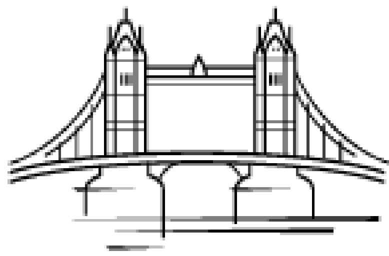


PortletFaces Bridge

Reference Documentation

2.0.1

PortletFaces Bridge

Copyright © 2010-2011 portletfaces.org

Legal Notice

Copyright © 2010-2011 by portletfaces.org. This copyrighted material is made available to anyone wishing to use, modify, copy, or redistribute it subject to the terms and conditions of the Apache License, Version 2.0.

1. Portlet Standard ... 1
1.1. Overview ... 1
1.2. Portlet Lifecycle ... 1
1.3. Portlet Modes ... 2
1.4. Portlet Window States ... 3
1.5. Portlet Preferences ... 3
1.6. Inter-Portlet Communication ... 4
2. Portlet Bridge Standard ... 7
2.1. Overview ... 7
2.2. Portlet Bridge 1.0 ... 7
2.3. Portlet Bridge 2.0 ... 7
2.4. Portlet Bridge 3.0 ... 7
2.5. Portlet Lifecycle and JSF Lifecycle ... 8
3. PortletFaces Bridge Configuration ... 9
3.1. Overview ... 9
3.2. Bridge Request Scope ... 9
3.3. PreDestroy & BridgePreDestroy Annotations ... 11
3.4. Portlet Container Abilities ... 13
3.5. Portlet Namespace Optimization ... 14
3.6. Resolving XML Entities ... 15
3.7. Resource Buffer Size ... 15
4. JSF Portlet Development ... 17
4.1. Overview ... 17
4.2. The PortletFaces Bridge ... 17
4.3. JSF and PortletPreferences ... 18
4.4. JSF ExternalContext and the Portlet API ... 22
4.4.1. Getting the PortletRequest and PortletResponse Objects ... 22
4.5. JSF and Inter-Portlet Communication ... 23
4.5.1. Portlet 2.0 Public Render Parameters ... 24
4.5.2. Portlet 2.0 Events ... 26
4.5.3. Portlet 2.0 Shared Session Scope ... 28
5. JSF Component Tags ... 31
5.1. Overview ... 31
5.2. Bridge UIComponent Tags ... 31
5.2.1. The bridge:inputFile tag ... 31
5.3. Portlet 2.0 UIComponent Tags ... 33
5.3.1. The portlet:actionURL tag ... 34
5.3.2. The portlet:namespace tag ... 35
5.3.3. The portlet:param tag ... 36
5.3.4. The portlet:renderURL tag ... 37
5.3.5. The portlet:resourceURL tag ... 38
6. Liferay Portal ... 41

6.1. Overview 41
6.2. JavaScript Concerns 41
7. ICEfaces 2 Portlet Development 43
7.1. Overview 43
7.2. ICEfaces Direct-To-DOM RenderKit 43
7.3. ICEfaces Ajax Push and Inter-Portlet Communication 44
7.4. ICEfaces Themes and Portal Themes 47
7.5. ICEfaces Themes and Liferay Themes 49

Chapter 1. Portlet Standard

1.1. Overview

Portlets are web applications that are designed to run inside a portlet container that implements either the Portlet 1.0 (JSR 168) or Portlet 2.0 (JSR 286) standard. Portlet containers provide a layer of abstraction over the Java EE Servlet API, and consequently require a servlet container like Apache Tomcat to function. The reference implementation for Portlet 1.0 and 2.0 is the Apache Pluto project: http://portals.apache.org/pluto.

Portals are standalone systems that use a portlet container as the runtime engine for executing portlets. When a portal is asked to deliver a portal page to the end-user's web browser, each portlet is asked to render itself as a fragment of HTML. It is the job of the portal to aggregate these HTML fragments into a complete HTML document.

1.2. Portlet Lifecycle

The Portlet 1.0 standard defines two lifecycle phases for the execution of a portlet that a compliant portlet container must support: The first is the javax.portlet.PortletRequest.RENDER_PHASE, in which the portlet container asks each portlet to render itself as a fragment of HTML. The second is the javax.portlet.PortletRequest.ACTION_PHASE, in which the portlet container invokes actions related to HTML form submission. When the portal receives an HTTP GET request for a portal page, the portlet container executes the portlet lifecycle and each of the portlets on the page undergoes the RENDER_PHASE. When the portal receives an HTTP POST request, the portlet container executes the portlet lifecycle and the portlet associated with the HTML form submission will first undergo the ACTION_PHASE before the RENDER_PHASE is invoked for all of the portlets on the page.

