



# **JBoss Enterprise Web Platform 5 RichFaces Developer Guide**

---

for Use with JBoss Enterprise Web Platform  
Edition 5.1.1

Red Hat Documentation Group

# JBoss Enterprise Web Platform 5 RichFaces Developer Guide

---

for Use with JBoss Enterprise Web Platform  
Edition 5.1.1

Red Hat Documentation Group

## Legal Notice

Copyright © 2011 Red Hat, Inc.

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack® Word Mark and OpenStack Logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

## Abstract

This book is a reference guide for developers about using RichFaces with JBoss Enterprise Web Platform 5 and its patch releases.

# Table of Contents

<b>Preface</b> .....	<b>3</b>
1. Document Conventions	3
1.1. Typographic Conventions	3
1.2. Pull-quote Conventions	4
1.3. Notes and Warnings	5
2. Getting Help and Giving Feedback	5
2.1. Do You Need Help?	5
2.2. Give us Feedback	6
<b>Chapter 1. Introduction</b> .....	<b>7</b>
<b>Chapter 2. Getting Started with RichFaces</b> .....	<b>8</b>
2.1. Simple JSF application with RichFaces	8
2.1.1. Adding RichFaces libraries into the project	8
2.1.2. Registering RichFaces in web.xml	8
2.1.3. Managed bean	11
2.1.4. Registering the bean in faces-config.xml	11
2.1.5. RichFaces Greeter index.jsp	11
2.2. Relevant Resources Links	13
<b>Chapter 3. Settings for different environments</b> .....	<b>14</b>
3.1. Web Application Descriptor Parameters	14
3.2. Sun JSF RI	17
3.3. Facelets Support	17
3.4. JBoss Seam Support	18
<b>Chapter 4. Basic concepts of the RichFaces Framework</b> .....	<b>23</b>
4.1. Introduction	23
4.2. RichFaces Architecture Overview	23
4.3. Request Errors and Session Expiration Handling	26
4.3.1. Request Errors Handling	26
4.3.2. Session Expired Handling	27
4.4. Skinnability	28
4.4.1. Why Skinnability	28
4.4.2. Using Skinnability	28
4.4.3. Example	29
4.4.4. Skin Parameters Tables in RichFaces	29
4.4.5. Creating and Using Your Own Skin File	31
4.4.6. Built-in Skinnability in RichFaces	31
4.4.7. Changing skin in runtime	32
4.4.8. Standard Controls Skinning	34
4.4.8.1. Standard Level	36
4.4.8.2. Extended level	39
4.4.9. Client-side Script for Extended Skinning Support	42
4.4.10. XCSS File Format	43
4.4.11. Plug-n-Skin	44
4.4.11.1. Details of Usage	48
4.5. State Manager API	50
4.6. Identifying User Roles	54
<b>Revision History</b> .....	<b>55</b>



# Preface

## 1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the [Liberation Fonts](#) set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later include the Liberation Fonts set by default.

### 1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

#### Mono-spaced Bold

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keys and key combinations. For example:

To see the contents of the file **my\_next\_bestselling\_novel** in your current working directory, enter the **cat my\_next\_bestselling\_novel** command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a key, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from an individual key by the plus sign that connects each part of a key combination. For example:

Press **Enter** to execute the command.

Press **Ctrl+Alt+F2** to switch to a virtual terminal.

The first example highlights a particular key to press. The second example highlights a key combination: a set of three keys pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **mono-spaced bold**. For example:

File-related classes include **filesystem** for file systems, **file** for files, and **dir** for directories. Each class has its own associated set of permissions.

#### Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, select the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications** → **Accessories** →

**Character Map** from the main menu bar. Next, choose **Search** → **Find...** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

### ***Mono-spaced Bold Italic** or **Proportional Bold Italic***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type **ssh *username@domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh *john@example.com***.

The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount */home***.

To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

Publican is a *DocBook* publishing system.

## 1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

```
books      Desktop  documentation  drafts  mss    photos  stuff  svn
books_tests Desktop1  downloads      images  notes  scripts svgs
```

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```

package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
    public static void main(String args[])
        throws Exception
    {
        InitialContext iniCtx = new InitialContext();
        Object          ref    = iniCtx.lookup("EchoBean");
        EchoHome        home   = (EchoHome) ref;
        Echo            echo    = home.create();

        System.out.println("Created Echo");

        System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
    }
}

```

### 1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.



#### Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



#### Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.



#### Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

## 2. Getting Help and Giving Feedback

### 2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, visit the Red Hat Customer Portal at <http://access.redhat.com>. Through the customer portal, you can:

- search or browse through a knowledgebase of technical support articles about Red Hat products.
- submit a support case to Red Hat Global Support Services (GSS).
- access other product documentation.



Red Hat also hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available mailing lists at <https://www.redhat.com/mailman/listinfo>. Click on the name of any mailing list to subscribe to that list or to access the list archives.

## 2.2. Give us Feedback

If you find a typographical error, or know how this guide can be improved, we would love to hear from you. Submit a report in Bugzilla against the product **JBoss Enterprise Application Platform 5** and the component **doc-RichFaces Dev Guide**. The following link will take you to a pre-filled bug report for this product: <http://bugzilla.redhat.com/>.

Fill out the following template in Bugzilla's **Description** field. Be as specific as possible when describing the issue; this will help ensure that we can fix it quickly.

Document URL:

Section Number and Name:

Describe the issue:

Suggestions for improvement:

Additional information:

Be sure to give us your name so that you can receive full credit for reporting the issue.

## Chapter 1. Introduction

RichFaces is an open source framework that adds AJAX capability into existing JSF applications without resorting to JavaScript.

RichFaces leverages aspects of the JavaServer Faces (JSF) framework, including lifecycle, validation, conversion facilities, and management of static and dynamic resources. RichFaces components with built-in AJAX support and a highly customizable look-and-feel can be easily incorporated into JSF applications.

RichFaces allows you to:

- ▶ Experience the benefits of JSF while working with AJAX. RichFaces is fully integrated into the JSF lifecycle. Where other frameworks only allow access to the managed bean facility, RichFaces lets you access the action and value change listeners, and invokes server-side validators and converters during the AJAX request-response cycle.
- ▶ Add AJAX capabilities to existing JSF applications. The RichFaces framework provides two component libraries (Core AJAX and UI). The Core library adds AJAX functionality to existing pages, so you need not write any JavaScript or replace existing components with new AJAX components manually. RichFaces enables page-wide rather than component-wide AJAX support, giving you the opportunity to define events on the page.
- ▶ Quickly and easily create different views with a variety of components, available out-of-the-box. The RichFaces UI library contains components for adding rich user interface (UI) features to JSF applications, providing you with a broad variety of AJAX-enabled components with extensive skins support. RichFaces components are designed to integrate seamlessly with other third-party component libraries, so you have more options when you develop applications.
- ▶ Write your own rich components with built-in AJAX support. The Component Development Kit (CDK) is constantly being expanded. It includes both code generation and templating facilities and a simple JSP-like (JavaServer Pages) syntax, letting you create first-class rich components with built-in AJAX functionality.
- ▶ Package resources with application Java classes. RichFaces provides advanced support for managing different resource types, including images, JavaScript code, and CSS stylesheets. The resource framework makes it easier to include these resources in JAR files with your custom component code.
- ▶ Quickly and easily generate binary resources. The resource framework can generate images, sounds, Excel spreadsheets, etc. in real time, so you can, for example, create images with the Java Graphics 2D library and other similar resources.
- ▶ Customize the look and feel of your user interface with skins-based technology. RichFaces lets you easily define and manage different color schemes and other user interface parameters by using named *skin parameters*. This means you can access UI parameters from JSP and Java code to adjust your UI in real time. RichFaces includes a number of predefined skins to kick-start your application's development, but it is easy to create your own custom skins.
- ▶ Simultaneously create and test your components, actions, listeners, and pages. RichFaces will soon include an automated testing facility to generate test cases for your component as you develop it. The testing framework tests not only the components, but also any other server-side or client-side functionality, including JavaScript code — and it will do so *without* deploying the test application into the Servlet container.

RichFaces UI components can be implemented immediately, right out of the box. This saves development time and gives you immediate access to RichFaces web application development features, so experience with RichFaces is fast and easy to obtain.

## Chapter 2. Getting Started with RichFaces

This chapter tells you how to plug RichFaces components into a JSF application. The instructions are based on a simple *JSF with RichFaces* creation process, from downloading the required libraries to running the application in a browser. These instructions do not depend on the integrated development environment that is in use.

### 2.1. Simple JSF application with RichFaces

**RichFaces Greeter** — the simple application — is similar to a typical *hello world* application, with one exception: the world of RichFaces will say "Hello!" to the user first.

Create a standard JSF 1.2 project named **Greeter**. Include all required libraries, and continue with the instructions that follow.

#### 2.1.1. Adding RichFaces libraries into the project

From the **RichFaces** folder where you unzipped the RichFaces binary files, open the **lib**. This folder contains three **\*.jar** files with API, UI, and implementation libraries. Copy these JARs from **lib** to the **WEB-INF/lib** directory of your **Greeter** JSF application.



#### Important

A JSF application with RichFaces assumes that the following JARs are available in the project:

- » **commons-beanutils-1.7.0.jar**
- » **commons-collections-3.2.jar**
- » **commons-digester-1.8.jar**
- » **commons-logging-1.0.4.jar**
- » **jhighlight-1.0.jar**

#### 2.1.2. Registering RichFaces in web.xml

After you add the RichFaces libraries to the project, you must register them in the project **web.xml** file. Add the following to **web.xml**:

```
...
<!-- Plugging the "Blue Sky" skin into the project -->
<context-param>
  <param-name>org.richfaces.SKIN</param-name>
  <param-value>blueSky</param-value>
</context-param>

<!-- Making the RichFaces skin spread to standard HTML controls -->
<context-param>
  <param-name>org.richfaces.CONTROL_SKINNING</param-name>
  <param-value>enable</param-value>
</context-param>

<!-- Defining and mapping the RichFaces filter -->
<filter>
  <display-name>RichFaces Filter</display-name>
  <filter-name>richfaces</filter-name>
  <filter-class>org.ajax4jsf.Filter</filter-class>
</filter>

<filter-mapping>
  <filter-name>richfaces</filter-name>
  <servlet-name>Faces Servlet</servlet-name>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
...
```

For more information about RichFaces skins, read [Section 4.4, “Skinnability”](#).

Finally, your **web.xml** should look like this:

