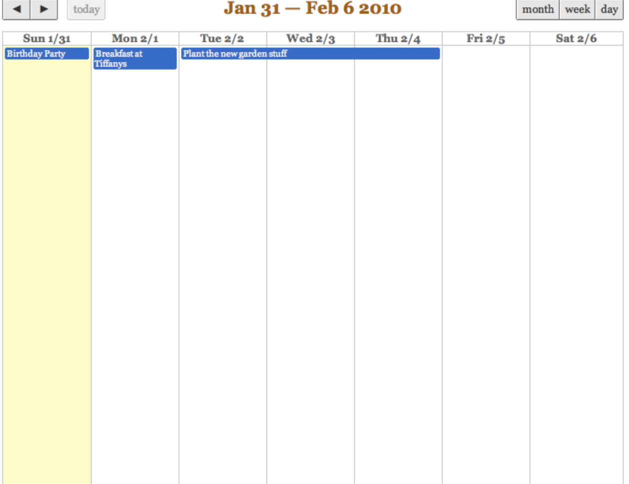
agendaDay

```
<p:schedule value="#{scheduleBean.model}" view="agendaDay"/>
```



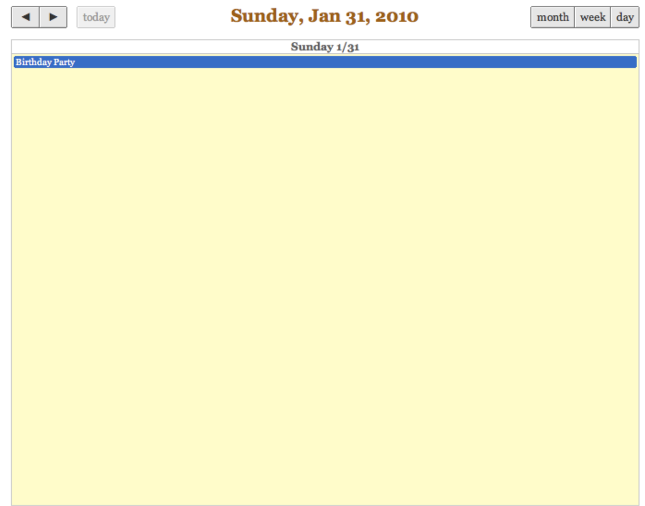
basicWeek

```
<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 overriden 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 javacript bundle object called PrimeFaces.widget.ScheduleResourceBundle. You can easily customize this object to add more languages in your application.

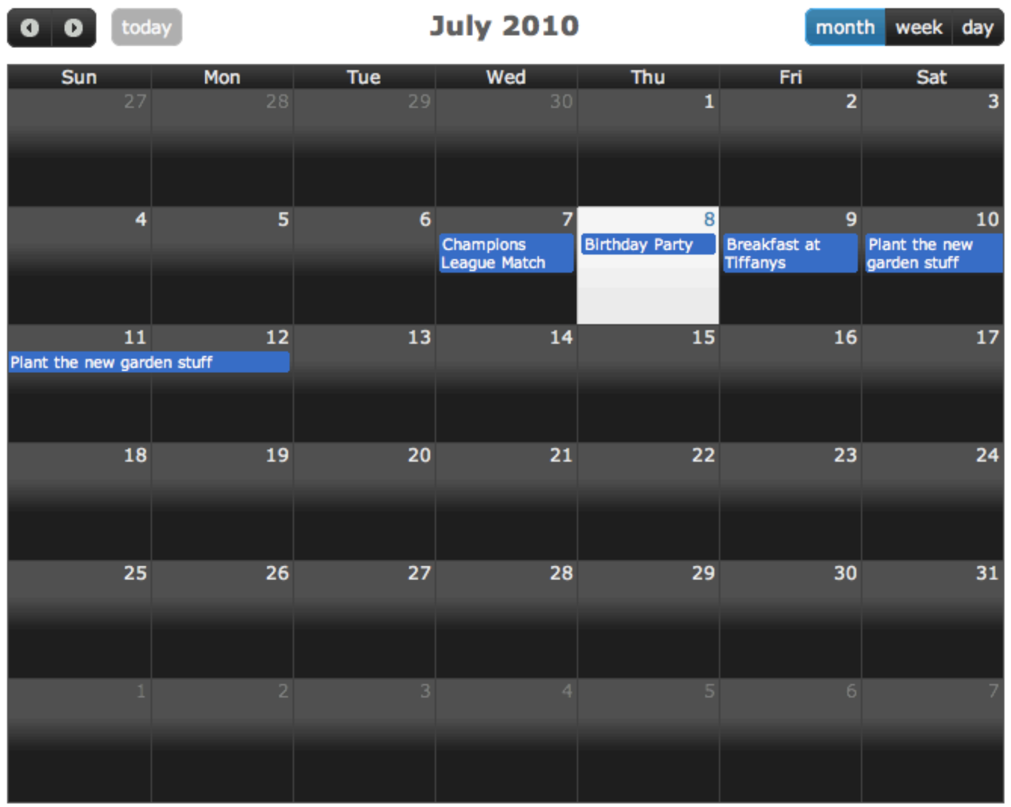
```
<script type="text/javascript">
    PrimeFaces.widget.ScheduleResourceBundle['key'] = {
      monthsNameShort:[],
      monthNames: [],
      dayNamesShort: [],
      today: "",
      month: "",
      week : "",
      day : "",
      allDayText : ""
    };
</script>
```

Make sure to execute this script block after the Schedule is initialized, ideal time would be when document is ready.

Skinning

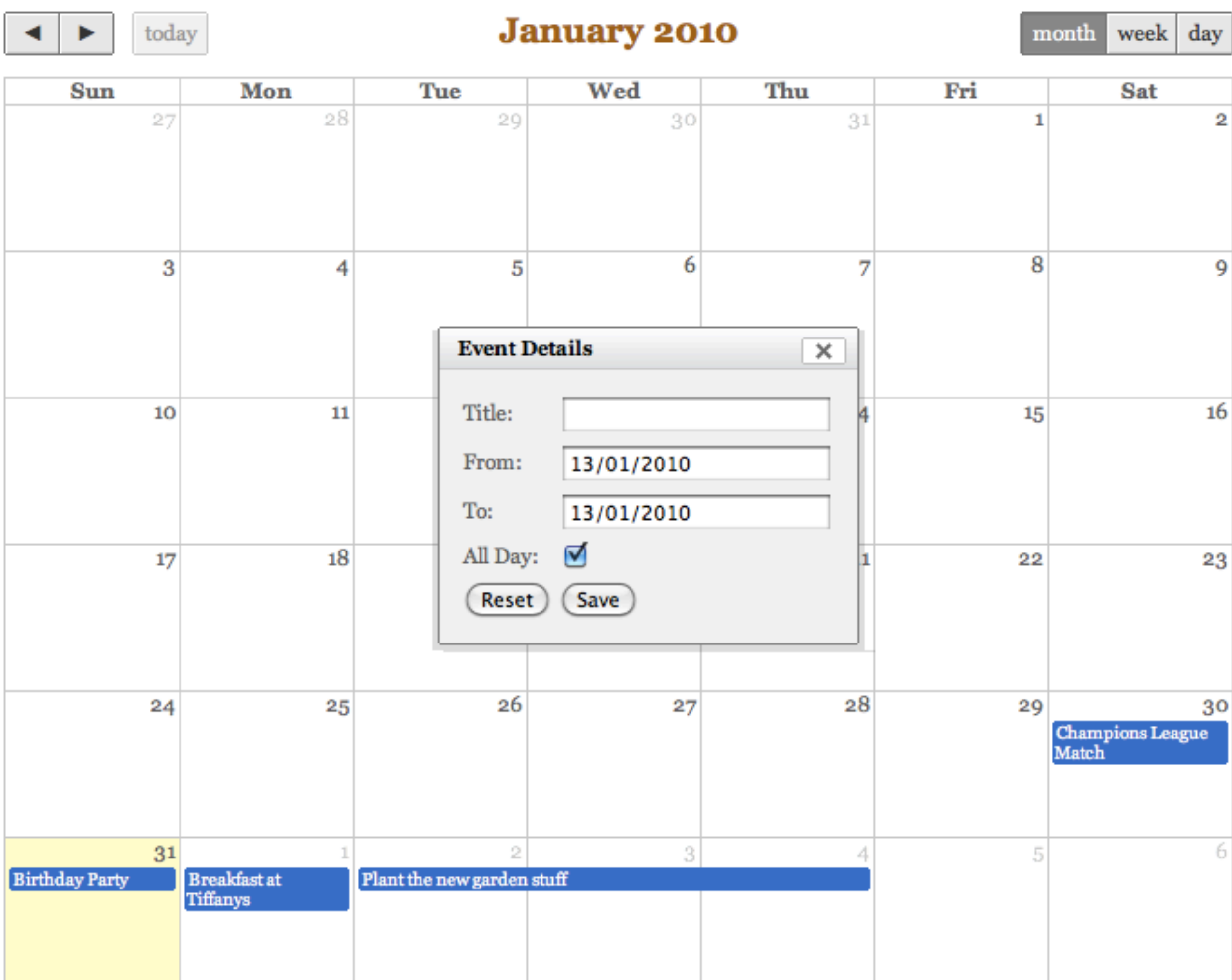
Schedule resides in a main container which *style* and *styleClass* attributes apply.

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



3.68 ScheduleEventDialog

ScheduleEventDialog is a customizable dialog specific to the Schedule component. Implementors need to create the UI of dialog content.



Info

Tag	scheduleEventDialog
Tag Class	org.primefaces.component.schedule.ScheduleEventDialogTag
Component Class	org.primefaces.component.schedule.ScheduleEventDialog
Component Type	org.primefaces.component.ScheduleEventDialog
Component Family	org.primefaces
Renderer Type	org.primefaces.component.ScheduleEventDialogRenderer
Renderer Class	org.primefaces.component.schedule.ScheduleEventDialogRenderer

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
header	null	String	Title of the dialog

Getting started with scheduleEventDialog

See schedule component section for more information regarding the usage of scheduleEventDialog.

3.69 Sheet

Sheet is used with spreadsheet component to display a set of data.

Info

Tag	sheet
Tag Class	org.primefaces.component.spreadsheet.SheetTag
Component Class	org.primefaces.component.spreadsheet.Sheet
Component Type	org.primefaces.component.Sheet
Component Family	org.primefaces.component

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	String	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
title	null	String	Title text of the sheet

Getting started with Sheet

Please see Spreadsheet section to find out how a sheet is used.

3.70 Slider

Slider is an input component with various customization options like horizontal&vertical sliding, display modes and skinning.

Info

Tag	slider
Tag Class	org.primefaces.component.slider.SliderTag
Component Class	org.primefaces.component.slider.Slider
Component Type	org.primefaces.component.Slider
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.SliderRenderer
Renderer Class	org.primefaces.component.slider.SliderRenderer

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
for	null	String	Id of the input text that the slider will be used for
display	null	String	Id of the component to display the slider value.
minValue	0	Integer	Minimum value of the slider
maxValue	100	Integer	Maximum value of the slider
style	null	String	Inline style of the container element
styleClass	null	String	Style class of the container element
animate	TRUE	Boolean	Boolean value to enable/disable the animated move when background of slider is clicked

Name	Default	Type	Description
type	horizontal	String	Sets the type of the slider, “horizontal” or “vertical”.
step	1	Integer	Fixed pixel incerements that the slider move in
disabled	FALSE	Boolean	Disables or enables the slider.

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.

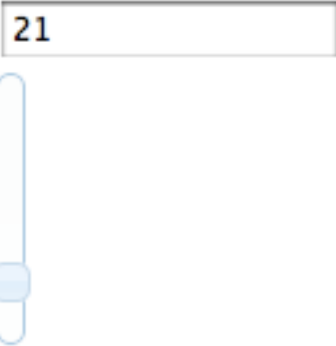
```
<h:inputHidden id=”number” value=”#{sliderBean.number}” />
<h:outputText value=”Set ratio to %” />
<h:outputText id=”output” value=”#{sliderBean.number}” />