The Portlet 2.0 standard adds two more lifecycle phases that define the execution of a portlet. The first is the javax.portlet.PortletRequest.EVENT_PHASE, in which the portlet container broadcasts events that are the result of an HTML form submission. During this phase, the portlet container asks each portlet to process events that they are interested in. The typical use case for the EVENT_PHASE is to achieve Inter-Portlet Communication (IPC), whereby two or more portlets on a portal page share data in some way. The other new phase added by the Portlet 2.0 standard is the javax.portlet.PortletRequest.RESOURCE_PHASE, in which the portlet container asks a specific portlet to perform resource-related processing. One typical use case for the RESOURCE_PHASE is for an

individual portlet to process Ajax requests. Another typical use case for the RESOURCE_PHASE is for an individual portlet to generate non-HTML content (for download purposes) such as a PDF or spreadsheet document.

1.3. Portlet Modes

The Portlet 1.0 and 2.0 standards define three portlet modes that a compliant portlet container must support: javax.portlet.PortletMode.VIEW, javax.portlet.PortletMode.EDIT, and javax.portlet.PortletMode.HELP. Portal vendors and portlet developers may supply custom modes as well. VIEW mode refers to the rendered portlet markup that is encountered by the user under normal circumstances. Perhaps a clearer name would be "normal" mode or "typical" mode, because the word "view" is also used by developers to review to the "view" concern of the MVC design pattern. EDIT mode refers to the rendered portlet markup that is encountered by the user when selecting custom values for portlet preferences. Perhaps a clearer name would be "preferences" mode. Finally, HELP mode refers to the rendered portlet markup that is encountered by the user when seeking help regarding the usage and/or functionality of the portlet.

Figure 1.1. Portlet VIEW Mode

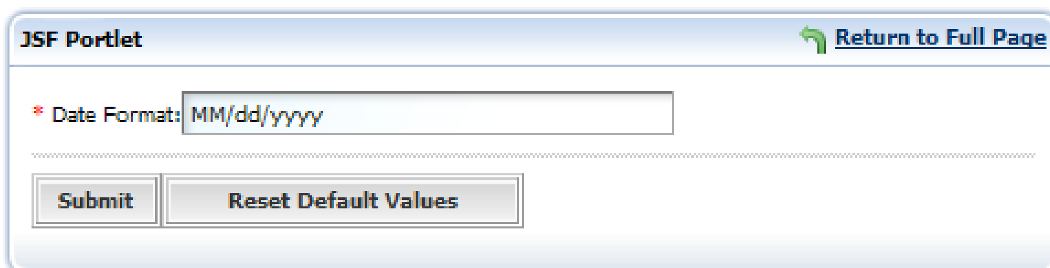


Figure 1.2. Portlet EDIT Mode

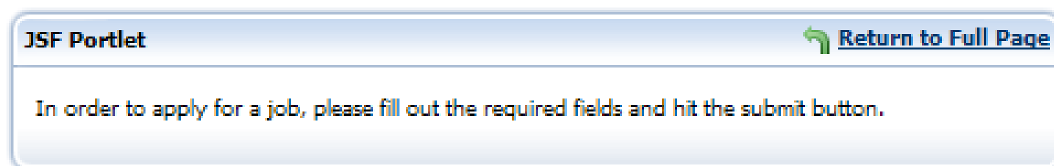


Figure 1.3. Portlet HELP Mode

1.4. Portlet Window States

Portals typically manifest the rendered markup of a portlet in a rectangular section of the browser known as a portlet window. The Portlet 1.0 and 2.0 standards define three window states that a compliant portlet container must support: javax.portlet.WindowState.NORMAL, javax.portlet.WindowState.MAXIMIZED, and javax.portlet.WindowState.MINIMIZED. The NORMAL window state refers to the way in which the portlet container displays the rendered markup of a portlet when it can appear on the same portal page as other portlets. The MAXIMIZED window state refers to the way in which the portlet container displays the rendered markup of a portlet when it is the only portlet on a page, or when the portlet is to be rendered more prominently than other portlets on a page. Finally, the MINIMIZED window state refers to the way in which the portlet container displays a portlet when the markup is not to be rendered.

1.5. Portlet Preferences

Developers often have the requirement to provide the end-user with the ability to personalize the portlet behavior in some way. To meet this requirement, the Portlet 1.0 and 2.0 standards provide the ability to define preferences for each portlet. Preference names and default values can be defined in the WEB-INF/portlet.xml configuration file. Portal end-users start out interacting with the portlet user interface in portlet VIEW mode but can switch to portlet EDIT mode in order to select custom preference values.