```
<?xml version="1.0"?>
<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
<display-name>Greeter</display-name>

<context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>server</param-value>
</context-param>

<context-param>
    <param-name>org.richfaces.SKIN</param-name>
    <param-value>blueSky</param-value>
</context-param>

<context-param>
    <param-name>org.richfaces.CONTROL_SKINNING</param-name>
    <param-value>enable</param-value>
</context-param>

<filter>
    <display-name>RichFaces Filter</display-name>
    <filter-name>richfaces</filter-name>
    <filter-class>org.ajax4jsf.Filter</filter-class>
</filter>

<filter-mapping>
    <filter-name>richfaces</filter-name>
    <servlet-name>Faces Servlet</servlet-name>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
</filter-mapping>

<listener>
    <listener-class>com.sun.faces.config.ConfigureListener</listener-class>
</listener>

<!-- Faces Servlet -->
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<!-- Faces Servlet Mapping -->
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
</servlet-mapping>

<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
</web-app>
```

### 2.1.3. Managed bean

The **RichFaces Greeter** application needs a managed bean. In the project's **JavaSource** directory, create a new managed bean named **user** in the **demo** package. Place the following code in **user**:

```
package demo;

public class user {
    private String name="";
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

### 2.1.4. Registering the bean in faces-config.xml

To register the **user** bean, add the following to the **faces-config.xml** file:

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config version="1.2"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xi="http://www.w3.org/2001/XInclude"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd">
    <managed-bean>
        <description>UsernName Bean</description>
        <managed-bean-name>user</managed-bean-name>
        <managed-bean-class>demo.user</managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
        <managed-property>
            <property-name>name</property-name>
            <property-class>java.lang.String</property-class>
            <value/>
        </managed-property>
    </managed-bean>
</faces-config>
```

### 2.1.5. RichFaces Greeter index.jsp

**RichFaces Greeter** has only one JSP page. Create **index.jsp** in the root of **WEB CONTENT** folder and add the following to the JSP file:

```

<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<!-- RichFaces tag library declaration -->
<%@ taglib uri="http://richfaces.org/a4j" prefix="a4j"%>
<%@ taglib uri="http://richfaces.org/rich" prefix="rich"%>

<html>
  <head>
    <title>RichFaces Greeter</title>
  </head>
  <body>
    <f:view>
      <a4j:form>
        <rich:panel header="RichFaces Greeter" style="width:
315px">
          <h:outputText value="Your name: " />
          <h:inputText value="#{user.name}" >
            <f:validateLength minimum="1" maximum="30"
/>
          </h:inputText>
          <a4j:commandButton value="Get greeting"
reRender="greeting" />
          <h:panelGroup id="greeting" >
            <h:outputText value="Hello, " rendered="#{not
empty user.name}" />
            <h:outputText value="#{user.name}" />
            <h:outputText value="!" rendered="#{not empty
user.name}" />
          </h:panelGroup>
        </rich:panel>
      </a4j:form>
    </f:view>
  </body>
</html>

```

The application uses three RichFaces components: **<rich:panel>** is used as visual container for information; **<a4j:commandButton>** with built-in AJAX support lets a greeting be rendered dynamically after a response returns; and **<a4j:form>** helps the button to perform the action.



## Note

The RichFaces tag library should be declared on each JSP page. For XHTML pages, add the following lines to declare your tag libraries:

```
<xmlns:a4j="http://richfaces.org/a4j">  
<xmlns:rich="http://richfaces.org/rich">
```

Now, run the application on the server by pointing your browser to the **index.jsp** page:  
**http://localhost:8080/Greeter/index.jsf**

RichFaces Greeter

Your name:

Hello, Alex!

**Figure 2.1. "RichFaces Greeter" application**

## 2.2. Relevant Resources Links

[JBoss Developer Studio](#) includes tight integration with the RichFaces component framework.



## Chapter 3. Settings for different environments

RichFaces includes support for all tags (components) included in the JavaServer Faces (JSF) specification. To add RichFaces capabilities to an existing JSF project, place the RichFaces libraries into the **lib** directory of the project, and add filter mapping. The behavior of the existing project does not change when you add RichFaces.

### 3.1. Web Application Descriptor Parameters

RichFaces does not require that any parameters be defined in your **web.xml**, but the RichFaces parameters listed below will help you during the development process and increase the flexibility of your RichFaces applications.

**Table 3.1. Initialization Parameters**

Name	Default	Description
org.richfaces.SKIN	DEFAULT	The name of a skin that is used in an application. Can be a literal string with a skin name or the EL expression ( <code>#{...}</code> ) associated with a String property (skin name) of a property of a <b>org.richfaces.framework.skin</b> type. In the latter case, that instance is used as the current skin.
org.richfaces.LoadScriptStrategy	DEFAULT	Defines how the RichFaces script files are loaded to the application. Possible values are <b>ALL</b> , <b>DEFAULT</b> and <b>NONE</b> .
org.richfaces.LoadStyleStrategy	DEFAULT	Defines how the RichFaces style files are loaded into the application. Possible values are: <b>ALL</b> , <b>DEFAULT</b> , or <b>NONE</b> .
org.ajax4jsf.LOGFILE	none	The URL of an application or a container log file (if applicable). If this parameter is set, content from the given URL is shown on a Debug page in the <b>iframe</b> window.
org.ajax4jsf.VIEW_HANDLERS	none	A comma-separated list of <b>ViewHandler</b> instances for inserting in a view handler chain. These handlers are inserted before the RichFaces viewhandlers, in the order they are listed. In a Facelets application, you would declare <b>com.sun.facelets.FaceletViewHandler</b> here instead of in the <b>faces-config.xml</b> file.
org.ajax4jsf.CONTROL_COMPONENTS	none	A comma-separated list of special <i>control case</i> components, such as the messages bundle loader or an alias bean component. These handlers are provided via a reflection from the static field <b>COMPONENT_TYPE</b> . Encoding methods for these components are always called while rendering AJAX responses, even if a component has not

		been updated.
org.ajax4jsf.ENCRIPT_RESOURCE_DATA	false	For generated resources (such as encrypt generation data), this is encoded in the resource URL. For example, the URL of an image generated by the <b>mediaOutput</b> component contains the name of a generation method. Since malicious code can exploit this to create a request for any JSF bean or attribute, this parameter should be set to <b>true</b> in critical applications. (This fix works with Java Runtime Environment 1.4.)
org.ajax4jsf.ENCRIPT_PASSWORD	random	A password used to encrypt resource data. If this is not set, a random password is used.
org.ajax4jsf.COMPRESS_SCRIPT	true	When defined, does not allow the framework to reformat JavaScript files. This means that the debug feature cannot be used.
org.ajax4jsf.RESOURCE_URI_PREFIX	a4j	Defines the prefix to be added to the URLs of all generated resources. This is designed to handle RichFaces generated resource requests.
org.ajax4jsf.GLOBAL_RESOURCE_URI_PREFIX	a4j/g	Defines the prefix to be added to the URI of all global resources. This prefix is designed to handle RichFaces generated resource requests.
org.ajax4jsf.SESSION_RESOURCE_URI_PREFIX	a4j/s	Defines the prefix to be used to track the sessions of generated resources. This prefix is designed to handle RichFaces generated resource requests.
org.ajax4jsf.DEFAULT_EXPIRE	86400	Defines the period (in seconds) for which resources are cached when they are streamed back to the browser.
org.ajax4jsf.SERIALIZE_SERVER_STATE	false	If set to <b>true</b> , the component state (not the tree) will be serialized before it is stored in the session. This can be useful in applications with view state that is sensitive to model changes. Alternatively, use <b>com.sun.faces.serialize ServerState</b> and

**org.apache.myfaces.SERIALIZE\_STATE\_IN\_SESSION**  
parameters in their respective environments.



### Note

**org.richfaces.SKIN** is used in the same way as **org.ajax4jsf.SKIN**.

**Table 3.2. org.ajax4jsf.Filter Initialization Parameters**

Name	Default	Description
log4j-init-file	-	A path (relative to the web application's context) to the <b>log4j.xml</b> configuration file. This can be used to set up per-application custom logging.
enable-cache	true	Enables caching of framework-generated resources (JavaScript, CSS, images, etc.). However, your cached resources will not be used when attempting to debug custom JavaScript or Styles.
forcenotrf	true	Forces all JSF pages to be parsed by a HTML syntax check filter. If set to <b>false</b> , only AJAX responses will be parsed and converted to well-formed XML. Setting this to <b>false</b> can improve performance, but may also cause unexpected information to be rendered during AJAX updates.

## 3.2. Sun JSF RI

RichFaces works with JavaServer Faces 1.2\_13 without needing to modify additional settings.

## 3.3. Facelets Support

RichFaces has high-level support for Facelets, regardless of the version used. However, some JSF frameworks (including Faces) require that their own **ViewHandler** be listed first in the **ViewHandler** chain. RichFaces also requires that its **AjaxViewHandler** be listed first, but because it is installed first, no settings will need to be altered. Where multiple frameworks are used without RichFaces, you can use the **VIEW\_HANDLERS** parameter to define the order in which the **ViewHandler**s are used. For example:

```
...  
<context-param>  
    <param-name>org.ajax4jsf.VIEW_HANDLERS</param-name>  
    <param-value>com.sun.facelets.FaceletViewHandler</param-value>  
</context-param>  
...
```

This declares that while **Facelets** will officially be first, **AjaxViewHandler** will briefly be ahead of it to perform some small, important task.



#### Note

In this case, you need not define **FaceletViewHandler** in **WEB-INF/faces-config.xml**.

## 3.4. JBoss Seam Support

RichFaces is compatible with **JBoss Seam** and Facelets when run within JBoss Enterprise Application Server. No additional JARs are required. All you need to do is package the RichFaces library with your application.

For **Seam 1.2**, your **web.xml** must be as follows:

```

<?xml version="1.0" ?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
          version="2.4">

  <!-- richfaces -->

  <filter>
    <display-name>RichFaces Filter</display-name>
    <filter-name>richfaces</filter-name>
    <filter-class>org.ajax4jsf.Filter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>richfaces</filter-name>
    <url-pattern>*.seam</url-pattern>
  </filter-mapping>

  <!-- Seam -->

  <listener>
    <listener-class>org.jboss.seam.servlet.SeamListener</listener-class>
  </listener>