<p:slider for=”number” display=”output” />
```

Set ratio to %21

Vertical Slider

By default slider works horizontally, 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 adn it can be customized with *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 of 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"/>
```

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;

3.71 Spinner

Spinner is an input component to provide a numerical input via increment and decrement buttons.

Info

Tag	**spinner**
Tag Class	**org.primefaces.component.spinner.SpinnerTag**
Component Class	**org.primefaces.component.spinner.Spinner**
Component Type	**org.primefaces.component.Spinner**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.SpinnerRenderer**
Renderer Class	**org.primefaces.component.spinner.SpinnerRenderer**

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 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	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	MethodExpression	A method binding expression that refers to a method validationg the input
valueChangeListener	null	MethodExpression	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.
stepFactor	1	Double	Stepping factor for each increment and decrement
min	null	Double	Minimum bounday 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	Html accesskey attribute
alt	null	String	Html alt attribute
dir	null	String	Html dir attribute
disabled	FALSE	Boolean	Html disabled attribute
lang	null	String	Html lang attribute
maxlength	null	Integer	Html maxlength attribute
onblur	null	String	Html onblur attribute
onchange	null	String	Html onchange attribute

Name	Default	Type	Description
onclick	null	String	Html onclick attribute
ondblclick	null	String	Html ondblclick attribute
onfocus	null	String	Html onfocus attribute
onkeydown	null	String	Html onkeydown attribute
onkeypress	null	String	Html onkeypress attribute
onkeyup	null	String	Html onkeyup attribute
onmousedown	null	String	Html onmousedown attribute
onmousemove	null	String	Html onmousemove attribute
onmouseout	null	String	Html onmouseout attribute
onmouseover	null	String	Html onmouseover attribute
onmouseup	null	String	Html onmouseup attribute
readonly	FALSE	Boolean	Html readonly attribute
size	null	Integer	Html size attribute
style	null	String	Html style attribute
styleClass	null	String	Html styleClass attribute
tabindex	null	Integer	Html tabindex attribute
title	null	String	Html title attribute

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 doubles so the fractional part can be controller with spinner as well. For doubles use the stepFactor attribute to specify stepping amount. Following example uses a stepFactor 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;

In case an increment happens a couple of times.

Prefix and Suffix

Prefix and Suffix options enable placing strings on input field. Note that you might 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>
```

$21

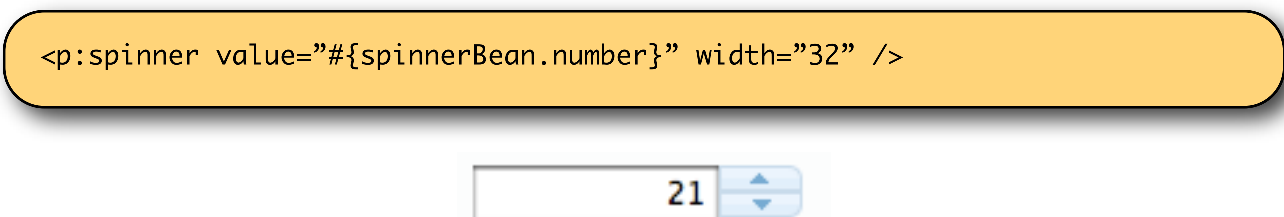
Boundaries

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" />
```

Skinning

Spinner consists of an input field and two buttons, input field can be 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;

3.72 Submenu

Submenu is nested in a menu component and represents a navigation group.

Info

Tag	**submenu**
Tag Class	**org.primefaces.component.submenu.SubmenuTag**
Component Class	**org.primefaces.component.submenu.Submenu**
Component Type	**org.primefaces.component.Submenu**
Component Family	**org.primefaces.component**

Attributes

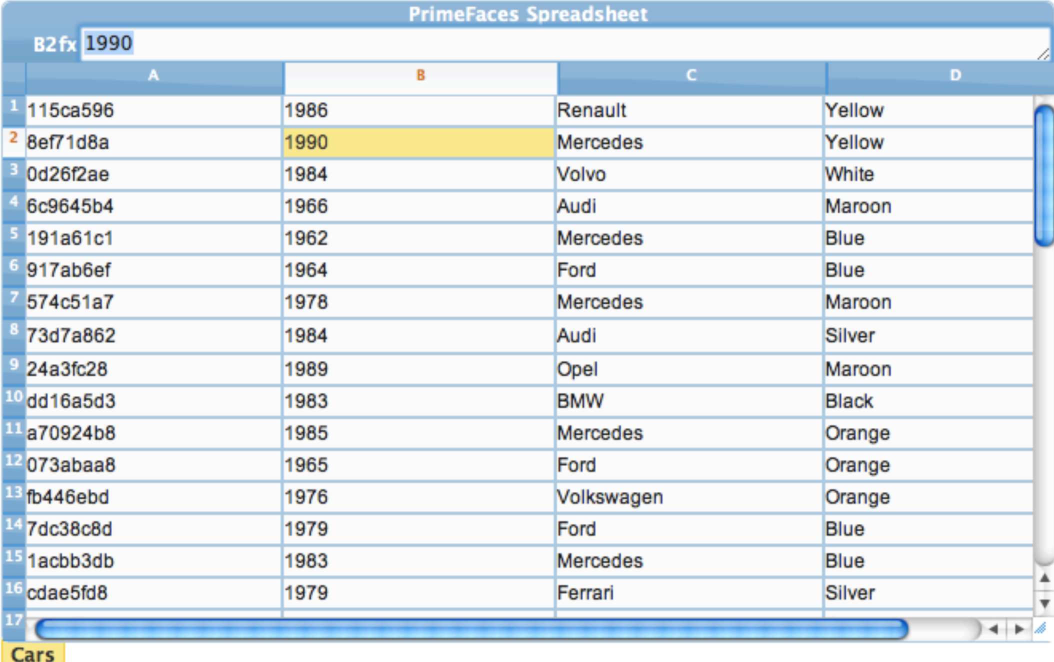
Name	Default	Type	Description
id	Assigned by JSF	String	Unique identifier of the component.
rendered	TRUE	boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance in a backing bean.
label	null	String	Label of the submenu header.
labelStyle	null	String	Style of the submenu label
labelStyleClass	null	String	Style class of the submenu label

Getting started with Submenu

Please see Menu or Menubar section to find out how submenu is used with the menus.

3.73 Spreadsheet

Spreadsheet is a data component with Excel like inline data editing features.



Info

Tag	spreadsheet
Tag Class	org.primefaces.component.spreadsheet.SpreadsheetTag
Component Class	org.primefaces.component.spreadsheet.Spreadsheet
Component Type	org.primefaces.component.Spreadsheet
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.SpreadsheetRenderer
Renderer Class	org.primefaces.component.spreadsheet.SpreadsheetRenderer

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.

Name	Default	Type	Description
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.
title	null	String	Title text of the spreadsheet.
editable	TRUE	Boolean	Defines if editing of cells are editable.
columnWidth	null	Integer	Defines commin column width in pixels.

Getting started with Spreadsheet

A spreadsheet can have one or more sheets. A sheet is a data component and it's usage is similar to a datatable.