Example 1.1. Specifying preference names and associated default values in the WEB-INF/portlet.xml configuration file

```
<portlet-app>
 <portlet>
  ...
  <portlet-preferences>
   <preference>
    <name>datePattern</name>
    <value>MM/dd/yyyy</value>
   </preference>
   <preference>
    <name>unitedStatesPhoneFormat</name>
    <value>###-###-####</value>
   </preference>
  </portlet-preferences>
  ...
 </portlet>
</portlet-app>
```

1.6. Inter-Portlet Communication

Inter-Portlet Communication (IPC) is a technique whereby two or more portlets on a portal page share data in some way. In a typical IPC use case, user interactions with one portlet affect the rendered markup of another portlet. The Portlet 2.0 standard provides two techniques to achieve IPC: Public Render Parameters and Server-Side Events.

The Public Render Parameters technique provides a way for portlets to share data by setting public/shared parameter names in a URL controlled by the portal. While the benefit of this approach is that it is relatively easy to implement, the drawback is that only small amounts of data can be shared. Typically the kind of data that is shared is simply the value of a database primary key.

The Server-Side Events technique provides a way for portlets to share data using an event-listener design. When using this form of IPC, the portlet container acts as broker and distributes events and payload (data) to portlets. One requirement of this approach is that the payload must implement the java.io.Serializable interface since it might be sent to a portlet in another WAR running in a different classloader.

It could be argued that the Portlet 2.0 approaches for IPC have a common drawback in that they can lead to a potentially disruptive end-user experience. This is because they cause either an HTTP GET or HTTP POST which results in a full page refresh. Technologies such as ICEfaces Ajax

Push can be used to solve this problem. Refer to the Ajax Push IPC section of this document for more details.

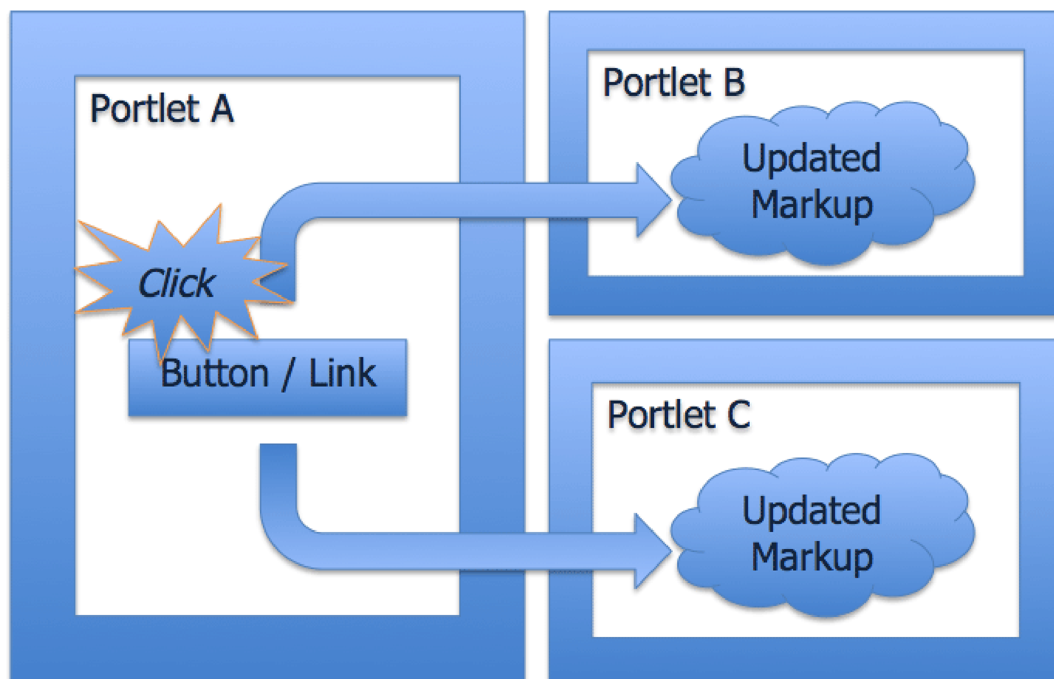


Figure 1.4. Illustration of Standard Portlet 2.0 IPC

Chapter 2. Portlet Bridge Standard

2.1. Overview

The Portlet 1.0 and JSF 1.0 specifications were formulated during roughly the same timeframe. Consequently, the JSF Expert Group (EG) was able to design a framework with portlet compatibility in mind. This is evidenced by methods like ExternalContext.getRequest() which returns a value of type `Object`, rather than a value of type javax.servlet.http.HttpServletRequest. When running inside a portlet container, the same method would return a value of type javax.portlet.PortletRequest. Although the JSF API provides a degree of portlet compatibility, it is necessary to introduce a bridge between the JSF lifecycle and the Portlet lifecycle in order to run JSF applications as portlets.