  <servlet>
    <servlet-name>Seam Resource Servlet</servlet-name>
    <servlet-class>org.jboss.seam.servlet.ResourceServlet</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>Seam Resource Servlet</servlet-name>
    <url-pattern>/seam/resource/*</url-pattern>
  </servlet-mapping>

  <filter>
    <filter-name>Seam Filter</filter-name>
    <filter-class>org.jboss.seam.web.SeamFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>Seam Filter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <!-- MyFaces -->

  <listener>
    <listener-
class>org.apache.myfaces.webapp.StartupServletContextListener</listener-class>
  </listener>

  <!-- JSF -->

  <context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
  </context-param>

```

```
<context-param>
  <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
  <param-value>.xhtml</param-value>
</context-param>

<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.seam</url-pattern>
</servlet-mapping>
</web-app>
```

**Seam 2.x** supports RichFaces Filter, so your **web.xml** must look like this:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

    <context-param>
        <param-name>org.ajax4jsf.VIEW_HANDLERS</param-name>
        <param-value>com.sun.facelets.FaceletViewHandler</param-value>
    </context-param>

    <!-- Seam -->

    <listener>
        <listener-class>org.jboss.seam.servlet.SeamListener</listener-class>
    </listener>

    <servlet>
        <servlet-name>Seam Resource Servlet</servlet-name>
        <servlet-class>org.jboss.seam.servlet.SeamResourceServlet</servlet-
class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Seam Resource Servlet</servlet-name>
        <url-pattern>/seam/resource/*</url-pattern>
    </servlet-mapping>

    <filter>
        <filter-name>Seam Filter</filter-name>
        <filter-class>org.jboss.seam.servlet.SeamFilter</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>Seam Filter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!-- JSF -->

    <context-param>
        <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
        <param-value>.xhtml</param-value>
    </context-param>

    <context-param>
        <param-name>facelets.DEVELOPMENT</param-name>
        <param-value>true</param-value>
    </context-param>

    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.seam</url-pattern>

```



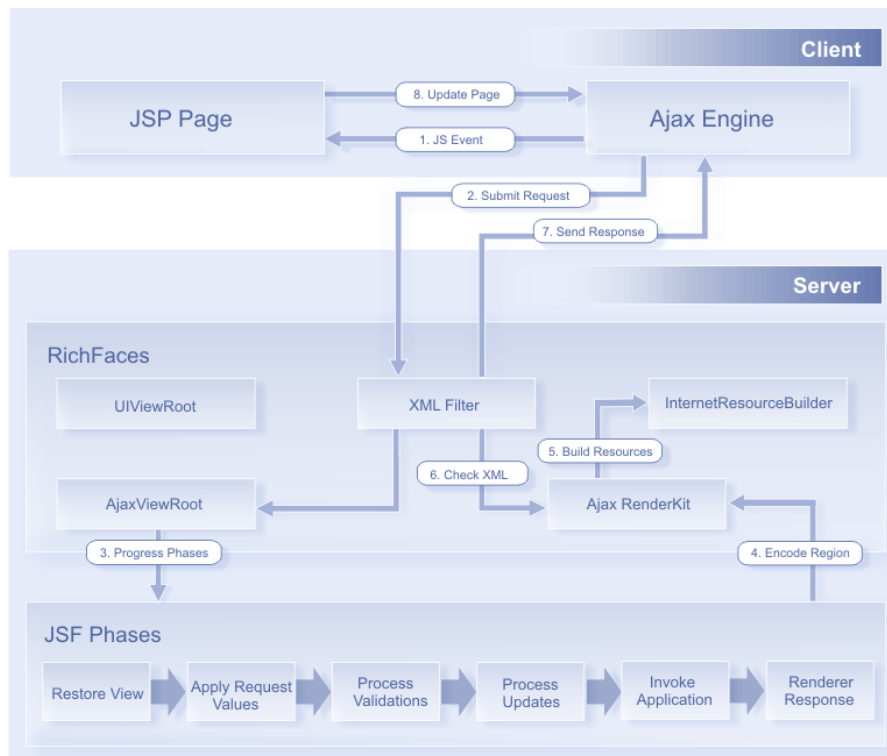
```
    </servlet-mapping>  
</web-app>
```

## Chapter 4. Basic concepts of the RichFaces Framework

### 4.1. Introduction

The RichFaces Framework is implemented as a component library that adds AJAX capabilities into existing pages. This means that you do not need to write any JavaScript code or replace existing components with new AJAX widgets. RichFaces enables page-wide AJAX support instead of the traditional component-wide support, so you can define areas of the page that will reflect changes made by AJAX events on the client.

The diagram following shows the process in full:

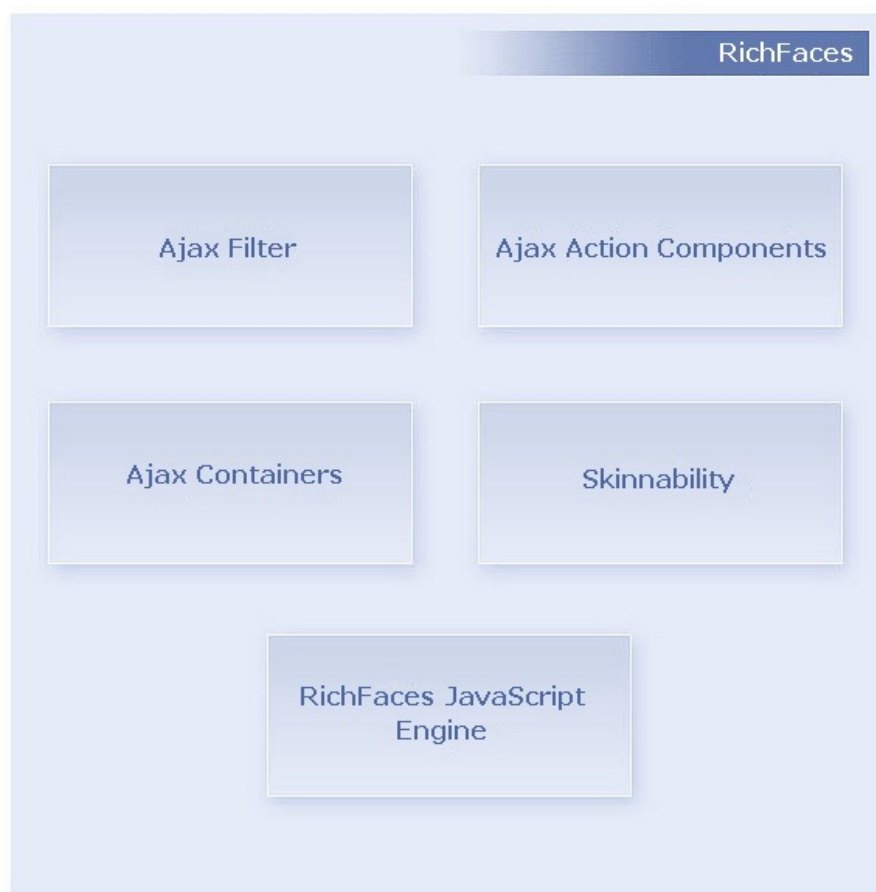


**Figure 4.1. Request Processing flow**

RichFaces lets you use JSF tags to define sections of a JSF page that you wish to update with the results of an AJAX request. It also provides you with several options for sending AJAX requests to the server. You do not need to write any JavaScript or **XMLHttpRequest** objects by hand — everything is done automatically.

### 4.2. RichFaces Architecture Overview

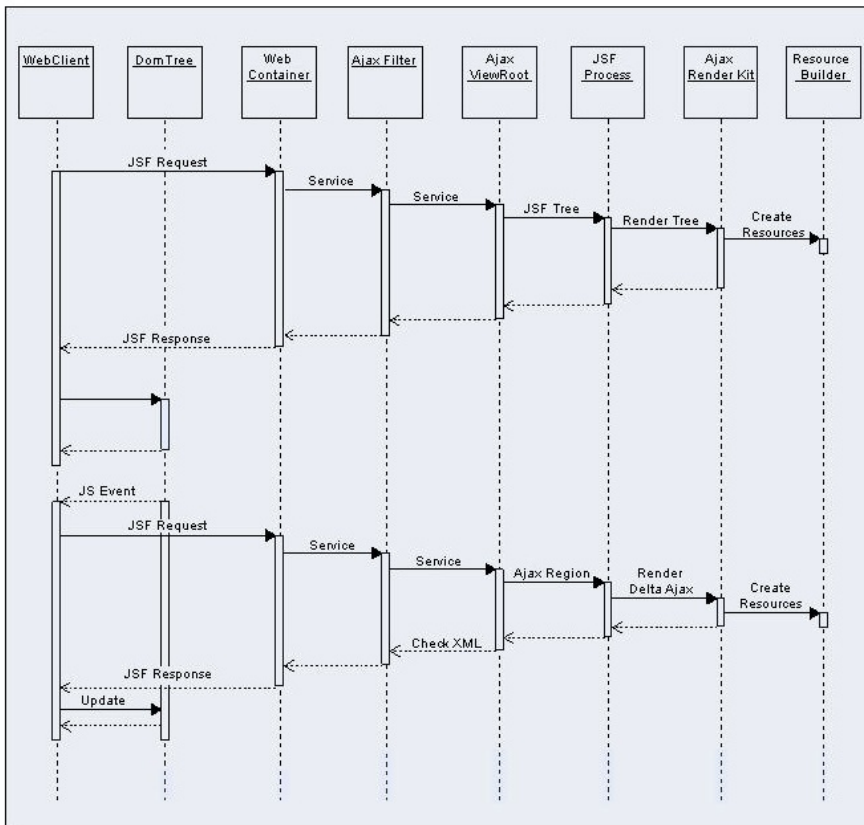
The following figure lists several important elements of the RichFaces Framework.



**Figure 4.2. Core AJAX component structure**

### AJAX Filter.

To make the most of RichFaces, you should register a **Filter** in your application's `web.xml`. The **Filter** recognizes multiple request types. The sequence diagram in Figure 5.3 shows the differences in processing between a *regular* JSF request and an AJAX request.



**Figure 4.3. Request processing sequence diagram**

In either case, the required static or dynamic resource information that your application requests is registered in the **ResourceBuilder** class.

When a resource request is issued, the RichFaces filter checks the **Resource Cache** for this resource. If it is present, the resource is returned to the client. Otherwise, the filter searches for the resource among those registered in the **ResourceBuilder**. If the resource is registered, the RichFaces filter requests that the **ResourceBuilder** creates (delivers) the resource.