We will be using a list of cars to display throughout the spreadsheet 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 a sheet to the car list.

```
public class CarBean {

    private List<Car> cars;

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

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

```
<p:spreadsheet widgetVar="ss" title="PrimeFaces Spreadsheet">
     <p:sheet var="car" value="#{tableBean.cars}" title="Cars">
           <p:column>
                 <h:inputText value="#{car.model}" />
           </p:column>

           <p:column>
                 <h:inputText value="#{car.year}" />
           </p:column>

           <p:column>
                 <h:inputText value="#{car.manufacturer}" />
           </p:column>

           <p:column>
                 <h:inputText value="#{car.color}" />
           </p:column>
     </p:sheet>
</p:spreadsheet>
```

Important point is that column content of each sheet column must be an input text.

Multiple Sheets

Any number of sheets can be displayed using spreadsheet and each sheet can display a different set of data.

```
<p:spreadsheet widgetVar="ss" title="PrimeFaces Spreadsheet">
     <p:sheet>
     </p:sheet>

     <p:sheet>
     </p:sheet>
     ...
</p:spreadsheet>
```

Saving Data

When saving the edited data displayed with spreadsheet, *save()* function of the client side api needs to be called when submitting the form.

```
<p:spreadsheet widgetVar="ss" title="PrimeFaces Spreadsheet">
     ...
</p:spreadsheet>

<p:commandButton onclick=”ss.save()” value=”Save” />
```

Readonly

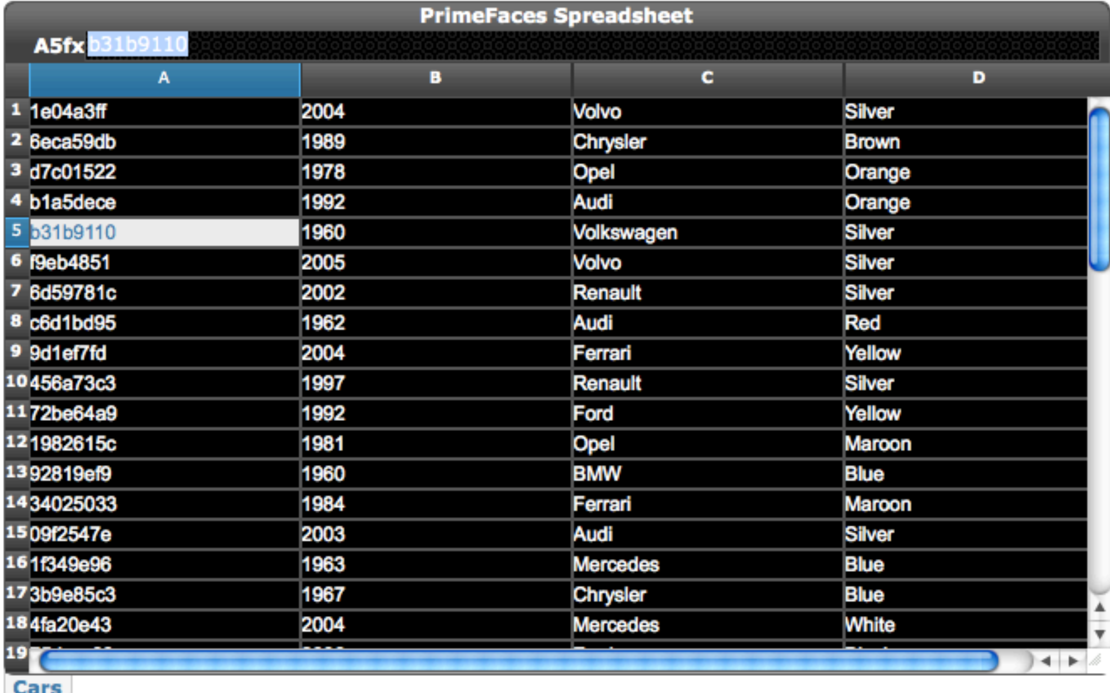
If you'd like to restrict editing on spreadsheet, set *editable* option to false. With this setting spreadsheet will be displayed as readonly.

```
<p:spreadsheet widgetVar="ss" editable="false">
    ...
</p:spreadsheet>
```

Skinning

Skinning CSS selectors of Spreadsheet are global, you can find more information at the main skinning documentation.

As an example, here is a spreadsheet based on a different theme;



	A	B	C	D
1	1e04a3ff	2004	Volvo	Silver
2	6eca59db	1989	Chrysler	Brown
3	d7c01522	1978	Opel	Orange
4	b1a5dece	1992	Audi	Orange
5	b31b9110	1960	Volkswagen	Silver
6	f9eb4851	2005	Volvo	Silver
7	6d59781c	2002	Renault	Silver
8	c6d1bd95	1962	Audi	Red
9	9d1ef7fd	2004	Ferrari	Yellow
10	456a73c3	1997	Renault	Silver
11	72be64a9	1992	Ford	Yellow
12	1982615c	1981	Opel	Maroon
13	92819ef9	1960	BMW	Blue
14	34025033	1984	Ferrari	Maroon
15	09f2547e	2003	Audi	Silver
16	1f349e96	1963	Mercedes	Blue
17	3b9e85c3	1967	Chrysler	Blue
18	4fa20e43	2004	Mercedes	White

3.74 Stack

Stack is a navigation component that mimics the stacks feature in Mac OS X.

Info

Tag	stack
Tag Class	org.primefaces.component.stack.StackTag
Component Class	org.primefaces.component.stack.Stack
Component Type	org.primefaces.component.Stack
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.StackRenderer
Renderer Class	org.primefaces.component.stack.StackRenderer

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.
icon	null	String	An optional image to contain stacked items.

Name	Default	Type	Description
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 *.pf-stack.* Override this style to change the location of stack.

```
.pf-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.

3.75 Tab

Tab is a generic container component used by other PrimeFaces components such as tabView or accordionPanel.

Info

Tag	**tabView**
Tag Class	**org.primefaces.component.tabview.TabTag**
Component Class	**org.primefaces.component.TabView.Tab**
Component Type	**org.primefaces.component.Tab**
Component Family	**org.primefaces.component**

Attributes

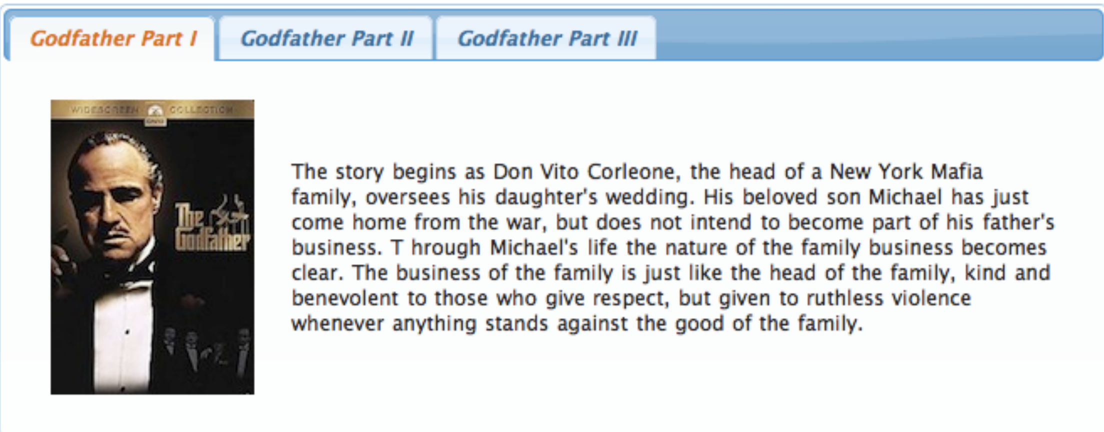
Name	Default	Type	Description
id	Assigned by JSF	String	Unique identifier of the component.
rendered	TRUE	Boolean	Boolean value to specify the rendering of the component, when set to false component will not be rendered.
binding	null	Object	An el expression that maps to a server side UIComponent instance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

3.76 TabView

TabView is a tabbed panel component featuring client side tabs, dynamic content loading with ajax and content transition effects.



Info

Tag	**tabView**
Tag Class	**org.primefaces.component.tabview.TabViewTag**
Component Class	**org.primefaces.component. tabview.TabView**
Component Type	**org.primefaces.component.TabView**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.TabViewRenderer**
Renderer Class	**org.primefaces.component.tabview.TabViewRenderer**

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.
widgetVar	null	String	Variable name of the client side widget.
activeIndex	0	Integer	Index of the active tab.

Name	Default	Type	Description
effect	null	String	Name of the transition effect.
effectDuration	null	String	Duration of the transition effect.
dynamic	FALSE	Boolean	Specifies the toggleMode.
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.

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 the choices for effect to use. EffectDuration specifies the speed of the effect, “*slow*”, “*normal*” (default) and “*fast*” are the valid options.

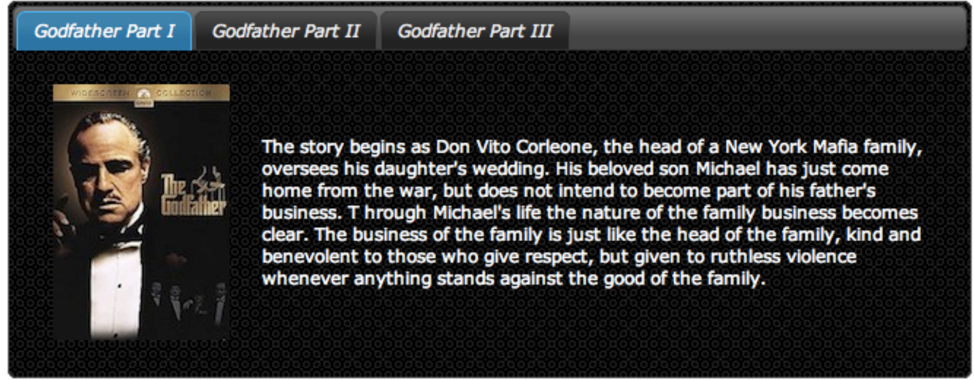
```
<p:tabView effect=”opacity” effectDuration=”fast”>
      //tabs
</p:tabView>
```

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;


Godfather Part I
Godfather Part II
Godfather Part III
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. T hrough 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.