2.2. Portlet Bridge 1.0

Starting in 2004, several different JSF portlet bridge implementations were developed in order to provide JSF developers with the ability to deploy their JSF webapps as portlets. In 2006 the JCP formed the Portlet Bridge 1.0 (JSR 301) EG in order to define a standard bridge API as well as detailed requirements for bridge implementations. JSR 301 was released in 2010 and targeted Portlet 1.0 and JSF 1.2.

2.3. Portlet Bridge 2.0

When the Portlet 2.0 (JSR 286) standard was released in 2008 it became necessary for the JCP to form the Portlet Bridge 2.0 (JSR 329) EG. JSR 329 was also released in 2010 and targeted Portlet 2.0 and JSF 1.2.

2.4. Portlet Bridge 3.0

After the JSR 314 EG released JSF 2.0 in 2009 and JSF 2.1 in 2010, it became evident that a Portlet Bridge 3.0 standard would be beneficial. At the time of this writing, the JCP has not formed such an EG. In the meantime, the developers at portletfaces.org began work on the **PortletFaces Bridge** project, which targets Portlet 2.0 and JSF 2.0/2.1.

The PortletFaces Bridge is inspired by an Early Draft Revision (EDR) of the JSR 329 Portlet Bridge Standard. However, since JSR 329 defines a bridge API for Portlet 2.0 + JSF 1.2, the PortletFaces Bridge cannot fulfill the license terms and therefore ships with it's own API with the org.portletfaces.bridge namespace instead of the javax.portlet.faces namespace. It also contains

innovative features that make it possible to leverage the power of JSF 2 inside a portlet application.

2.5. Portlet Lifecycle and JSF Lifecycle

JSF portlet bridges are responsible for providing a “bridge” between the portlet lifecycle and the JSF lifecycle. For example, when a portal page that contains a JSF portlet is requested via HTTP GET, then the RENDER_PHASE of the portlet lifecycle should in turn execute the RESTORE_VIEW and RENDER_RESPONSE phases of the JSF lifecycle. Similarly, when the user submits a form contained within a JSF portlet via HTTP POST, then the ACTION_PHASE of the portlet lifecycle should execute the complete JSF lifecycle of RESTORE_VIEW, APPLY_REQUEST_VALUES, PROCESS_VALIDATIONS, UPDATE_MODEL_VALUES, INVOKE_APPLICATION, and RENDER_RESPONSE. Since the portal is in full control of managing URLs, JSF portlet bridges are also responsible for asking the portal to generate URLs that are compatible with actions that invoke JSF navigation rules. If a different JSF view is to be rendered as a result of a JSF navigation-rule, then the JSF portlet bridge simply displays the new JSF view in the same portlet window.

Chapter 3. PortletFaces Bridge Configuration

3.1. Overview

The JSR 329 standard defines several configuration options prefixed with the javax.portlet.faces namespace. The PortletFaces Bridge defines additional implementation specific options prefixed with the org.portletfaces.bridge namespace.

3.2. Bridge Request Scope

One of the key requirements in creating a JSF portlet bridge is managing JSF request-scoped data within the Portlet lifecycle. This is normally referred to as the "Bridge Request Scope" by JSR 329. The lifespan of the BridgeRequestScope works like this:

1. ActionRequest/EventRequest: BridgeRequestScope begins
2. RenderRequest: BridgeRequestScope is preserved
3. Subsequent RenderRequest: BridgeRequestScope is reused
4. Subsequent ActionRequest/EventRequest: BridgeRequestScope ends, and a new BridgeRequestScope begins
5. If the session expires or is invalidated, then similar to the PortletSession scope, all BridgeRequestScope instances associated with the session are made avialable for garbage collection by the JVM

The main use-case for having the BridgeRequestScope preserved in #2 (above) is for "re-render" of portlets. One example would be when two or more JSF portlets are placed on a portal page (Portlets X and Y), and those portlets are not using f:ajax for form submission. In such a case, if the user were to submit a form (via full ActionRequest postback) in Portlet X, and then submit a form in Portlet Y, then Portlet X should be re-rendered with its previously submitted form data.