The diagram that follows illustrates the process of requesting a resource.

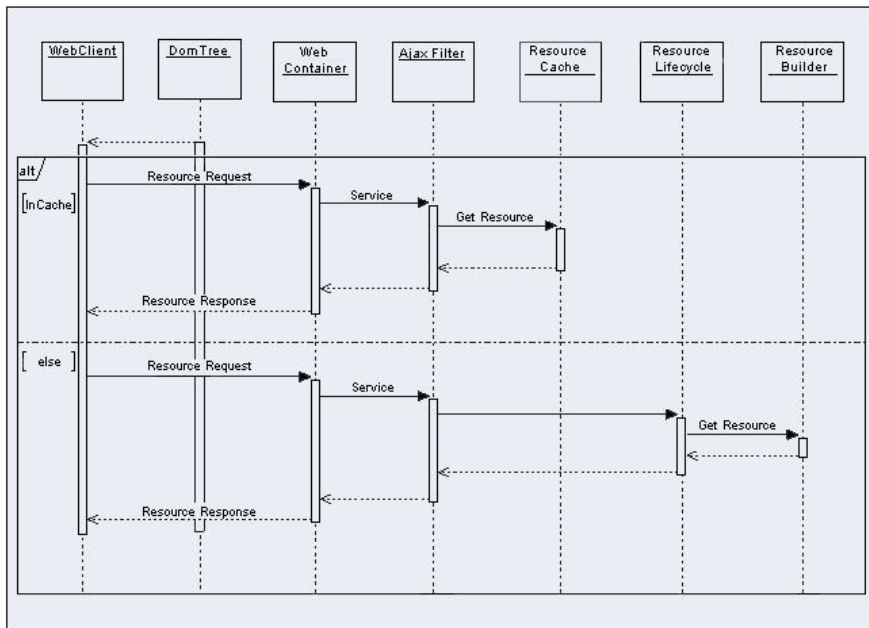


Figure 4.4. Resource request sequence diagram

## AJAX Action Components

AJAX Action components are used to send AJAX requests from the client side. There are a number of AJAX Action components, including `<a4j:commandButton>`, `<a4j:commandLink>`, `<a4j:poll>`, and `<a4j:support>`.

## AJAX Containers

**AjaxContainer** is an interface that defines an area on your JSF page that should be decoded during an AJAX request. **AjaxViewRoot** and **AjaxRegion** are both implementations of this interface.

## JavaScript Engine

The RichFaces JavaScript Engine runs on the client side, and updates different areas of your JSF page based on information from the AJAX response. This JavaScript code operates automatically, so there is no need to use it directly.

## 4.3. Request Errors and Session Expiration Handling

RichFaces lets you redefine the standard handlers responsible for processing exceptions. We recommend defining your own JavaScript, which will be executed when exceptional situations occur.

Add the following code to **web.xml**:

```
<context-param>
  <param-name>org.ajax4jsf.handleViewExpiredOnClient</param-name>
  <param-value>true</param-value>
</context-param>
```

### 4.3.1. Request Errors Handling

To execute your own code on the client in the event of an error during an AJAX request, you must

redefine the standard **A4J.AJAX.onError** method like so:

```
A4J.AJAX.onError = function(req, status, message){  
    window.alert("Custom onError handler "+message);  
}
```

This function accepts the following as parameters:

**req**

a parameter string of a request that calls an error

**status**

the number of an error returned by the server

**message**

a default message for the given error

Therefore, you can create your own handler that is called when timeouts, internal server errors, etc. occur.

### 4.3.2. Session Expired Handling

You can also redefine the **onExpired** framework method that is called on the **SessionExpiration** event.

*Example:*

```
A4J.AJAX.onExpired = function(loc, expiredMsg){  
    if(window.confirm("Custom onExpired handler "+expiredMsg+" for a location:  
"+loc)){  
        return loc;  
    } else {  
        return false;  
    }  
}
```

This function can take the following parameters:

**loc**

the URL of the current page (can be updated on demand)

**expiredMsg**

a default message for display in the event of **SessionExpiration**.



## Note

Customized **onExpire** handlers do not work under MyFaces. MyFaces handles exceptions by internally generating a debug page. To prevent this behavior, use the following:

```
...
<context-param>
  <param-name>org.apache.myfaces.ERROR_HANDLING</param-name>
  <param-value>false</param-value>
</context-param>
...
```

## 4.4. Skinnability

### 4.4.1. Why Skinnability

If you look at any CSS (Cascading Style Sheets) file in an enterprise application, you will notice how often the same color is noted. Standard CSS cannot define a particular colour abstractly as a panel header color, the background color of an active pop-up menu item, a separator color, etc. To define common interface styles, you must copy the same value multiple times, and the more interfaces you have, the more repetition is required.

Therefore, if you want to change the palette of an application, you must change all interrelating values, or your interface can appear clumsy. If a customer wants to be able to adjust their interface's look and feel in real time, you must be able to alter several CSS files, each of which will contain the same value multiple times.

You can solve these problems with the *skins* that are built into and fully implemented in RichFaces. Every named skin has *skin parameters* that define a palette and other attributes of the user interface. By changing a few skin parameters, you can alter the appearance of dozens of components simultaneously, without interfering with interface consistency.

The **skinnability** feature cannot completely replace standard CSS, and does not eliminate its usage. Instead, it is a high-level extension of standard CSS that can be used in combination with regular CSS declarations. You can also refer to skin parameters in CSS through the JSF Expression Language. This lets you completely synchronize the appearance of all elements in your pages.

### 4.4.2. Using Skinnability

RichFaces *skinnability* is designed for use alongside:

- skin parameters defined in the RichFaces framework,
- predefined CSS classes for components, and
- user style classes.

A component's color scheme can be applied to its elements using any of three style classes:

- **A default style class inserted into the framework**

This contains style parameters that are linked to some constants from a skin. It is defined for every component and specifies a default level of representation. You can modify an application interface by changing the values of the skin parameters.

### ► A style class of skin extension

This class name is defined for every component element, and inserted into the framework to let you define a class with the same name in your CSS files. This lets you easily extend the appearance of all components that use this class.

### ► User style class

You can use one of the **styleClass** parameters to define your own class for component elements. As a result, the appearance of one particular component is changed according to a CSS style parameter specified in the class.

## 4.4.3. Example

The following is an example of a simple *panel* component:

```
<rich:panel> ... </rich:panel>
```

This code generates a panel component on a page, which consists of two elements: a wrapper **<div>** element and a **<div>** element for the panel body with the specified style properties. The wrapper **<div>** element will look like this:

```
<div class="dr-pnl rich-panel">
  ...
</div>
```

**dr-pnl** is a CSS class that is specified in the framework via skin parameters:

- **background-color** is defined with **generalBackgroundColor**
- **border-color** is defined with **panelBorderColor**

You can change all colors for all panels on all pages by changing these skin parameter values. However, if you specify a **<rich:panel>** class on the page, its parameters are also acquired by all panels on this page.

Developers can also change the style properties for panel. For example:

```
<rich:panel styleClass="customClass" />
```

The previous definition could add some style properties from **customClass** to one particular panel. As a result, we will get three styles:

```
<div class="dr_pnl rich-panel customClass">
  ...
</div>
```

## 4.4.4. Skin Parameters Tables in RichFaces

RichFaces provides eight predefined skin parameters (skins) at the simplest level of common customization:

- DEFAULT
- plain
- emeraldTown



- blueSky
- wine
- japanCherry
- ruby
- classic
- deepMarine

To apply a skin, you must specify a skin name in the **org.richfaces.SKIN** context parameter.

The following table shows the values for each parameter in the **blueSky** skin:

**Table 4.1. Colors**

Parameter name	Default value
headerBackgroundColor	#BED6F8
headerGradientColor	#F2F7FF
headTextColor	#000000
headerWeightFont	bold
generalBackgroundColor	#FFFFFF
generalTextColor	#000000
generalSizeFont	11px
generalFamilyFont	Arial, Verdana, sans-serif
controlTextColor	#000000
controlBackgroundColor	#FFFFFF
additionalBackgroundColor	#ECF4FE
shadowBackgroundColor	#000000
shadowOpacity	1
panelBorderColor	#BED6F8
subBorderColor	#FFFFFF
tabBackgroundColor	#C6DEFF
tabDisabledTextColor	#8DB7F3
trimColor	#D6E6FB
tipBackgroundColor	#FAE6B0
tipBorderColor	#E5973E
selectControlColor	#E79A00
generalLinkColor	#0078D0
hoverLinkColor	#0090FF
visitedLinkColor	#0090FF

**Table 4.2. Fonts**

Parameter name	Default value
headerSizeFont	11px
headerFamilyFont	Arial, Verdana, sans-serif
tabSizeFont	11px
tabFamilyFont	Arial, Verdana, sans-serif
buttonSizeFont	11px
buttonFamilyFont	Arial, Verdana, sans-serif
tableBackgroundColor	#FFFFFF
tableFooterBackgroundColor	#cccccc
tableSubfooterBackgroundColor	#f1f1f1
tableBorderColor	#C0C0C0

The **plain** skin was added in version 3.0.2. It has no parameters, and is important when embedding RichFaces components into existing projects with their own styles.

#### 4.4.5. Creating and Using Your Own Skin File

To create your own skin file:

- Create a file. In it, define skin constants to be used by style classes (see [Section 4.4.4, “Skin Parameters Tables in RichFaces”](#)). The name of the skin file should follow this format: **<name>.skin.properties**. (For examples of this file, see the RichFaces predefined skin parameters: **blueSky**, **classic**, **deepMarine**, etc. These files are located in the **richfaces-impl-xxxxx.jar** archive in the **/META-INF/skins** folder.
- Add the skin definition **<context-param>** to the **web.xml** of your application, like so:

```
...
<context-param>
    <param-name>org.richfaces.SKIN</param-name>
    <param-value>name</param-value>
</context-param>
...
```

- Place your **<name>.skin.properties** file in either your **/META-INF/skins** or **/WEB-INF/classes** directory.