3.77 Terminal

Terminal is an ajax powered web based terminal that brings desktop terminals to JSF.

```
Terminal
Welcome to PrimeFaces Terminal, how are you today?
prime $ date
Sun Nov 08 18:23:58 UTC 2009
prime $ greet Optimus
Hello Optimus
prime $ lss
lss not found
prime $ |
```

Info

Tag	terminal
Tag Class	org.primefaces.component.terminal.TerminalTag
Component Class	org.primefaces.component.terminal.Terminal
Component Type	org.primefaces.component.Terminal
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.TerminalRenderer
Renderer Class	org.primefaces.component.terminal.TerminalRenderer

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
width	null	String	Width of the terminal
height	null	String	Height of the terminal
welcomeMessage	null	String	Welcome message to be displayed on initial terminal load.
prompt	prime $	String	Primary prompt text.
commandHandler	null	javax.el.MethodExpression	Method to be called with arguments to process.
widgetVar	null	String	Javascript variable name of the wrapped widget

Getting started with the Terminal

A command handler is necessary to interpret commands entered in terminal.

```
<p:terminal commandHandler=”#{terminalBean.handleCommand}” />
```

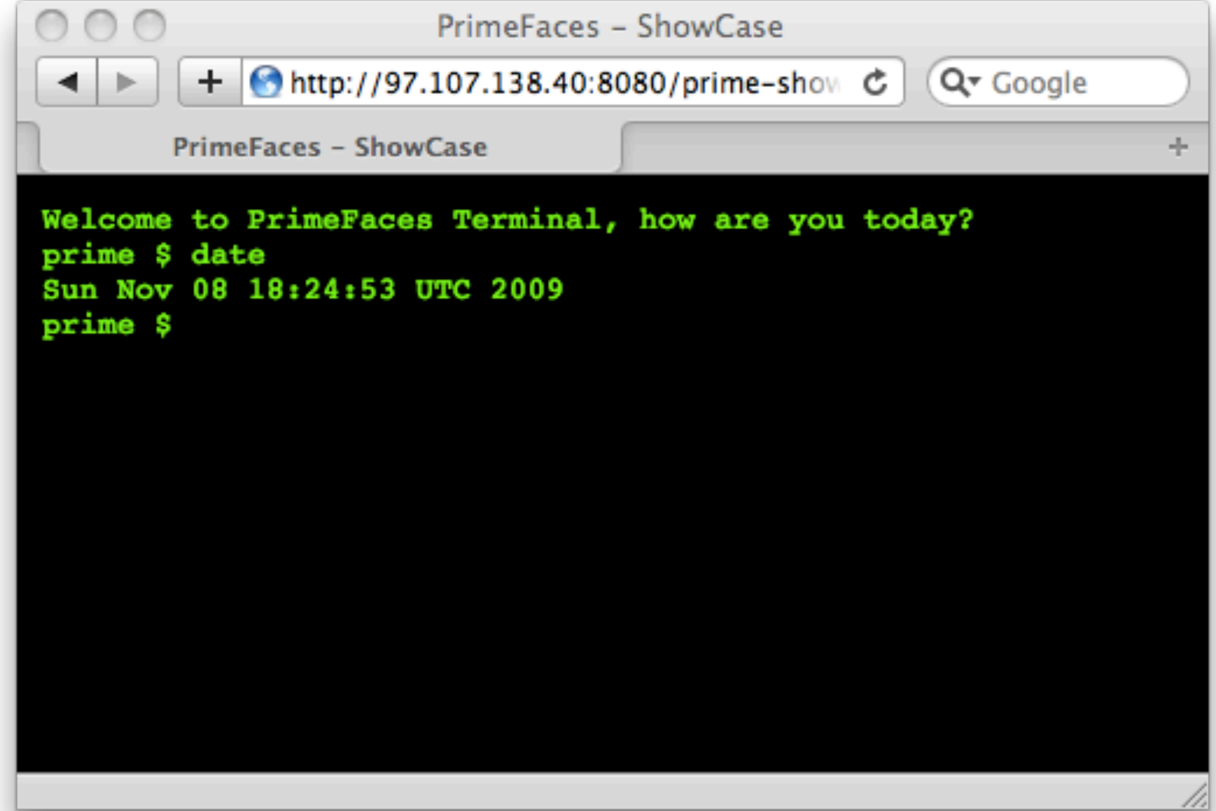
```
public class TerminalBean {

     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.

Full Screen Terminal

Setting width and height to 100% and placing the terminal as a direct child of body element is enough to create a full page terminal.

```
<body>
    <p:terminal width=”100%” height=”100%” />
</body>
```



3.78 ThemeSwitcher

ThemeSwitcher enables switching PrimeFaces themes on the fly with no page refresh.

Info

Tag	themeSwitcher
Tag Class	org.primefaces.component.themeswitcher.ThemeSwitcherTag
Component Class	org.primefaces.component.themeswitcher.ThemeSwitcher
Component Type	org.primefaces.component.ThemeSwitcher
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.ThemeSwitcherRenderer
Renderer Class	org.primefaces.component.themeswitcher.ThemeSwitcherRenderer

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

Name	Default	Type	Description
theme	null	String	Width of the terminal
width	150	Integer	Height of the terminal
height	200	Integer	Welcome message to be displayed on initial terminal load.
buttonHeight	14	Integer	Primary prompt text.
initialText	Switch Theme	String	Method to be called with arguments to process.
buttonPreText	Theme:	String	Javascript variable name of the wrapped 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 />
```

3.79 Tooltip

Tooltip goes beyond the legacy html title attribute by providing custom effects, events, html content and advance skinning support.

Info

Tag	**tooltip**
Tag Class	**org.primefaces.component.tooltip.TooltipTag**
Component Class	**org.primefaces.component.tooltip.Tooltip**
Component Type	**org.primefaces.component.Tooltip**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.TooltipRenderer**
Renderer Class	**org.primefaces.component.tooltip.TooltipRenderer**

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 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
widgetVar	null	String	Javascript variable name of client side tooltip object.

Name	Default	Type	Description
global	FALSE	boolean	A global tooltip converts each title attribute to a tooltip.
targetPosition	bottomRig ht	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.
showEvent	mouseov er	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.
style	blue	String	Name of the skinning theme.
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 it's 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"/>
```

Events

A tooltip is shown on mouseover event and hidden when mouse is out. 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="blue"/>
```

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>
```

Skinning Tooltip

Tooltip supports built-in themes, default theme is blue. Here’s the list of supported themes.

- blue
- cream
- dark
- green
- light
- red

If you need to create your own style rather than using the built-on ones, use the style configuration. Just like styling the charts provide your options with a custom javascript object.

```
<script type=”text/javascript”>
var custom = {
      width: 200,
      padding: 5,
      background: '#A2D959',
      color: 'black',
      textAlign: 'center',
      border: {
            width: 7,
            radius: 5,
            color: '#A2D959'
      },
      tip: 'topLeft',
      name: 'dark'
};
</script>
```

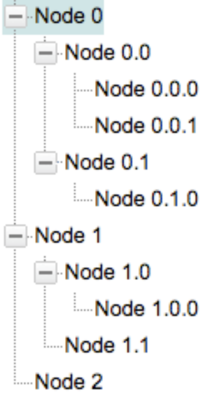
```
<h:outputLink id=”lnk” value=”#”>
   <h:outputText value=”Custom” />
</h:inputSecret>

<p:tooltip for=”lnk” value=”Tooltip with custom style” style=”custom”/>
```

Custom

Tooltip with custom style

3.80 Tree

Tree is is used for displaying hierarchical data or creating site navigations.



Info

Tag	**tree**
Tag Class	**org.primefaces.component.tree.TreeTag**
Component Class	**org.primefaces.component.tree.Tree**
Component Type	**org.primefaces.component.Tree**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.TreeRenderer**
Renderer Class	**org.primefaces.component.tree.TreeRenderer**

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	A TreeNode instance as the backing model.

Name	Default	Type	Description
var	null	String	Name of the request-scoped variable that'll be used to refer each treenode data during rendering
dynamic	FALSE	Boolean	Specifies the ajax/client toggleMode
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	javax.el.Method Expression	Method expression to listen node select events
nodeExpandListener	null	javax.el.Method Expression	Method expression to listen node expand events
nodeCollapseListener	null	javax.el.Method Expression	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

Name	Default	Type	Description
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 expaned 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

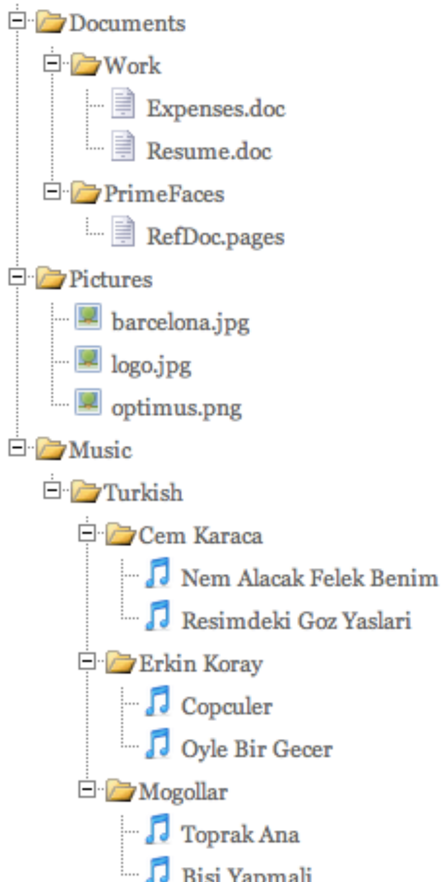
When toggling is set to client all the treenodes in model are rendered to the client and tree is created, this mode is suitable for relatively small datasets and provides fast user interaction. On the otherhand it's not suitable for large data since all the data is sent to the client.

Dynamic

Dynamic mode uses ajax to fetch the treenodes from server side on deman, 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>
```

Tree Node Styles

```
 .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 }
```

DocumentBean

```
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;
      }
}
```

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 particalar tree at runtime.