With the advent of JSF 2 and Ajax, there are four drawbacks for supporting this use-case as the default behavior:

- Request-scoped data basically semi-session-scoped in nature, because the BridgeRequestScope is preserved (even though the user might NEVER click the Submit button again).

- BridgeRequestScope can't be stored in the PortletSession because the data is Request-scoped in nature, and the data stored in the scope isn't guaranteed to be Serializable for replication. Therefore it doesn't really work well in a clustered deployment.

- The developer might have to specify the javax.portlet.faces.MAX_MANAGED_REQUEST_SCOPES init-param in the WEB-INF/web.xml descriptor in order to tune the memory settings on the server.

- The developer is forced to add a <listener> for the BridgeSessionListener to the WEB-INF/web.xml descriptor.

Therefore, since the PortletFaces Bridge is designed for JSF 2 and Ajax in mind, the bridge makes the following assumptions:

- That developers are not primarily concerned about the "re-render" of portlets use-case mentioned above.

- That developers don't want any of the drawbacks mentioned above.

- That developers are making heavy use of the f:ajax tag (or implicitly doing so with ICEfaces) and submitting forms via Ajax with their modern-day portlets.

- That developers want to be as zero-config as possible, and don't want to be forced to add anything to the WEB-INF/web.xml descriptor.

Consequently, the default behavior of the PortletFaces Bridge is to cause the BridgeRequestScope to end at the end of the RenderRequest. If the standard behavior is desired, then the following options can be placed in the WEB-INF/web.xml descriptor.

Example 3.1. Specifying standard behavior for BridgeRequestScope lifespan via the WEB-INF/web.xml descriptor

```
<!-- The default value of the following -->
<!-- context-param is false, meaning that -->
<!-- The PortletFaces Bridge will cause -->
<!-- the BridgeRequestScope to end after -->
<!-- the RENDER_PHASE of the portlet -->
<!-- lifecycle. Setting the value to true -->
<!-- will cause the PortletFaces Bridge to -->
<!-- cause the BridgeRequestScope to last -->
<!-- until the next ACTION_PHASE or -->
<!-- EVENT_PHASE of the portlet lifecycle. -->
<context-param>
 <param-name>org.portletfaces.bridgeRequestScopePreserved</param-
name>
 <param-value>true</param-value>
</context-param>
<!-- The default value of the following -->
<!-- context-param is 100. It defines -->
<!-- the maximum number of BridgeRequestScope -->
<!-- instances to keep in memory on the server -->
<!-- if the bridgeRequestScopePreserved -->
<!-- option is true. -->
<context-param>
 <param-name>javax.portlet.faces.MAX_MANAGED_REQUEST_SCOPES</param-
name>
 <param-value>2000</param-value>
</context-param>
<!-- The following listener is required to cleanup -->
<!-- BridgeRequestScope instances upon session timeout -->
<listener>
 <listener-
class>org.portletfaces.bridge.servlet.BridgeSessionListener</
listener-class>
</listener>
```

Alternatively, the org.portletfaces.bridgeRequestScopePreserved value can be specified on a portlet-by-portlet basis in the WEB-INF/portlet.xml descriptor.

3.3. PreDestroy & BridgePreDestroy Annotations

When JSF developers want to perform cleanup on managed-beans before they are destroyed, they typically annotate a method inside the bean with the @PreDestroy annotation. However, section 6.8.1 of the JSR 329 standard discusses the need for the @BridgePreDestroy and @BridgeRequestScopeAttributeAdded annotations in the bridge API.

In-Depth Discussion Avialable Online

For a in-depth discussion of this issue, please refer to http://jira.portletfaces.org/browse/BRIDGE-146

In order to explain this requirement, it is necessary to make a distinction between *local* portals and *remote* portals. Local portals invoke portlets that are deployed within the same (local) servlet container. Remote portals invoke portlets that are deployed elsewhere via WSRP (Web Services for Remote Portlets). The @BridgePreDestroy and @BridgeRequestScopeAttributeAdded annotations were introduced into the JSR 329 standard primarily to support WSRP in remote portals. That being the case, the standard indicates that developers should always use @BridgePreDestroy instead of @PreDestroy. The PortletFaces Bridge however takes a different approach: rather than assuming the remote portal use-case, the PortletFaces Bridge assumes the local portal use-case. When developing with a local portal like Liferay, the PortletFaces Bridge ensures that the standard @PreDestroy annotation works as expected. This means there is no reason to use the @BridgeRequestScope annotation with a local portal when using the PortletFaces Bridge. Developers must manually configure the PortletFaces Bridge via the WEB-INF/web.xml descriptor in order to leverage the @BridgePreDestroy and @BridgeRequestScopeAttributeAdded annotations for WSRP.