#### 4.4.6. Built-in Skinnability in RichFaces

RichFaces lets you incorporate skins into your user interface (UI) design. This framework lets you use named skin parameters in your properties files to control skin appearance consistently across a set of components. You can see examples of predefined skins at: <http://livedemo.exadel.com/richfaces-demo/>

Skins let you define a style in which to render standard JSF components and custom JSF components built with RichFaces. You can experiment with skins by following these steps:

- Create a custom render kit and register it in the **faces-config.xml** like so:

```
<render-kit>
  <render-kit-id>NEW_SKIN</render-kit-id>
  <render-kit-
class>org.ajax4jsf.framework.renderer.ChameleonRenderKitImpl</render-kit-class>
</render-kit>
```

- Next, create and register custom renderers for the component based on the look-and-feel predefined variables:

```
<renderer>
  <component-family>javax.faces.Command</component-family>
  <renderer-type>javax.faces.Link</renderer-type>
  <renderer-class>newskin.HtmlCommandLinkRenderer</renderer-class>
</renderer>
```

- Finally, place a properties file with skin parameters into the class path root. There are two requirements for the properties file:
  - The file must be named **skinName.skin.properties**. In this case, we would call it **newskin.skin.properties**.
  - The first line in this file should be **render.kit=render-kit-id**. In this case, we would use **render.kit=NEW\_SKIN**.

More information about creating custom renderers can be found at:  
<http://java.sun.com/javaee/javaserverfaces/reference/docs/index.html>.

#### 4.4.7. Changing skin in runtime

You can change skins during runtime by defining the following EL-expression in your **web.xml**.

```
<context-param>
  <param-name>org.richfaces.SKIN</param-name>
  <param-value>#{skinBean.skin}</param-value>
</context-param>
```

The **skinBean** code looks like this:

```
public class SkinBean {
    private String skin;

    public String getSkin() {
        return skin;
    }
    public void setSkin(String skin) {
        this.skin = skin;
    }
}
```

You must also set the **skin** property's initial value in the configuration file. To set **classic**:

```

<managed-bean>
  <managed-bean-name>skinBean</managed-bean-name>
  <managed-bean-class>SkinBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>skin</property-name>
    <value>classic</value>
  </managed-property>
</managed-bean>

```

You can also change the properties of the default skin. To do so, edit the properties of the default skin. The following shows you example page code:

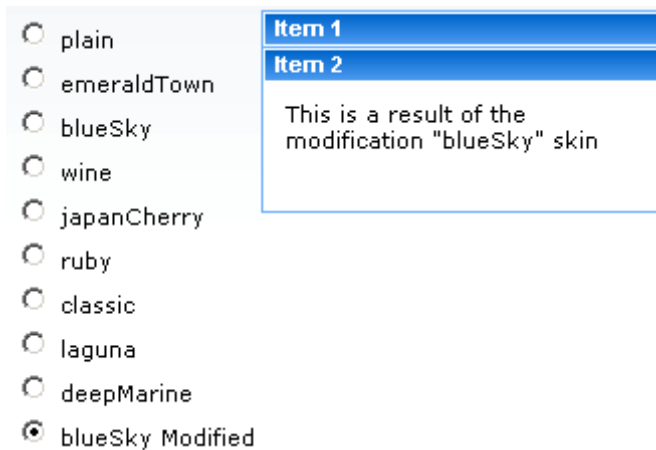
```

<h:form>
  <div style="display: block; float: left">
    <h:selectOneRadio value="#{skinBean.skin}" border="0"
  layout="pageDirection" title="Changing skin" style="font-size: 8; font-family:
  comic" onchange="submit()">
      <f:selectItem itemLabel="plain" itemValue="plain" />
      <f:selectItem itemLabel="emeraldTown" itemValue="emeraldTown" />
      <f:selectItem itemLabel="blueSky" itemValue="blueSky" />
      <f:selectItem itemLabel="wine" itemValue="wine" />
      <f:selectItem itemLabel="japanCherry" itemValue="japanCherry" />
      <f:selectItem itemLabel="ruby" itemValue="ruby" />
      <f:selectItem itemLabel="classic" itemValue="classic" />
      <f:selectItem itemLabel="laguna" itemValue="laguna" />
      <f:selectItem itemLabel="deepMarine" itemValue="deepMarine" />
      <f:selectItem itemLabel="blueSky Modified" itemValue="blueSkyModify" />
    </h:selectOneRadio>
  </div>
  <div style="display: block; float: left">
    <rich:panelBar height="100" width="200">
      <rich:panelBarItem label="Item 1" style="font-family: monospace;
font-size: 12;">
        Changing skin in runtime
      </rich:panelBarItem>

      <rich:panelBarItem label="Item 2" style="font-family: monospace; font-size: 12;">
        This is a result of the modification "blueSky" skin
      </rich:panelBarItem>
    </rich:panelBar>
  </div>
</h:form>

```

The above code will generate the following list of options:



**Figure 4.5. Changing skin in runtime**

#### 4.4.8. Standard Controls Skinning

This feature is designed to unify the look and feel of standard HTML elements and RichFaces components. Skinning can be applied to all controls on a page based on element names and attribute types (where applicable). This feature also provides a set of CSS styles that let skins be applied by assigning **rich-\*** classes to particular elements, or to a container of elements that nests controls.

Standard Controls Skinning provides two levels of skinning: *Basic* and *Extended*. The level used depends on the browser type detected. If the browser type cannot be detected, **Extended** is used. However, if you want to explicitly specify the level to be applied, add a **org.richfaces.CONTROL\_SKINNING\_LEVEL** context parameter to your **web.xml** and set the value to either **basic** or **extended**.

- The *Basic* level provides customization for only basic style properties. Basic skinning is applied to the following browsers:
  - Internet Explorer 6
  - Internet Explorer 7 in BackCompat mode (see [document.compatMode property in MSDN](#))
  - Opera
  - Safari
- The *Extended* level introduces a broader number of style properties on top of basic skinning, and is applied to browsers with rich visual styling control capabilities. The following browsers support Extended skinning:
  - Mozilla Firefox
  - Internet Explorer 7 in Standards-compliant mode (CSS1Compat mode)

The following elements can be modified with skins:

- **input**
- **select**
- **textarea**
- **keygen**
- **isindex**
- **legend**
- **fieldset**

- **hr**
- **a** (together with the **a:hover**, **a:visited** pseudo-elements)

There are two ways to initialize skinning for standard HTML controls:

- add the **org.richfaces.CONTROL\_SKINNING** parameter to **web.xml**.  
**org.richfaces.CONTROL\_SKINNING** takes **enable** and **disable** as parameters. This method implies that skinning style properties are applied per-element and attribute type (where applicable). No additional steps are required. See the [Section 4.4.8.1, “Standard Level”](#) and [Section 4.4.8.2, “Extended level”](#) tables for elements to which skinning can be applied.
- add the **org.richfaces.CONTROL\_SKINNING\_CLASSES** parameter to **web.xml**.  
**org.richfaces.CONTROL\_SKINNING\_CLASSES** takes **enable** and **disable** as parameters. When enabled, you are provided with a set of predefined CSS classes to apply skins to your HTML components.

Enabling **org.richfaces.CONTROL\_SKINNING\_CLASSES** provides you style classes that can be applied to:

- basic elements nested within elements with a *rich-container* class. For example:

```
...
.rich-container select {
    //class content
}
...
```

- Elements with a class name that corresponds to one of the basic element names or types are mapped with the **rich-<elementName>[-<elementType>]** scheme, as in the following example:

```
...
.rich-select {
    //class content
}

.rich-input-text {
    //class content
}
...
```



### Note

Elements are given classes depending upon their **link** type and pseudo-class name, for example, **rich-link**, **rich-link-hover**, **rich-link-visited**

The predefined rich CSS classes provided can be used as classes for both basic and complex HTML elements.

The following code snippet shows several elements as an example:

```

...
<u:selector name=".rich-box-bgcolor-header">
    <u:style name="background-color" skin="headerBackgroundColor" />
</u:selector>
<u:selector name=".rich-box-bgcolor-general">
    <u:style name="background-color" skin="generalBackgroundColor" />
</u:selector>
...
//gradient elements
...
<u:selector name=".rich-gradient-menu">
    <u:style name="background-image">
        <f:resource
f:key="org.richfaces.renderkit.html.gradientimages.MenuGradientImage"/>
        </u:style>
        <u:style name="background-repeat" value="repeat-x" />
</u:selector>
<u:selector name=".rich-gradient-tab">
    <u:style name="background-image">
        <f:resource
f:key="org.richfaces.renderkit.html.gradientimages.TabGradientImage"/>
        </u:style>
        <u:style name="background-repeat" value="repeat-x" />
</u:selector>
...

```

For a more thorough look at standard component skinning, we recommend exploring the CSS files located in the `ui/core/src/main/resources/org/richfaces/` directory of the RichFaces SVN repository.

#### 4.4.8.1. Standard Level

**Table 4.3. HTML Element Skin Bindings for input, select, textarea, button, keygen, isindex and legend**

CSS Properties	Skin Parameters
font-size	generalSizeFont
font-family	generalFamilyFont
color	controlTextColor

**Table 4.4. HTML Element Skin Bindings for fieldset**

CSS Properties	Skin Parameters
border-color	panelBorderColor

**Table 4.5. HTML Element Skin Bindings for hr**

CSS Properties	Skin Parameters
border-color	panelBorderColor

**Table 4.6. HTML Element Skin Bindings for a**

CSS Properties	Skin Parameters
color	generalLinkColor

**Table 4.7. HTML Element Skin Bindings for a:hover**

CSS Properties	Skin Parameters
color	hoverLinkColorgeneralLinkColor

**Table 4.8. HTML Element Skin Bindings for a:visited**

CSS Properties	Skin Parameters
color	visitedLinkColor

**Table 4.9. Rich Elements Skin Bindings for .rich-input, .rich-select, .rich-textarea, .rich-keygen, .rich-isindex, .rich-link**

CSS Properties	Skin Parameters
font-size	generalSizeFont
font-family	generalFamilyFont
color	controlTextColor

**Table 4.10. Rich Element Skin Bindings for .rich-fieldset**

CSS Properties	Skin Parameters
border-color	panelBorderColor

**Table 4.11. Rich Element Skin Bindings for .rich-hr**

CSS Properties	Skin Parameters
border-color	panelBorderColor
border-width	1px
border-style	solid