Selection Modes

Node selection is a built-in feature of tree and it supports three different modes.

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>
```

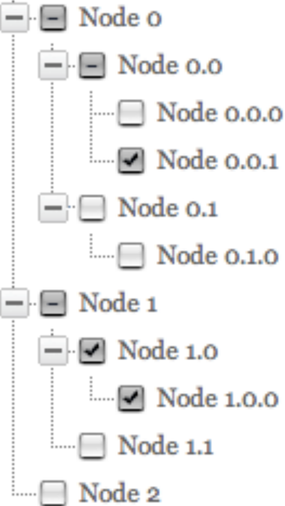
Selection should be an array of TreeNodes, tree will find the selected nodes and assign them to your selection model.

```
public class TreeBean {

      private TreeNode root;

      private TreeNode[] selectedNodes;

      public TreeBean() {
            root = new TreeNode("Root", null);
            //populate nodes
      }

      public TreeNode getRoot() {
            return root;
      }

      public TreeNode[] getSelectedNodes() {
            return selectedNodes;
      }

      public void setSelectedNodes(TreeNode[] selectedNodes) {
            this.selectedNodes = selectedNodes;
      }
}
```

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 tree looks like;



That's it, 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.

> 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 Tree

Treeview has certain css selectors for nodes, for full list selectors visit;

http://developer.yahoo.com/yui/treeview/#style

Skinning example below demonstrates a sample navigation menu implementation.

```
 .ygtvtn {
  background:transparent none repeat scroll 0 0;
  height:20px;
  width:1em;
}
.ygtvtm {
  background:transparent uri(sprite-menu.gif) no-repeat scroll -8px 2px;
  height:20px;
  width:1em;
}
.ygtvtmh {
  background:transparent uri(sprite-menu.gif) no-repeat scroll -8px -77px;
  height:20px;
  width:1em;
}
.ygtvtp {
  background:transparent uri(sprite-menu.gif) no-repeat scroll -8px -315px;
  height:20px;
  width:1em;
}
.ygtvtph {
  background:transparent uri(sprite-menu.gif) no-repeat scroll -8px -395px;
  height:20px;
  width:1em;
}
.ygtvln {
  background:transparent none repeat scroll 0 0;
  height:20px;
  width:1em;
}
.ygtvlm {
  background:transparent uri(sprite-menu.gif) no-repeat scroll -8px 2px;
  height:20px;
  width:1em;
}
.ygtvlmh {
  background:transparent uri(sprite-menu.gif) no-repeat scroll -8px -77px;
  height:20px;
  width:1em;
}
.ygtvlp {
  background:transparent uri(sprite-menu.gif) no-repeat scroll -8px -315px;
  height:20px;
  width:1em;
}
.ygtvlph {
   background:transparent uri(sprite-menu.gif) no-repeat scroll -8px -395px;
   height:20px;
   width:1em;
}
```

```
 .ygtvdepthcell {
   background:transparent none repeat scroll 0 0;
   height:20px;
   width:1em;
   cursor:default;
 }
.ygtvln, .ygtvtn {
   cursor:default;
}
```

- Node 0
  - Node 0.0
    - Node 0.0.0
    - Node 0.0.1
  - Node 0.1
    - Node 0.1.0
- Node 1
  - Node 1.0
    - Node 1.0.0
  - Node 1.1
- Node 2

3.81 TreeNode

TreeNode is used with Tree component to represent a node in tree.

Info

Tag	**treeNode**
Tag Class	**org.primefaces.component.tree.UITreeNodeTag**
Component Class	**org.primefaces.component.tree.UITreeNode**
Component Type	**org.primefaces.component.UITreeNode**
Component Family	**org.primefaces.component**

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
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.

3.82 TreeTable

Treet is 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	treeTable
Tag Class	org.primefaces.component.treetable.TreeTableTag
Component Class	org.primefaces.component.treetable.TreeTable
Component Type	org.primefaces.component.TreeTable
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.TreeTableRenderer
Renderer Class	org.primefaces.component.treetable.TreeTableRenderer

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	A TreeNode instance as the backing model.

Name	Default	Type	Description
var	null	String	Name of the request-scoped variable used to refer each treenode.
style	null	String	Inline style of the container element.
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.

```
public class DocumentsController {

  private TreeNode root;

  public DocumentsController() {
    root = new TreeNode(”root”, null);

    TreeNode documents = new DefaultTreeNode(”Documents”, root);
    TreeNode pictures = new DefaultTreeNode(”Pictures”, root);
    TreeNode music = new DefaultTreeNode(”Music”, root);

    TreeNode work = new DefaultTreeNode(”Work”, documents);
    TreeNode primefaces = new DefaultTreeNode(”PrimeFaces”, documents);

    //Documents
    TreeNode expenses = new DefaultTreeNode(”document”, “Expenses.doc”, work);
    TreeNode resume = new DefaultTreeNode(”document”, “Resume.doc”, work);
    TreeNode refdoc = new DefaultTreeNode(”document”, “RefDoc.pages”, primefaces);

    //Pictures
    TreeNode barca = new DefaultTreeNode(”picture”, “barcelona.jpg”, pictures);
    TreeNode primelogo = new DefaultTreeNode(”picture”, “logo.jpg”, pictures);
    TreeNode optimus = new DefaultTreeNode(”picture”, “optimus.png”, pictures);

    //Music
    TreeNode turkish = new TreeNode(”Turkish”, music);
   }

   public TreeNode getRoot() {
      return root;
   }
}
```

Usage on a page is very similar to a datatable;

```
<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>
```

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 Gozyaslari	2400 KB	Audio File
Erkin Koray	-	Folder
Copculer	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 structual 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	

3.83 UIAjax

UIAjax is a generic component that can enable ajax behavior on a regular JSF component. UIAjax is attached to a javascript event of it's parent.

UIAjax Classes

Tag	ajax
Tag Class	org.primefaces.component.uiajax.UIAjaxTag
Component Class	org.primefaces.component.uiajax.UIAjax
Component Type	org.primefaces.component.UIAjax
Component Family	org.primefaces.component
Renderer Type	org.primefaces.component.UIAjaxRenderer
Renderer Class	org.primefaces.component.uiajax.UIAjaxRenderer

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
event	null	String	Javascript event to attach the uiajax. Examples are "blur, keyup, click, etc"
action	null	javax.el.MethodExpression	A method expression that'd be processed in the partial request caused by uiajax.
actionListener	null	javax.faces.event.ActionListener	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.

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 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 UIAjax

UIAjax needs a parent and a javascript event of it's parent to work with.

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

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

In this example, each time the user types and releases the key, an ajax request is sent to the server and normal JSF lifecycle is executed. When the response is received uiajax partially updates the output text with id “out”.

Update is not mandatory, if you do not provide the update attribute, by default p:ajax updates it's parent form.

Note: Id attribute needs to be present because JSF implementations behave differently. While mojarra does not render the clientId of the inputText if id is not provided, myfaces does. UIAjax requires the clientId of it's parent at the rendered output.

UIAjax and ActionEvents

UIAjax extends from UICommand, this means it can execute action methods and actionListeners defined in a JSF backing bean. Following example executes an actionlistener each time keyup event occurs and counts the number of keyups.

```
<h:inputText id="counter">
    <p:ajax event="keyup" update="out"          actionListener="#
{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(ActionEvent actionEvent) {
        count++;
    }
}
```

UIAjax and Validations

A tricky example of uiajax is validations to mimic client side validation. Following example validates the inputtext on blur event of the input text being validated.

```
h:form prependId="false">
    <p:panel id="panel" header="New Person">
        <h:messages />
        <h:outputText value="5 characters minimum" />

        <h:panelGrid columns="3">
            <h:outputLabel for="firstname" value="Firstname: *" />
            <h:inputText id="firstname" value="#{pprBean.firstname}" >
                <f:validateLength minimum="5" />
                <p:ajax event="blur" update="panel" />
            </h:inputText>
            <h:message for="firstname" />
        </h:panelGrid>
    </p:panel>
</h:form>
```

3.84 Watermark

Watermark displayes a hint on an input field describing what the field is for.

Search with a keyword

Info

Tag	**watermark**
Tag Class	**org.primefaces.component.watermark.WatermarkTag**
Component Class	**org.primefaces.component.watermark.Watermark**
Component Type	**org.primefaces.component.Watermark**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.WatermarkRenderer**
Renderer Class	**org.primefaces.component.watermark.WatermarkRenderer**

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	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” />
```

forElement

Another way to define a watermark is to use a jQuery selector with the forElement feature. This would useful to deal with complex cases.

```
<p:calendar id=”date” value=”#{bean.date}” />
<p:watermark forElement=”#date input” value=”Choose a date” />
```

Form Submissions

Watermark is set as the text of an input field which shouldn’t be sent to the server when an enclosing for 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 Watermark

There’s only one css style class applying watermark which is ‘*.ui-watermark*’, you can override this class to bring in your own style.

3.85 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.

Info

Tag	**wizard**
Tag Class	**org.primefaces.component.wizard.WizardTag**
Component Class	**org.primefaces.component.wizard.Wizard**
Component Type	**org.primefaces.component.Wizard**
Component Family	**org.primefaces.component**
Renderer Type	**org.primefaces.component.WizardRenderer**
Renderer Class	**org.primefaces.component.wizard.WizardRenderer**

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
step	0	String	Id of the current step in flow
effect	fade	String	Effect to be used for switching between steps.
style	null	String	Style of the main wizard container element.