Example 3.2. Specifying support for @BridgePreDestroy and @BridgeRequestScopeAttributeAdded in the WEB-INF/web.xml descriptor

```
<!-- The default value of the following -->
<!-- context-param is false, meaning that -->
<!-- The PortletFaces Bridge will invoke -->
<!-- methods annotated with @PreDestroy -->
<!-- over those annoated with -->
<!-- @BridgePreDestroy. -->
<!-- Setting the value of the following -->
<!-- context-param instructs the PortletFaces -->
<!-- Bridge to prefer the @BridgePreDestroy -->
<!-- annotation over the standard @PreDestroy -->
<!-- annotation in order to support a WSRP -->
<!-- remote portal environment. -->
<context-param>
 <param-name>org.portletfaces.bridge.preferPreDestroy</param-name>
 <param-value>false</param-value>
</context-param>
<!-- The following listener is required to support -->
<!-- the @BridgeRequestScopeAttributeAdded -->
<!-- annotation in a WSRP remote portal environment. -->
<listener>
 <listener-
class>org.portletfaces.bridge.context.map.BridgeRequestAttributeListener</
listener-class>
</listener>
```

Alternatively, the org.portletfaces.bridge.preferPreDestroy value can be specified on a portlet-by-portlet basis in the WEB-INF/portlet.xml descriptor.

3.4. Portlet Container Abilities

The PortletFaces Bridge can be run in a variety of portlet containers (Liferay, Pluto, etc.) and is aware of some of the abilities (or limitations) of these containers. Regardless, the PortletFaces Bridge enables the developer to configure the abilities of the portlet container in the WEB-INF/web.xml descriptor.

Example 3.3. Setting portlet container abilities via the WEB-INF/web.xml descriptor

```
<!-- The default value of the following -->
<!-- context-param depends on which -->
<!-- portlet container the bridge -->
<!-- is running in. The value determines -->
<!-- whether or not the bridge resource -->
<!-- handler will attempt to set the -->
<!-- status code of downloaded resources -->
<!-- to values like -->
<!-- HttpServletResponse.SC_NOT_MODIFIED -->
<context-param>
 <param-
name>org.portletfaces.bridge.containerAbleToSetHttpStatusCode</
param-name>
 <param-value>true</param-value>
</context-param>
```

3.5. Portlet Namespace Optimization

The JSR 329 standard requires the bridge implementation to prepend the portlet namespace to the value of the "id" attribute of every component that is rendered by a JSF view. This guarantees the uniqueness of the "id" attribute when there are multiple JSF portlets on a portal page that contain similar component hierarchies and naming. Also, the JSR 329 standard indicates that the bridge implementation of the ExternalContext.encodeNamesapce(String) method is to prepend the value of javax.portlet.PortletResponse.getNamespace() to the specified String. The problem is that since the value returned by getNamespace() can be a lengthy string, the size of the rendered HTML portal page can become unnecessarily large. This can be especially non-performant when using the f:ajax tag in a Facelet view in order to perform partial-updates the browser's DOM.

The PortletFaces Bridge has a built-in optimization that minimizes the value returned by the the ExternalContext.encodeNamesapce(String) method, while still guaranteeing uniqueness. Developers must manually configure the PortletFaces Bridge via the WEB-INF/web.xml descriptor in order to disable the namespace optimization and leverage the default behavior specified by JSR 329.

Example 3.4. Disabling the namespace optimization via the WEB-INF/web.xml descriptor

```
<!-- The default value of the following -->
<!-- context-param is true, meaning that -->
<!-- The PortletFaces Bridge will optimize -->
<!-- the portlet namespace. Setting the value -->
<!-- of the following context-param to false -->
<!-- disables the optimization. -->
<context-param>
 <param-name>org.portletfaces.bridge.optimizePortletNamespace</
param-name>
 <param-value>false</param-value>
</context-param>
```

3.6. Resolving XML Entities

The PortletFaces Bridge provides the ability to set a flag indicating whether or not XML entities are required to be resolved when parsing faces-config.xml files in the classpath. The default value of this option is false.

Example 3.5. Specifying the resolving of XML entities via the WEB-INF/web.xml descriptor

```
<!-- The default value of the following -->
<!-- context-param is false. -->
<context-param>
 <param-name>org.portletfaces.bridge.resolveXMLEntities</param-name>
 <param-value>true</param-value>
</context-param>
```

3.7. Resource Buffer Size

The PortletFaces Bridge provides the ability to set the size of the buffer used to load resources into memory as the file contents are being copied to the response. The default value of this option is 1024 (1KB).