**Table 4.12. Rich Element Skin Bindings for .rich-link**

CSS Properties	Skin Parameters
color	generalLinkColor

**Table 4.13. Rich Element Skin Bindings for .rich-link:hover**

CSS Properties	Skin Parameters
color	hoverLinkColor



**Table 4.14. Rich Element Skin Bindings for .rich-link:visited**

CSS Properties	Skin Parameters
color	visitedLinkColor

**Table 4.15. Rich Element Skin Bindings for .rich-field**

CSS Properties	Skin parameters/Value
border-width	1px
border-style	inset
border-color	panelBorderColor
background-color	controlBackgroundColor
background-repeat	no-repeat
background-position	1px 1px

**Table 4.16. Rich Element Skin Bindings for .rich-field-edit**

CSS Properties	Skin Parameters
border-width	1px
border-style	inset
border-color	panelBorderColor
background-color	editBackgroundColor

**Table 4.17. Rich Element Skin Bindings for .rich-field-error**

CSS Properties	Skin Parameters
border-width	1px
border-style	inset
border-color	panelBorderColor
background-color	warningBackgroundColor
background-repeat	no-repeat
background-position	center left
padding-left	7px

**Table 4.18. Rich Element Skin Bindings for .rich-button, .rich-button-disabled, .rich-button-over**

CSS Properties	Skin Parameters
border-width	1px
border-style	solid
border-color	panelBorderColor
background-color	trimColor
padding	2px 10px 2px 10px
text-align	center
cursor	pointer
background-repeat	repeat-x
background-position	top left

**Table 4.19. Rich Element Skin Bindings for .rich-button-press**

CSS Properties	Skin Parameters
background-position	bottom left

**Table 4.20. Rich Element Skin Bindings for .rich-container fieldset, .rich-fieldset**

CSS Properties	Skin Parameters
border-color	panelBorderColor
border-width	1px
border-style	solid
padding	10px
padding	10px

**Table 4.21. Rich Element Skin Bindings for .rich-legend**

CSS Properties	Skin Parameters
font-size	generalSizeFont
font-family	generalFamilyFont
color	controlTextColor
font-weight	bold

**Table 4.22. Rich Element Skin Bindings for .rich-form**

CSS Properties	Skin Parameters
padding	0px
margin	0px

#### 4.4.8.2. Extended level

**Table 4.23. HTML Element Skin Bindings for input, select, textarea, button, keygen, isindex**

CSS Properties	Skin Parameters
border-width	1px
border-color	panelBorderColor
color	controlTextColor

**Table 4.24. HTML Element Skin Bindings for \*|button**

CSS Properties	Skin Parameters
border-color	panelBorderColor
font-size	generalSizeFont
font-family	generalFamilyFont
color	headerTextColor
background-color	headerBackgroundColor
background-image	org.richfaces.renderkit.html.images.ButtonBackgroundImage

**Table 4.25. HTML Element Skin Bindings for button[type=button], button[type=reset], button[type=submit], input[type=reset], input[type=submit], input[type=button]**

CSS Properties	Skin Parameters
border-color	panelBorderColor
font-size	generalSizeFont
font-family	generalFamilyFont
color	headerTextColor
background-color	headerBackgroundColor
background-image	org.richfaces.renderkit.html.images.ButtonBackgroundImage

**Table 4.26. HTML Element Skin Bindings for \*|button[disabled], .rich-container \*|button[disabled], .rich-button-disabled**

CSS Properties	Skin Parameters
color	tabDisabledTextColor
border-color	tableFooterBackgroundColor
background-color	tableFooterBackgroundColor
background-image	org.richfaces.renderkit.html.images.ButtonDisabledBackgroundImage

**Table 4.27. HTML Element Skin Bindings for .rich-button-disabled, .rich-container button[type="button"][disabled], .rich-button-button-disabled, .rich-container button[type="reset"][disabled], .rich-button-reset-disabled, .rich-container button[type="submit"][disabled], .rich-button-submit-disabled, .rich-container input[type="reset"][disabled], .rich-input-reset-disabled, .rich-container input[type="submit"][disabled], .rich-input-submit-disabled, .rich-container input[type="button"][disabled], .rich-input-button-disabled**

CSS Properties	Skin Parameters
color	tabDisabledTextColor
background-color	tableFooterBackgroundColor
border-color	tableFooterBackgroundColor
background-image	org.richfaces.renderkit.html.images.ButtonDisabledBackgroundImage

**Table 4.28. HTML Element Skin Bindings for \*button[type="button"][disabled], button[type="reset"][disabled], button[type="submit"][disabled], input[type="reset"][disabled], input[type="submit"][disabled], input[type="button"][disabled]**

CSS Properties	Skin Parameters
color	tabDisabledTextColor
border-color	tableFooterBackgroundColor
background-color	tableFooterBackgroundColor

**Table 4.29. HTML Element Skin Bindings for \*|textarea**

CSS Properties	Skin Parameters
border-color	panelBorderColor
font-size	generalSizeFont
font-family	generalFamilyFont
color	controlTextColor
background-color	controlBackgroundColor
background-image	org.richfaces.renderkit.html.images.InputBackgroundImage

**Table 4.30. HTML Element Skin Bindings for textarea[type=textarea], input[type=text], input[type=password], select**

CSS Properties	Skin Parameters
border-color	panelBorderColor
font-size	generalSizeFont
font-family	generalFamilyFont
color	controlTextColor
background-color	controlBackgroundColor
background-image	org.richfaces.renderkit.html.images.InputBackgroundImage

**Table 4.31. HTML Element Skin Bindings for `*|textarea[disabled]`, `.rich-container` `*|textarea[disabled]`**

CSS Properties	Skin Parameters
color	tableBorderColor

**Table 4.32. `textarea[type="textarea"][disabled]`, `input[type="text"][disabled]`, `input[type="password"][disabled]`**

CSS Properties	Skin Parameters
color	tableBorderColor

**Table 4.33. `textarea[type="textarea"][disabled]`, `input[type="text"][disabled]`, `input[type="password"][disabled]`**

CSS Properties	Skin Parameters
color	tableBorderColor

**Note**

The basic skinning level can fail if the **ajaxPortlet** is configured as follows:

```
...
<portlet>
  <portlet-name>ajaxPortlet</portlet-name>
  <header-content>
    <script src="/faces/rfRes/org/ajax4jsf/framework.pack.js"
type="text/javascript" />
    <script src="/faces/rfRes/org/richfaces/ui.pack.js"
type="text/javascript" />
    <link rel="stylesheet" type="text/css"
href="/faces/rfRes/org/richfaces/skin.xcss" />
  </header-content>
</portlet>
...
```

#### 4.4.9. Client-side Script for Extended Skinning Support

Extended skinning of standard HTML controls is applied automatically: the browser type is detected, and if a browser does not fully support extended skinning, only basic skinning is applied.

There are some problems with standard HTML controls in certain browsers (Opera and Safari) that may cause problems if you wish to skin your RichFaces components and standard HTML controls manually.

To disable skinnability, set the **org.richfaces.LoadStyleStrategy** parameter to **NONE** in your **web.xml** file, like so:

```
...
<context-param>
  <param-name>org.richfaces.LoadStyleStrategy</param-name>
  <param-value>NONE</param-value>
</context-param>
...
```

You should also include the style sheets that apply skins to RichFaces components and standard HTML controls.

To work around the problem of extended skinning in Opera and Safari, the **skinning.js** client script is added to the RichFaces library. This detects the browser type and enables extended skinning only for browsers that fully support it.

Activate the script by inserting the following JavaScript into your page:

```
<script type="text/javascript">
  window.RICH_FACES_EXTENDED_SKINNING_ON = true;
</script>
```

When no script-loading strategy is used and extended skinning is enabled, a warning message appears in the console.

You must also specify the **media** attribute in the **link** tag. This adds the **extended\_both.xcss** style sheet to **rich-extended-skinning**.

To include your style sheets to the page when automatic skinnability is disabled, add the following:

```
<link href='/YOUR_PROJECT_NAME/a4j_3_2_2-
SNAPSHOTorg/richfaces/renderkit/html/css/basic_both.xcss/DATB/eAF7sqpgb-
jyGdIAFrMEaw__.jsf' type='text/css' rel='stylesheet' class='component' />
<link media='rich-extended-skinning' href='/ YOUR_PROJECT_NAME /a4j_3_2_2-
SNAPSHOTorg/richfaces/renderkit/html/css/extended_both.xcss/DATB/eAF7sqpgb-
jyGdIAFrMEaw__.jsf' type='text/css' rel='stylesheet' class='component' />
<link href='/ YOUR_PROJECT_NAME /a4j_3_2_2-
SNAPSHOTorg/richfaces/skin.xcss/DATB/eAF7sqpgb-jyGdIAFrMEaw__.jsf' type='text/css'
rel='stylesheet' class='component' />
```



## Note

The Base64 encoder now uses **!** instead of **.**, so remember to use the **a4j/versionXXX** resources prefix instead of **a4j\_versionXXX**.

### 4.4.10. XCSS File Format

Cross-site Cascading Style Sheet (XCSS) files are the core of RichFaces component skinnability. XCSS is XML-formatted CSS that extends the skinning process. RichFaces parses the XCSS file containing all look and feel parameters of a particular component and compiles the information into a standard CSS file that can be recognized by a web browser.

The XCSS file contains CSS properties and skin parameter mappings. Mapping a CSS selector to a skin parameter can be done with **< u:selector >** and **< u:style>** XML tags, which define the mapping structure, as in the following example:

```
...
<u:selector name=".rich-component-name">
  <u:style name="background-color" skin="additionalBackgroundColor" />
  <u:style name="border-color" skin="tableBorderColor" />
  <u:style name="border-width" skin="tableBorderWidth" />
  <u:style name="border-style" value="solid" />
</u:selector>
...
```

During processing, this code will be parsed and assembled into a standard CSS format, like so:

```
...
.rich-component-name {
    background-color: additionalBackgroundColor; /*the value of the constant
defined by your skin*/
    border-color: tableBorderColor; /*the value of the constant defined by your
skin*/
    border-width: tableBorderWidth; /*the value of the constant defined by your
skin*/
    border-style: solid;
}
...
```

The **name** attribute of **<u:selector>** defines the CSS selector, while the **name** attribute of the **<u:style>** tag defines the skin constant that is mapped to a CSS property. You can also use the **value** attribute of the **<u:style>** tag to assign a value to a CSS property.