Name	Default	Type	Description
styleClass	null	String	Style class of the main wizard container element.
flowListener	null	MethodExpression	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.
widgetVar	null	String	Variable 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. For JSF 1.2 libraries like PrimeFaces optimus, Seam, MyFaces orchestra provides this scope.

```
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=”Adress 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: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 powered by 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.previous();”>Back</h:outputLink>
```

Navigators can also be customized using facets named *back* and *next*.

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 Events

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.

Skinning Wizard

Wizard can be easily customized in terms of styling with the use of style/styleClass attributes and the well defined CSS selectors. 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.

4. TouchFaces

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.

```
<i:view id=”home” title=”Home Page”>

     //content

</i:view>
```

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>
```

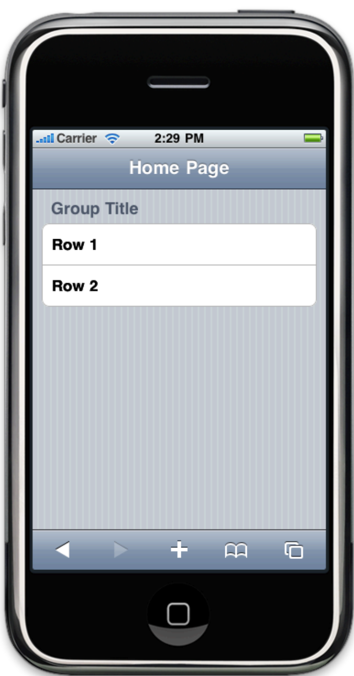
When you run this page, only the home view would be displayed, a view can be built with core JSF and components and TouchFaces specific components like tableView, rowGroups, rowItems and more.

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>
```

Output of this page would be;



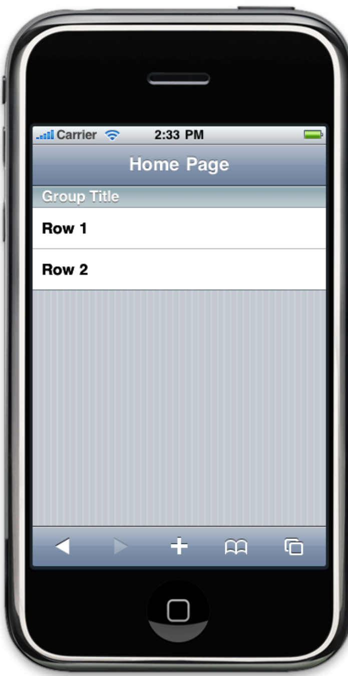
Group Display Modes

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

Default animation used when navigation to a view is “slide”. Additional effects are;

- slide
- slideup
- flip
- dissolve
- fade
- flip
- pop
- swap
- cube

```
<i:view id=”otherview” title=”Other view”>
     <f:facet name=”leftNavBar”>
           <i:navBarControl label=”Settings” view=”settings”
           effect=”flip”/>
     </f:facet>

     //view content

</i:view>
```

TouchFaces Navigation API

TouchFaces client side object provides two useful navigation metods;

- 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: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 (preferrably 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	application
Tag Class	org.primefaces.touch.component.application.ApplicationTag
Component Class	org.primefaces.touch.component.applicaiton.Application
Component Type	org.primefaces.touch.Application
Component Family	org.primefaces.touch
Renderer Type	org.primefaces.touch.component.ApplicationRenderer
Renderer Class	org.primefaces.touch.component.application.ApplicationRenderer

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
theme	null	String	Theme of the app, “default” or “dark”.
icon	null	String	Icon of the app.

4.6.2 NavBarControl

Info

Tag	navBarControl
Tag Class	org.primefaces.touch.component.navbarcontrol.NavBarControlTag
Component Class	org.primefaces.touch.component.navbarcontrol.NavBarControl
Component Type	org.primefaces.touch.NavBarControl
Component Family	org.primefaces.touch

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
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	rowGroup
Tag Class	org.primefaces.touch.component.rowgroup.RowGroupTag
Component Class	org.primefaces.touch.component.rowgroup.RowGroup
Component Type	org.primefaces.touch.RowGroup
Component Family	org.primefaces.touch
Renderer Type	org.primefaces.touch.component.RowGroupRenderer
Renderer Class	org.primefaces.touch.component.rowgroup.RowGroupRenderer

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
title	null	String	Optional title of the row group.

4.6.4 RowItem

Info

Tag	rowItem
Tag Class	org.primefaces.touch.component.rowitem.RowItemTag
Component Class	org.primefaces.touch.component.rowitem.RowItem
Component Type	org.primefaces.touch.RowItem
Component Family	org.primefaces.touch
Renderer Type	org.primefaces.touch.component.RowItemRenderer
Renderer Class	org.primefaces.touch.component.rowitem.RowItemRenderer

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
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	javax.el.MethodExpression	A method expression that'd be processed in the partial request caused by uiajax.
actionListener	null	javax.faces.event.ActionListener	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	switch
Tag Class	org.primefaces.touch.component.switch.SwitchTag
Component Class	org.primefaces.touch.component.switch.Switch
Component Type	org.primefaces.touch.Switch
Component Family	org.primefaces.touch
Renderer Type	org.primefaces.touch.component.SwitchRenderer
Renderer Class	org.primefaces.touch.component.switch.SwitchRenderer

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 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	MethodBindi ng	A method binding expression that refers to a method validatong the input

Name	Default	Type	Description
valueChangeListener	null	ValueChang eListener	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.

4.6.6 TableView

Info

Tag	tableView
Tag Class	org.primefaces.touch.component.tableview.TableView
Component Class	org.primefaces.touch.component.tableview.TableView
Component Type	org.primefaces.touch.TableView
Component Family	org.primefaces.touch
Renderer Type	org.primefaces.touch.component.TableViewRenderer
Renderer Class	org.primefaces.touch.component.tableview.TableViewRenderer

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

4.6.7 View

Info

Tag	view
Tag Class	org.primefaces.touch.component.view.ViewTag
Component Class	org.primefaces.touch.component.view.View
Component Type	org.primefaces.touch.View
Component Family	org.primefaces.touch
Renderer Type	org.primefaces.touch.component.ViewRenderer
Renderer Class	org.primefaces.touch.component.viewrenderer.ViewRenderer

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
title	regular	String	Optional title of the view.

5. Partial Rendering and Processing

PrimeFaces provides a partial rendering and view processing feature to enable choosing what to process in JSF lifecyle and what to render in the end.

5.1 Partial Rendering

In addition to components like autoComplete with built-in ajax capabilities, PrimeFaces also provides a generic PPR(Partial Page Rendering) mechanism to update any JSF component with an ajax request. Several components are equipped with the common PPR attributes (update, process, onstart, oncomplete, etc).

5.1.1 Infrastructure

PrimeFaces Ajax Framework follows a lightweight approach compared to other AJAX and JSF solutions. PrimeFaces uses only one artifact: a PhaseListener to bring in AJAX. We don't approach AJAX requests different than the regular requests. As a result there's no need for JSF extensions like AjaxViewRoot, AjaxStateManager, AjaxViewHandler, Servlet Filters, HtmlParsers and so on. PrimeFaces aims to keep it clean, fast and lightweight.

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 in 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. This might be the case when you are doing custom scripting on client side.

```
<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 (eg. two forms), you need to define explicit clientIds to update. PPR can handle requests that are triggered inside a namingcontainer that updates another namingcontainer.

```
<h:form id="form1">

    <p:commandButton update="form2:display" />

</h:form>

<h:form id="form2">

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

</h:form>
```

This is same as using naming container seperator char as the first character of id search expression so following would work as well;

```
<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 seperated 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

Plain HTML and JSP

When using JSP for JSF pages, PrimeFaces PPR has a limitation to update a component that contains plain html which means the html part will be ignored in partial response. This is only a case for JSP and does not happen with Facelets.

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, similary 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 actionListener="#{bean.populateSuburbs}"
            event="change" update="suburbs"/>
  </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.

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="cities"/>
  </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.

5.2.2 Keywords

Just like PPR, Partial processing also supports keywords.

Keyword	Description
@this	Component that triggers the PPR is processed.
@parent	Parent of the PPR trigger is processed.

Keyword	Description
@form	Encapsulating form of the PPR trigger is processed
@none	No component is processed.
@all	Whole component tree is processed just like a regular request.

Same city-suburb example can be written with keywords as well.

```
<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>
```

Important point to emphasize 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.

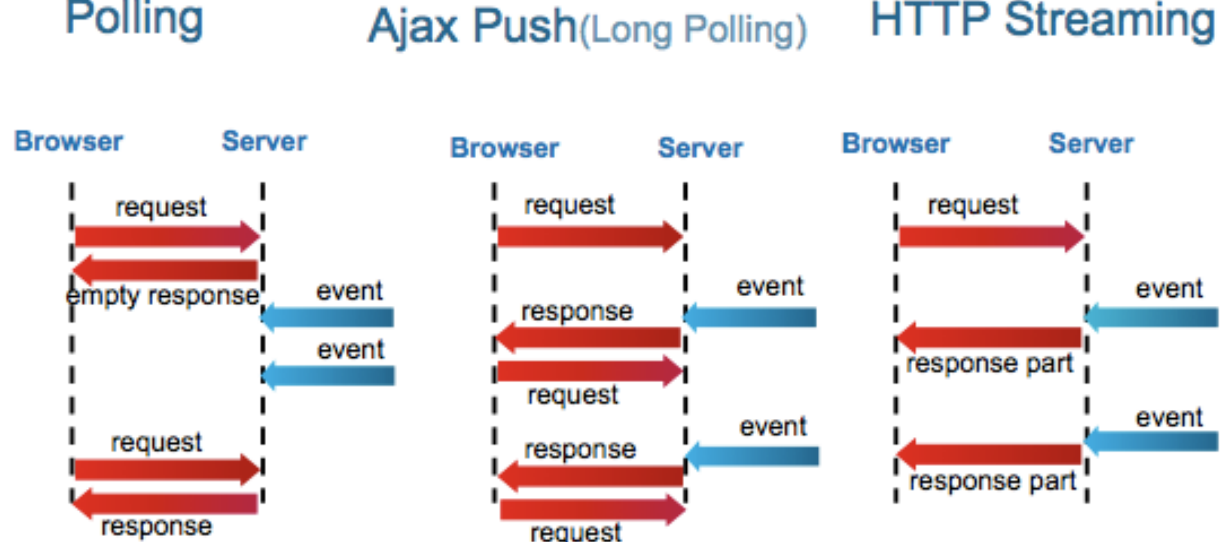
5.2.4 Ajax vs Non-Ajax

An important feature of partial process is being a generic solution for both ajax and non-ajax cases. Whether you are doing a ajax request or a regular ajax request that causes a full page submit, you can take advantage of partial processing. Example below demonstrates how to avoid executing page validations before navigating to another page. This is usually achieved with setting immediate to true in standard JSF. Compared to immediate, partial processing is much more flexible.

```
<h:form>
     //Components with validation constraints
     <p:commandButton action=”navigate” process=”@this” ajax=”false”/>
</h:form>
```

6. Ajax Push/Comet

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 with either long-polling or http-streaming. 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.AsycnListener.