Example 3.6. Specifying the resource buffer size via the WEB-INF/web.xml descriptor

```
<!-- The default value of the following -->
<!-- context-param is 1024. -->
<context-param>
 <param-name>org.portletfaces.bridge.resourceBufferSize</param-name>
 <param-value>4096</param-value>
</context-param>
```

Alternatively, the org.portletfaces.resourceBufferSize value can be specified on a portlet-by-portlet basis in the WEB-INF/portlet.xml descriptor.

Chapter 4. JSF Portlet Development

4.1. Overview

The main goal of JSF portlet bridges is to make the JSF portlet development experience as close as possible to JSF webapp development. Consequently, many JSF webapps can be easily migrated to a portlet container using such a bridge.

4.2. The PortletFaces Bridge

In order to use JSF2 in a portlet, developers must specify the **PortletFaces Bridge** in the `<portlet-class>` element of the `WEB-INF/portlet.xml` descriptor.

Example 4.1. Specifying the PortletFaces Bridge in the WEB-INF/portlet.xml configuration file, as well as default Facelet views that are to be rendered for VIEW mode, EDIT mode, and HELP mode

```
<portlet-app>
 <portlet>
  <portlet-name>my_portlet</portlet-name>
  <display-name>My Portlet</display-name>
  <portlet-class>
   org.portletfaces.bridge.GenericFacesPortlet
  </portlet-class>
  <init-param>
   <name>javax.portlet.faces.defaultViewId.view</name>
   <value>/xhtml/applicantForm.xhtml</value>
  </init-param>
  <init-param>
   <name>javax.portlet.faces.defaultViewId.edit</name>
   <value>/xhtml/edit.xhtml</value>
  </init-param>
  <init-param>
   <name>javax.portlet.faces.defaultViewId.help</name>
   <value>/xhtml/help.xhtml</value>
  </init-param>
  <supports>
   <mime-type>text/html</mime-type>
   <portlet-mode>view</portlet-mode>
   <portlet-mode>edit</portlet-mode>
   <portlet-mode>help</portlet-mode>
  </supports>
  ...
 </portlet>
</portlet-app>
```

4.3. JSF and PortletPreferences

JSF portlet developers often have the requirement to provide the end-user with the ability to personalize the portlet in some way. To meet this requirement, the Portlet 2.0 specification provides the ability to define portlet preferences for each portlet. Preference names and default values can be defined in the `WEB-INF/portlet.xml` descriptor. Portal end-users start out interacting with the portlet user interface in portlet `VIEW` mode but switch to portlet `EDIT` mode in order to select custom preference values.

Example 4.2. Portlet Preferences in WEB-INF/portlet.xml

```
<portlet-preferences>
 <preference>
  <name>datePattern</name>
  <value>MM/dd/yyyy</value>
 </preference>
</portlet-preferences>
```

Additionally, Portlet 2.0 provides the ability to specify support for `EDIT` mode in the `WEB-INF/portlet.xml` descriptor.

Example 4.3. Enabling Support for Portlet EDIT Mode in WEB-INF/portlet.xml

```
<supports>
 <mime-type>text/html</mime-type>
 <portlet-mode>view</portlet-mode>
 <portlet-mode>edit</portlet-mode>
</supports>
```

However, just because support portlet `EDIT` mode has been specified, it doesn't mean that the portlet container knows which JSF view should be rendered when the user enters portlet portlet `EDIT` mode. JSF portlet developers must specify the Facelet view that is to be displayed for each supported portlet mode.

Example 4.4. Specifying a Facelet View for EDIT mode with The PortletFaces Bridge

```
<init-param>
 <name>javax.portlet.faces.defaultViewId.edit</name>
 <value>/edit.xhtml</value>
</init-param>
```

Facelet views that are designed to be used in portlet `EDIT` mode are typically forms that contain JSF component tags that enable the portlet end-user to select custom preference values that override the default values specified in the `WEB-INF/portlet.xml` descriptor. JSR 329 bridge implementations are required to provide an EL resolver that introduces the `mutablePortletPreferencesValues` variable into the EL, which is a mutable `java.util.Map` that provides read/write access to each portlet preference. By utilizing the JSR 329 `mutablePortletPreferencesValues` variable within an EL `ValueExpression`, portlet developers can declaratively bind the Facelet

view to the portlet preference model data. In order to save the preferences, a backing bean must call the PortletPreferences.store() method.