CSS selectors with identical skin properties can be included in a comma-separated list:

```
...
<u:selector name=".rich-ordering-control-disabled, .rich-ordering-control-top,
.rich-ordering-control-bottom, .rich-ordering-control-up, .rich-ordering-control-
down">
  <u:style name="border-color" skin="tableBorderColor" />
</u:selector>
...
```

#### 4.4.11. Plug-n-Skin

*Plug-n-Skin* lets you easily create, customize, and plug in a custom skin to your project. You can create skins based on the parameters of predefined RichFaces skins. Plug-n-Skin also lets you unify the appearance of rich controls with standard HTML elements. This section contains step-by-step instructions for creating your own skin with Plug-n-Skin.

First, use Maven to create a template for your new skin. (You can find more information about configuring Maven for RichFaces in the [JBoss wiki article](#). These Maven instructions can be copied and pasted into the command line interface to execute them.

```
...
mvn archetype:create
-DarchetypeGroupId=org.richfaces.cdk
-DarchetypeArtifactId=maven-archetype-plugin-skin
-DarchetypeVersion=RF-VERSION
-DartifactId=ARTIFACT-ID
-DgroupId=GROUP-ID
-Dversion=VERSION
...
```

Primary keys for the command:

- **archetypeVersion** — indicates the RichFaces version; for example, **3.3.1.GA**
- **artifactId** — the artifact ID of the project
- **groupId** — the group ID of the project
- **version** — the version of the project you create. By default, this is set to **1.0.-SNAPSHOT**

This operation creates a directory named after your **ARTIFACT - ID**. The directory contains a template of the Maven project.

The following steps will guide you through creating of the skin itself.

Run the following command from the root directory of the Maven project. (This directory will contain your **pom.xml** file.)

```
...
mvn cdk:add-skin -Dname=SKIN-NAME -Dpackage=SKIN-PACKAGE
...
```

Primary keys for the command:

- **name** — defines the name of the new skin
- **package** — the base package of the skin. By default, the project's **groupId** is used.

Additional optional keys for the command:

- **baseSkin** — defines the name of the base skin.
- **createExt** — if set to **true**, extended CSS classes are added. For more information, please, see [Section 4.4.8, “Standard Controls Skinning”](#).

Refer to [Table 4.34, “Files and Folders Created By the mvn cdk:add-skin -Dname=SKIN-NAME -Dpackage=SKIN-PACKAGE Command”](#) for which files are created by the command.



**Table 4.34. Files and Fodlers Created By the mvn cdk:add-skin -Dname=SKIN-NAME -Dpackage=SKIN-PACKAGE Command**

File Name	Location	Description
<b>BaseImage.java</b>	<b>\src\main\java\SKIN-PACKAGE\SKIN-NAME\images\</b>	the base class used to store images.
BaseImageTest.java	\src\test\java\SKIN-PACKAGE\SKIN-NAME\images\	a test version of a class that stores images.
XCSS files	\src\main\resources\SKIN-PACKAGE\SKIN-NAME\css\	define the new look of RichFaces components affected by the new skin.
<b>SKIN-NAME.properties</b>	<b>\src\main\resources\SKIN-PACKAGE\SKIN-NAME\css\</b>	<p>a file that contains the new skin's properties.</p> <p>The following properties are used to configure the <b>SKIN-NAME.properties</b> file:</p> <p><b>baseSkin</b> the name of the skin to be used as a basis for your own skin. The look of the skin you define will be affected by the new style properties.</p> <p><b>generalStyleSheet</b> a path to the style sheet (<b>SKIN-NAME.xcss</b>) that imports your component's style sheets to be modified by the new skin.</p> <p><b>extendedStyleSheet</b> the path to a style sheet that is used to unify the appearance of RichFaces components and standard HTML controls. For additional information, read <a href="#">Section 4.4.8, "Standard Controls Skinning"</a>.</p> <p><b>gradientType</b> a predefined property to set the type of gradient applied to the</p>

		new skin. Possible values are <b>glass</b> , <b>plastic</b> , <b>plain</b> . More information on gradient implementation you can find further in this chapter.
<b>SKIN-NAME.xcss</b>	<b>src\main\resources\META-INF\skins</b>	an XCSS file that imports the component's XCSS files to be modified by the new skin.
XCSS files	<b>\src\main\resources\SKIN-PACKAGE\SKIN-NAME\css\</b>	determine styles for standard controls ( <b>extended_classes.xcss</b> and <b>extended.xcss</b> ), if the <b>createExt</b> key is set to <b>true</b> .
<b>SKIN-NAME-ext.xcss</b>	<b>src\main\resources\META-INF\skins.</b>	imports defining styles for standard controls if <b>createExt</b> is set to <b>true</b> .
<b>SKIN-NAME-resources.xml</b>	<b>src\main\config\resources.</b>	contains descriptions of all files listed previously.

You can now start editing the XCSS files located in **\src\main\resources\SKIN-PACKAGE\SKIN-NAME\css\**. Assign new style properties to your selectors (listed in the XCSS files) in either of the following ways:

- Standard CSS coding approach (that is, add CSS properties to the selectors). Remember that the selectors must be within **<f:verbatim> </f:verbatim>** tags. For example:

```
...
.rich-calendar-cell {
    background: #537df8;
}
...
```

- XCSS coding approach (the usual method of creating XCSS files in RichFaces). XCSS tags must be placed *outside* **<f:verbatim> </f:verbatim>** tags.

```
...
<u:selector name=".rich-calendar-cell">
    <u:style name="border-bottom-color" skin="panelBorderColor"/>
    <u:style name="border-right-color" skin="panelBorderColor"/>
    <u:style name="background-color" skin="tableBackgroundColor"/>
    <u:style name="font-size" skin="generalSizeFont"/>
    <u:style name="font-family" skin="generalFamilyFont"/>
</u:selector>
...
```

Once you have performed these steps and edited the XCSS files, build the new skin and plug it into the project. To build the skin, execute the following command from the root directory of your skin project (the directory that contains your **pom.xml** file):

```
...
mvn clean install
...
```

The Plug-n-skin feature also has a number of predefined gradients. The following code can be used to apply a gradient:

```
...
<u:selector name=".rich-combobox-item-selected">
  <u:style name="border-width" value="1px" />
  <u:style name="border-style" value="solid" />
  <u:style name="border-color" skin="newBorder" />
  <u:style name="background-position" value="0% 50%" />
  <u:style name="background-image">
    <f:resource f:key="org.richfaces.renderkit.html.CustomizeableGradient">
      <f:attribute name="valign" value="middle" />
      <f:attribute name="gradientHeight" value="17px" />
      <f:attribute name="baseColor" skin="headerBackgroundColor" />
    </f:resource>
  </u:style>
</u:selector>
...
```

The **background-image** CSS property is defined with **<f:resource f:key="org.richfaces.renderkit.html.CustomizeableGradient">**, which sets the gradient. The gradient type can be specified in the **SKIN-NAME.properties** with the **gradientType** property, which can be set to **glass**, **plastic**, or **plain**. The gradient can then be adjusted with the **baseColor**, **gradientColor**, **gradientHeight**, **valign** attributes, as seen in the previous code snippet.

You can now use your newly-created skin in your project by adding your new skin parameters to the **web.xml** file, and placing the JAR file containing your skin (located in the **target** directory of your skin project) in the **\WebContent\WEB-INF\lib\**.

```
...
<context-param>
  <param-name>org.ajax4jsf.SKIN</param-name>
  <param-value>SKIN-NAME</param-value>
</context-param>
...
```

#### 4.4.11.1. Details of Usage

This section covers some practical aspects of Plug-n-Skin implementation. We assume that you have read the section of the guide describing the Plug-n-Skin prototype creation process.

First, we must create a new skin (as described in the previous section). The following creates a template of the new skin project:

```
mvn archetype:create
-DarchetypeGroupId=org.richfaces.cdk
-DarchetypeArtifactId=maven-archetype-plug-n-skin
-DarchetypeVersion=3.3.1.GA
-DartifactId=P-n-S
-DgroupId=GROUPID
-Dversion=1.0.-SNAPSHOT
```

You can now browse the **P-n-S** directory to view the files and folders created.

Next, use Maven to add all required files to the skin project, like so:

```
mvn cdk:add-skin -DbaseSkin=blueSky -DcreateExt=true -Dname=PlugnSkinDemo -
Dpackage=SKINPACKAGE
```

As mentioned in the previous section, **-DbaseSkin** defines the RichFaces built-in skin to use as a base, and **-DcreateExt=true**, which determines that the new skin will include XCSS files that unify the appearance of the rich components and the standard HTML controls.

Once your resources have been created, you can begin refining the newly-created skin. Begin by editing the rich components' XCSS files.

As an example of the Plug-n-Skin feature, we will edit some **<rich:calendar>** style attributes and some basic HTML controls. We will show you how to:

- Recolor the background of the current day in the **<rich:calendar>**;
- Recolor a standard HTML submit button;

To edit **<rich:properties>**'s style properties, you must open the **calendar.xcss** file, located in **P-n-S\src\main\resources\skinpackage\plugnskindemo\css\**.

In the **calendar.xcss** file, find the **.rich-calendar-today** selector and amend it as follows: **background-color: #075ad1;**. This will change the background color of the current day.

Next we will change the font style of a standard HTML *submit* button. Open the **extended.xcss** file from the **P-n-S\src\main\resources\skinpackage\plugnskindemo\css\** directory and insert **font-weight: bold;** between the curly braces of these selectors, like so:

```
button[type="button"], button[type="reset"], button[type="submit"],
input[type="reset"], input[type="submit"], input[type="button"] {
    font-weight: bold;
}
```

The desired changes have now been made, and you can proceed to building the new **PlugnSkinDemo** skin and importing it into the project.

Build the skin by executing **mvn clean install** from the **P-n-S** directory. This creates a **target** directory containing a JAR file with a newly-compiled skin. In our case, the file is named **P-n-S-1.0.-SNAPSHOT.jar**.