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 Lead Cagatay Civici is also a committer of Atmosphere Framework and as a result PrimeFaces Push is powered by Atmosphere Runtime. PrimeFaces simplifies 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 accesible using *response.data*.

7. Javascript API

PrimeFaces renders unobstrusive javascript which cleanly seperates behavior from the html. There're two libraries we use, YUI for most of the widget controls and jQuery for ajax, dom manipulation plus some UI plugins.

YUI version is 2.8.r4 and jQuery version is 1.4.2, these are the latest versions at the time of writing.

7.1 PrimeFaces Global Object

PrimeFaces is the main javascript object providing utilities like onContentReady and more.

Method	Description
escapeClientId(id)	Escaped JSF ids with semi colon to work with jQuery selectors
onContentReady(id, fn)	Executes the fn callback function when a specific dom element with identifier "id" is ready when document is being loaded
addSubmitParam(parent, params)	Adds hidden request parameters dynamically
cleanWatermarks()	Watermark component extension, cleans all watermarks on page before submitting the form.
showWatermarks()	Show

7.2 Namespaces

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.

7.3 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.
onstart(xhr)	Javascript callback to process before sending the ajax request, return false to cancel the request. Takes xmlhttprequest as the parameter.
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 RequestContent 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.

Parameters

You can send any number of parameters as the third argument of request function, in addition there're some predefined parameters name that have a special meaning to PrimeFaces.

Name	Description
PrimeFaces.PARTIAL_UPDATE_PARAM	Component Id(s) to update
PrimeFaces.PARTIAL_SOURCE_PARAM	Component Id(s) to process

Examples

Suppose you have a JSF page called createUser.jsf 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'});
```

Adding a status callback is also easy using the configuration object;

```
PrimeFaces.ajax.AjaxRequest('/myapp/createUser.jsf',
                  {
                    formId:'userForm',
                    oncomplete:function(xhr, status) {alert('Done');}
                  });
```

You can pass as many parameters as you want using the parameters option.

```
PrimeFaces.ajax.AjaxRequest('/myapp/createUser.jsf',
                        {formId:'userForm'},
                        {
                          'param_name1':'value1',
                          'param_name2':'value2'
                        }
);
```

If you'd like to update a component with ajax, provide the id using the parameters option.

```
PrimeFaces.ajax.AjaxRequest('/myapp/createUser.jsf',
                        {formId:'userForm'},
                        {PrimeFaces.PARTIAL_UPDATE_PARAM:'username'}
);
```

Finally you can configure request to what to process and what to update. Example below processes createUserButton on the server side and update username component.

```
PrimeFaces.ajax.AjaxRequest('/myapp/createUser.jsf',
                        {formId:'userForm'},
                        {
                          PrimeFaces.PARTIAL_UPDATE_PARAM:'username',
                          PrimeFaces.PARTIAL_PROCESS_PARAM:'createUserButton'
                        }
);
```

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. Skinning

PrimeFaces is integrated with powerful ThemeRoller CSS Framework. Currently there are 28 pre-designed themes that you can preview and download from PrimeFaces theme gallery.

http://www.primefaces.org/themes.html

9.1 Applying a Theme

Applying a theme to your PrimeFaces project is very easy, once you download the zipped theme file from PrimeFaces Theme Gallery, extract the contents to a folder in your applications. A theme consists of a css file called skin.css and images that are located in images folder.

1) Add the skin.css file to your JSF page using link or h:outputStylesheet(JSF 2.0).

```
<link type=”text/css” rel=”stylesheet”
          href=”%PATH_TO_THEME_FOLDER%/skin.css” />
```

2) Configure PrimeFaces not to add it’s bundled default skin (sam).

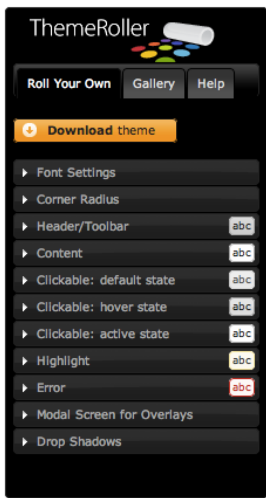
```
<context-param>
     <param-name>primefaces.skin</param-name>
     <param-value>none</param-value>
</context-param>
```

That’s it, as an example assume you want to use redmond theme which you’ve copied the theme contents to themes folder of your application.

```
<link type=”text/css” rel=”stylesheet”
     href=”#{request.contextPath}/themes/redmond/skin.css” />
```

9.2 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 tool for this. Once you have created your own theme the way you like follow the steps in installing themes section to start using it.



9.3 How Skinning Works

Powered by ThemeRoller, PrimeFaces seperates structual 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, colors, background images are some examples of type of selectors.

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-icon	An icon of a component

These classes are not aware of structural css like margins and paddings, mostly they only define colors. This clean seperation brings great flexibility in Skinning 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* style 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.

Skinning Tips

1) To change the font-size of PrimeFaces components globally, use the .ui-widget style class. An example with 12px font size.

```
.ui-widget,
.ui-widget .ui-widget {
     font-size: 12px !important;
}
```

2) To create a themeable application, keep the user's selected theme in a session scoped bean and use EL to change it dynamically when including the skin.css to your page.

```
<link type="text/css" rel="stylesheet"
     href="#{request.contextPath}/themes/#{user.theme}/skin.css" />
```

3) 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.

4) 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.

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);	Specifyes 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 accessesible 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 implicity, 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 javacript.

```
<h:form id="form1">
    <h:inputText id="name" />
</h:form>

//#{p:component('name')} returns 'form1:name'
```

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.

Function	Description
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 principial 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. 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)
- Seam

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.

Application	Technologies
MovieCollector	PrimeFaces-Spring-JPA
PhoneBook	PrimeFaces-Seam-JPA
BookStore	PrimeFaces-Optimus-Guice-JPA

All applications are built with maven and use in memory databases so it's as easy as running;

mvn clean jetty:run or mvn clean jetty:run-exploded

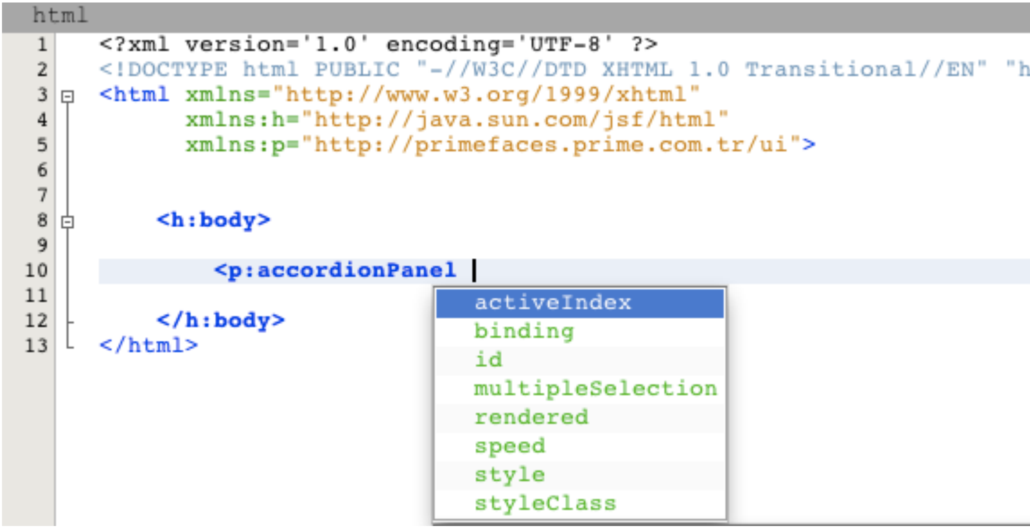
command to deploy in your local environment.

11. IDE Support

11.1 NetBeans

PrimeFaces tag completion is supported by NetBeans 6.8 out of the box. Presence of PrimFaces jar in classpath enables tag and attribute completion support.