Next, import the new **PlugnSkinDemo** skin into your project:

- Copy the **P-n-S-1.0.-SNAPSHOT.jar** file to the **\WebContent\WEB-INF\lib\** directory.
- Add the name of the new skin to the **web.xml** file, like so:

```
<context-param>
  <param-name>org.ajax4jsf.SKIN</param-name>
  <param-value>PlugnSkinDemo</param-value>
</context-param>
```

Remember, standard controls skinning must be enabled in **web.xml**. Add the following to enable standard controls skinning:

```

<context-param>
  <param-name>org.richfaces.CONTROL_SKINNING</param-name>
  <param-value>enable</param-value>
</context-param>

```

The results of each alteration to the skin are shown in the figures that follow:

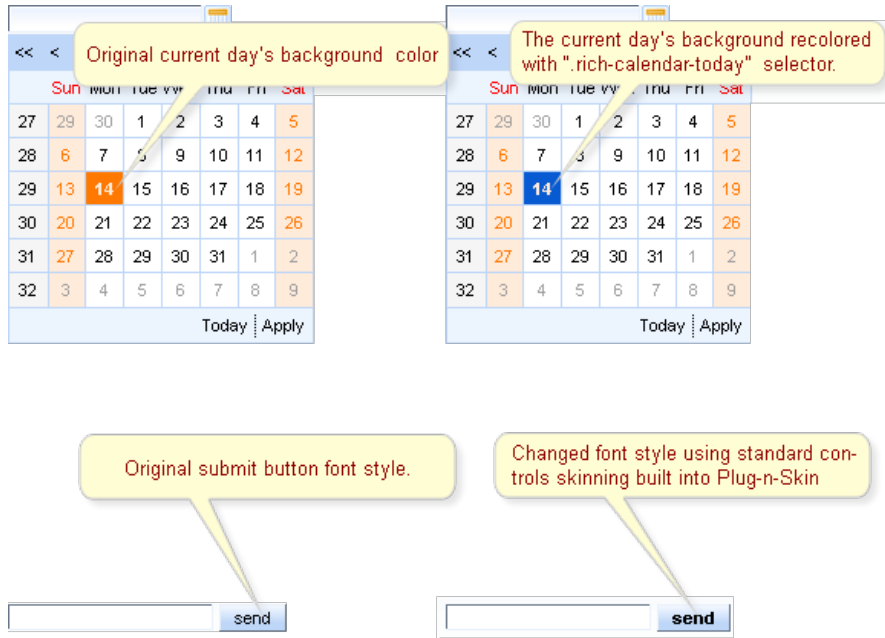


Figure 4.6. Plug-n-Skin feature in action.

## 4.5. State Manager API

JSF has an advanced navigation mechanism that lets you define *navigation* from view to view. In a web application, navigation occurs when a user changes from one page to another by clicking on a button, a hyperlink, or another command component. There is no switch mechanism between some logical states of the same view. For example, in *Login/Register dialog*, an existing user signs in with his user name and password, but if a new user attempts to register, an additional field (**Confirm**) is displayed, and button labels and methods are changed when the user clicks the **To register** link:

The figure shows a login dialog box titled "Login Existing User (To register)". Inside the dialog, there are two input fields: "username" and "password". Below these fields is a "Login" button. The dialog has a light blue border and a title bar.

Figure 4.7. Login Dialog

Register New User (To login)

username

password

confirm

Figure 4.8. Register Dialog

*RichFaces State API* lets you easily define a set of states for pages, and properties for these states.

The **States** class interfaces with a map, where the **keySet** defines the State name and the **entrySet** is a **State** map. The **State** map defines the properties, method bindings, or constant state variables of a key or object, where these values may change depending on the active State.

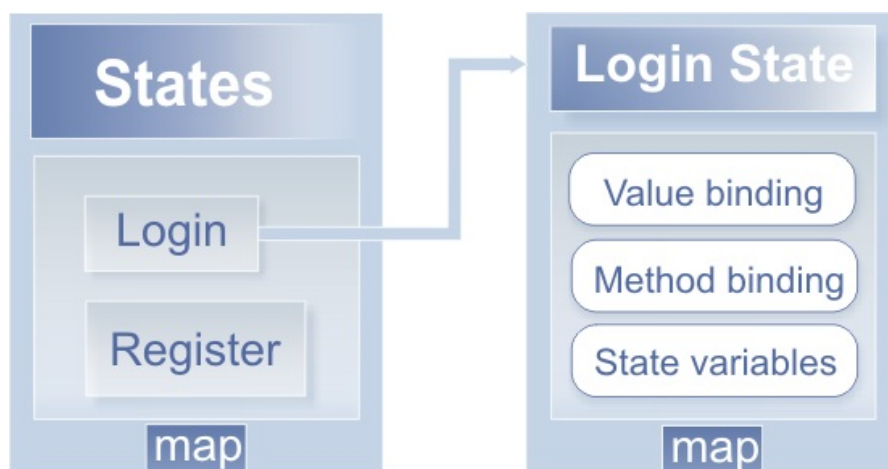


Figure 4.9. RichFaces State API

One of the most convenient features of the RichFaces State API is the ability to navigate between **States**. The API implements changes in **State** through standard JSF navigation. When the action component returns an outcome, the JSF navigation handler (extended through the RichFaces State API) checks whether the outcome is registered as a **State change outcome**. If **true**, the corresponding **State** is activated. If **false**, standard navigation handling is called.

Implement the RichFaces State API like so:

- Register the State Navigation Handler and the EL Resolver in your **faces-config.xml** file:

```
...
<application>
  <navigation-
    handler>org.richfaces.ui.application.StateNavigationHandler</navigation-handler>
    <el-resolver>org.richfaces.el.StateELResolver</el-resolver>
  </application>
...
```

- Register an additional application factory in the **faces-config.xml**:

```
...
<factory>
  <application-
factory>org.richfaces.ui.application.StateApplicationFactory</application-
factory>
</factory>
...
```

- Register two managed beans in the **faces-config.xml**:

```
...
<managed-bean>
  <managed-bean-name>state</managed-bean-name>
  <managed-bean-class>org.richfaces.ui.model.States</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>states</property-name>
    <property-class>org.richfaces.ui.model.States</property-class>
    <value>#{config.states}</value>
  </managed-property>
</managed-bean>
<managed-bean>
  <managed-bean-name>config</managed-bean-name>
  <managed-bean-class>org.richfaces.demo.stateApi.Config</managed-bean-class>
  <managed-bean-scope>none</managed-bean-scope>
</managed-bean>
...
```

One bean (**config**) defines and stores **State** as seen in the following example:

```

...
public class Config {

    /**
     * @return States
     */
    public States getStates() {
        FacesContext facesContext = FacesContext.getCurrentInstance();
        States states = new States();

        // Registering new User State definition
        states.setCurrentState("register"); // Name of the new state

        // Text labels, properties and Labels for controls in "register" state
        states.put("showConfirm", Boolean.TRUE); // confirm field rendering
        states.put("link", "(To login)"); // Switch State link label
        states.put("okBtn", "Register"); // Login/Register button label
        states.put("stateTitle", "Register New User"); // Panel title

        ExpressionFactory expressionFactory = facesContext.getApplication()
            .getExpressionFactory();

        // Define "registerbean" available under "bean" EL binding on the page
        ValueExpression beanExpression = expressionFactory
            .createValueExpression(facesContext.getELContext(),
                "#{registerbean}", Bean.class);
        states.put("bean", beanExpression);

        // Define "registeraction" available under "action" EL binding on the
        // page
        beanExpression = expressionFactory.createValueExpression(facesContext
            .getELContext(), "#{registeraction}", RegisterAction.class);
        states.put("action", beanExpression);

        // Define method expression inside registeraction binding for this state
        MethodExpression methodExpression = expressionFactory.createMethodExpression(
            facesContext.getELContext(), "#{registeraction.ok}",
            String.class, new Class[] {});
        states.put("ok", methodExpression);

        // Outcome for switching to login state definition
        states.setNavigation("switch", "login");

        // Login Existent User State analogous definition
        states.setCurrentState("login");
        states.put("showConfirm", Boolean.FALSE);
        states.put("link", "(To register)");
        states.put("okBtn", "Login");
        states.put("stateTitle", "Login Existing User");

        beanExpression = expressionFactory.createValueExpression(facesContext
            .getELContext(), "#{loginbean}", Bean.class);
        states.put("bean", beanExpression);

        beanExpression = expressionFactory.createValueExpression(facesContext
            .getELContext(), "#{loginaction}", LoginAction.class);
        states.put("action", beanExpression);

        methodExpression = expressionFactory.createMethodExpression(
            facesContext.getELContext(), "#{loginaction.ok}",

```



```

        String.class, new Class[] {}));
        states.put("ok", methodExpression);

        states.setNavigation("switch", "register");

        return states;
    }
}
...

```

The second bean, with the **org.richfaces.ui.model.States** type (**state**), contains the managed property states, which is bound to the first **config** bean.

- Next, use state bindings on the page, as in the following example:

```

...
<h:panelGrid columns="3">
    <h:outputText value="username" />
    <h:inputText value="#{state.bean.name}" id="name" required="true" />
    <h:outputText value="password" />
    <h:inputSecret value="#{state.bean.password}" id="password" required="true" />
    <h:outputText value="confirm" rendered="#{state.showConfirm}" />
    <h:inputSecret value="#{state.bean.confirmPassword}"
        rendered="#{state.showConfirm}" id="confirm" required="true" />
</h:panelGrid>
<a4j:commandButton actionListener="#{state.action.listener}" action="#{state.ok}"
    value="#{state.okBtn}" id="action"/>
...

```

To see complete example of the Login/Register dialog, see the [RichFaces Live Demo](#).

## 4.6. Identifying User Roles

RichFaces also lets you check whether the logged-in user belongs to a certain user role with the **rich:isUserInRole(Object)** function. This function takes a String or a comma-delineated list of Strings, a Collection, etc. as arguments and returns a Boolean value.

As an example, imagine that you need to render some controls only for administrators. To do so, create an administrator role (**admin**) in your **web.xml** file. Then implement authorization that assigns the **admin** role to the user that has logged in as an administrator. Once this has been done, you can use the **rich:isUserInRole(Object)** function with the **rendered** attribute of any component. For example:

```

...
<rich:editor value="#{bean.text}" rendered="#{rich:isUserInRole('admin')}" />
...

```

Here, only a logged-in user with an **admin** role can see the text editor, which will not be rendered for users with other roles.

## Revision History

<b>Revision 5.1.1-104.400</b>	<b>2013-10-31</b>	<b>Rüdiger Landmann</b>
Rebuild with publican 4.0.0		
<b>Revision 5.1.1-104</b>	<b>2012-07-18</b>	<b>Anthony Towns</b>
Rebuild for Publican 3.0		
<b>Revision 5.1.1-100</b>	<b>Mon Jul 18 2011</b>	<b>Jared Morgan</b>
Incorporated changes for JBoss Enterprise Web Platform 5.1.1 GA. For information about documentation changes to this guide, refer to <i>Release Notes 5.1.1</i> .		