```
1   <?xml version='1.0' encoding='UTF-8' ?>
2   <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "htt
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:|
11             <p:accordionPanel>    primefaces-p.tld
12      </     <p:ajax>              primefaces-p.tld
13  </html>    <p:ajaxStatus>        primefaces-p.tld
               <p:autoComplete>      primefaces-p.tld
               <p:barChart>          primefaces-p.tld
               <p:calendar>          primefaces-p.tld
               <p:captcha>           primefaces-p.tld
               <p:carousel>          primefaces-p.tld
               <p:chartSeries>       primefaces-p.tld
               <p:collector>         primefaces-p.tld
               <p:colorPicker>       primefaces-p.tld
               <p:column>            primefaces-p.tld
               <p:columnChart>       primefaces-p.tld
               <p:commandButton>     primefaces-p.tld
               <p:commandLink>       primefaces-p.tld
               <p:confirmDialog>     primefaces-p.tld
               <p:dataExporter>      primefaces-p.tld
```



11.2 Eclipse

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.

Preferences

type filter text

General
Appearance
Capabilities
Compare/Patch
Content Types
Editors
Keys
Network Connectior
Perspectives
Search
Security
Service Policies
Startup and Shutdo
Web Browser
Workspace
Ant
Data Management
Help
Install/Update
Java
Java EE
Plug-in Development
Remote Systems

Content Types

See 'File Associations' for associating editors with file types.

Content types:

Java Source File
JavaScript Source File
JSP
Refactoring History File
Refactoring History Index
Runtime log files
WikiText
XML

File associations:

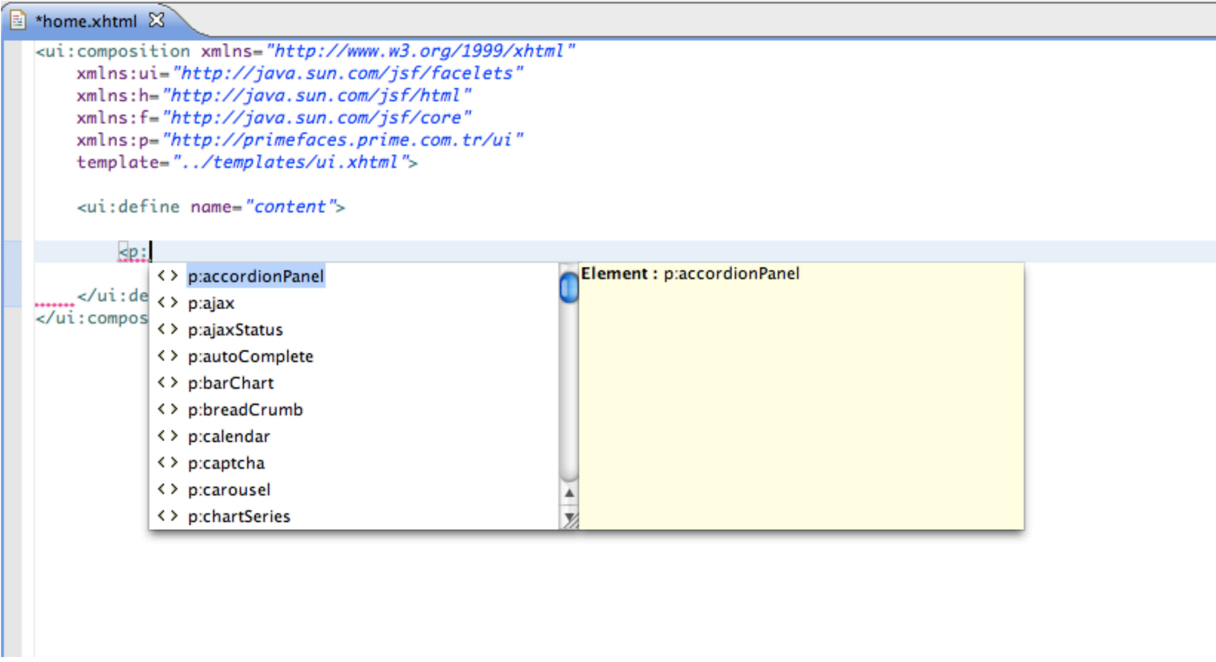
*.xhtml
*.jsp (locked)
*.jspx (locked)
*.jsv (locked)
*.jtpl (locked)

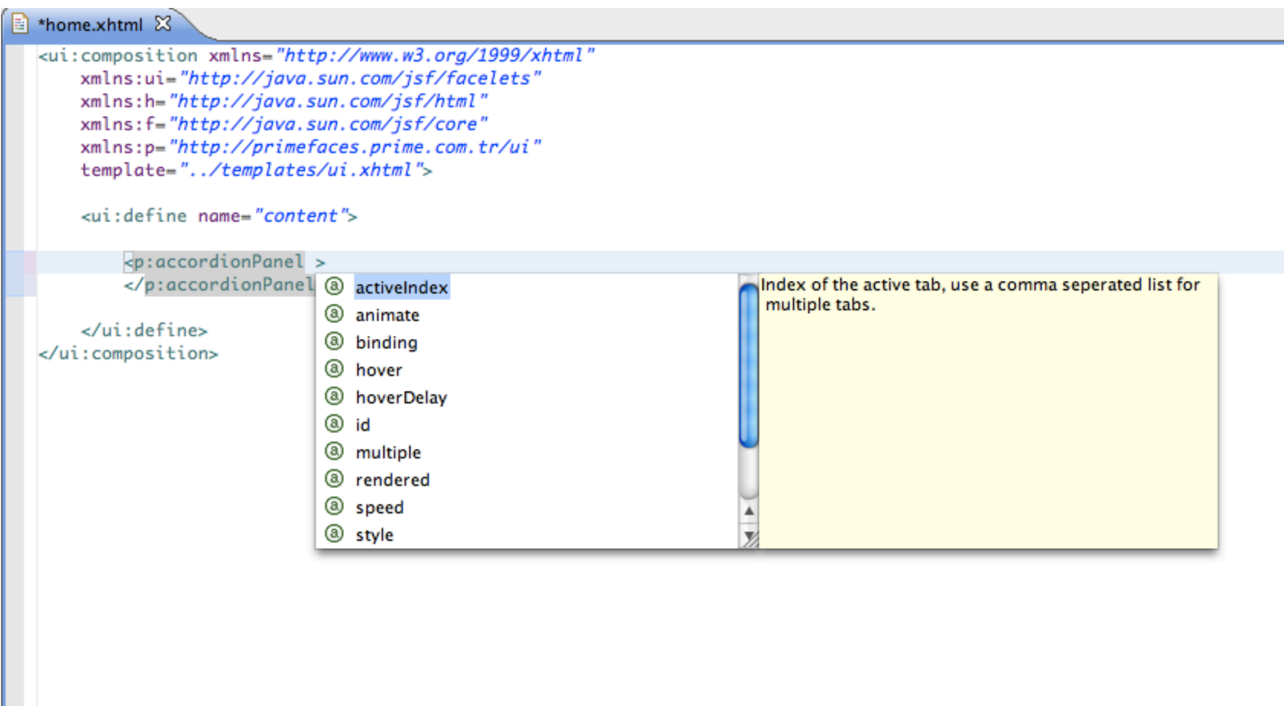
Add...
Edit...
Remove

Default encoding: ISO-8859-1 Update

Cancel OK

With this setting, PrimeFaces components can get tag/attribute completion when opened with jsp editor.




*home.xhtml
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
xmlns:ui="http://java.sun.com/jsf/facelets"
xmlns:h="http://java.sun.com/jsf/html"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:p="http://primefaces.prime.com.tr/ui"
template="../templates/ui.xhtml">
<ui:define name="content">
<p:accordionPanel >
</p:accordionPanel
</ui:define>
</ui:composition>
activeIndex
animate
binding
hover
hoverDelay
id
multiple
rendered
speed
style
Index of the active tab, use a comma seperated list for multiple tabs.

12. Portlets

PrimeFaces works well in a portlet environment, both portlet 1.0 and portlet 2.0 JSRs are tested and supported. PrimeFaces Ajax infrastructure is tuned for portal environments.

PrimeFaces portlets should work with any compliant portlet-bridge implementation, we highly encourage using MyFaces portlet bridge which is the reference implementation.

portlet.xml

Here's a sample portlet.xml from the prime-portlet application that is available in subversion repository of PrimeFaces.

```
 <?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>primeportlet</portlet-name>
       <display-name>Prime Portlet</display-name>
       <portlet-class>javax.portlet.faces.GenericFacesPortlet</portlet-class>

       <init-param>
            <name>javax.portlet.faces.defaultViewId.view</name>
            <value>/view.jsp</value>
      </init-param>
      <init-param>
            <name>javax.portlet.faces.defaultViewId.edit</name>
            <value>/edit.jsp</value>
      </init-param>
      <init-param>
            <name>javax.portlet.faces.preserveActionParams</name>
            <value>true</value>
      </init-param>

      <supports>
            <mime-type>text/html</mime-type>
            <portlet-mode>view</portlet-mode>
            <portlet-mode>edit</portlet-mode>
      </supports>

      <portlet-info>
            <title>Prime Portlet</title>
            <short-title>Prime</short-title>
      </portlet-info>
    </portlet>
</portlet-app>
```

Important note here is to preserve the action parameters to make PrimeFaces PPR work with portlets.

```
<init-param>
      <name>javax.portlet.faces.preserveActionParams</name>
      <value>true</value>
</init-param>
```

Other than this there's no specific change in the application configuration.

13. Project Resources

Documentation

Reference documentation is the major 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:8080/prime-showcase

Twitter and Facebook

You can follow PrimeFaces on twitter using **@primefaces** and join the Facebook group**.**

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.

PrimeFaces leader Cagatay Civici is a JavaServer Faces Expert Group Member, Apache MyFaces PMC member and committer of Atmosphere Ajax Push Framework.

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?
Compatibility is a major goal of PrimeFaces, we aim to be compatible with major component suites.

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. Where can I get an unreleased snapshot?
Nightly snapshot builds of a future release is deployed at http://repository.prime.com.tr.

8. What is the license PrimeFaces have?
PrimeFaces is free to use and licensed under Apache License V2.

9. Can I use PrimeFaces in a commercial software?
Yes, Apache V2 License is a commercial friendly library. JBoss Richfaces and Apache Trinidad(see this blog entry) are tested with and known to be compatible with PrimeFaces. IceFaces is known to be incompatible.

THE END