Example 4.5. EDIT Mode Facelet XHTML

```
<!--
This is a file named edit.xhtml that can be used for portlet EDIT
mode. It utilizes the JSR 329 mutablePortletPreferencesValues EL
variable for gaining read/write access to
javax.portlet.PortletPreferences.
-->
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns:f="http://java.sun.com/jsf/core"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:ui="http://java.sun.com/jsf/facelets">
 <h:head />
 <h:body>
 <h:form>
  <h:messages globalOnly="true" />
  <table>
   <tr>
    <td><h:outputLabel for="datePattern" value="Date Format" /></td>
    <td><h:inputText id="datePattern"
 value="#{mutablePortletPreferencesValues['datePattern'].value}"
 /></td>
    <td><h:message for="datePattern" /></td>
   </tr>
  </table>
  <hr />
  <h:commandButton
 actionListener="#{portletPreferencesBackingBean.submit}"
 value="Submit" />
 </h:form>
 </h:body>
</f:view>
```

Example 4.6. EDIT Mode Backing Bean - Part I

```
/**
 * This is a JSF backing managed-bean that has an action-listener
 * for saving portlet preferences.
 */
@ManagedBean(name = "portletPreferencesBackingBean")
@RequestScoped
public class PortletPreferencesBackingBean {

 public void submit() {

  // The JSR 329 specification defines an EL variable named
  // mutablePortletPreferencesValues that is being used in
  // the portletPreferences.xhtml Facelet composition. This
  // object is of type Map<String, Preference> and is
  // designed to be a model managed-bean (in a sense) that
  // contain preference values. However the only way to
  // access this from a Java class is to evaluate an EL
  // expression (effectively self-injecting) the map into
  // this backing bean.
  FacesContext facesContext = FacesContext.getCurrentInstance();
  ExternalContext externalContext =
   facesContext.getExternalContext();
  String elExpression = "mutablePortletPreferencesValues";
  ELResolver elResolver =
   facesContext.getApplication().getELResolver();
  @SuppressWarnings("unchecked")
  Map<String, Preference> mutablePreferenceMap =
   (Map<String, Preference>) elResolver.getValue(
    facesContext.getELContext(), null, elExpression);

  // Get a list of portlet preference names.
  PortletRequest portletRequest =
   (PortletRequest) externalContext.getRequest();
  PortletPreferences portletPreferences =
   portletRequest.getPreferences();
  Enumeration<String> preferenceNames =
   portletPreferences.getNames();
```

Example 4.7. EDIT Mode Backing Bean - Part II

```
  try {

   // For each portlet preference name:
   while (preferenceNames.hasMoreElements()) {

    // Get the value specified by the user.
    String preferenceName = preferenceNames.nextElement();
    String preferenceValue =
     mutablePreferenceMap.get(preferenceName).getValue();

    // Prepare to save the value.
    if (!portletPreferences.isReadOnly(preferenceName)) {
     portletPreferences.setValue(
      preferenceName, preferenceValue);
    }
   }

   // Save the preference values.
   portletPreferences.store();

   // Switch the portlet mode back to VIEW.
   ActionResponse actionResponse =
    (ActionResponse) externalContext.getResponse();
   actionResponse.setPortletMode(PortletMode.VIEW);

   // Report a successful message back to the user as feedback.
   FacesMessageUtil.addGlobalSuccessInfoMessage(facesContext);
  }
  catch (Exception e) {
   FacesMessageUtil.addGlobalUnexpectedErrorMessage(facesContext);
  }
 }
}
```

4.4. JSF ExternalContext and the Portlet API

Just as JSF web application developers rely on ExternalContext in order to get access to the Servlet API, JSF portlet developers also rely on ExternalContext in order to get access to the Portlet API.

4.4.1. Getting the PortletRequest and PortletResponse Objects

The two most common tasks that JSF portlet developers need to perform is to obtain an instance of the javax.portlet.PortletRequest or javax.portlet.PortletResponse objects.

Example 4.8. Getting the PortletRequest and PortletResponse objects from within a JSF backing managed-bean action method

```
public class BackingBean {

 public void submit() {
  FacesContext facesContext =
   FacesContext.getCurrentInstance();

  ExternalContext externalContext =
   facesContext.getExternalContext();

  PortletRequest portletRequest =
   (PortletRequest) externalContext.getRequest();

  PortletResponse portletResponse =
   (PortletResponse) externalContext.getResponse();
 }
}
```

LiferayFaces Project

For Liferay portlet developers, the LiferayFaces project features the LiferayFacesContext singleton which contains convenience methods like liferayFacesContext.getPortletRequest() and liferayFacesContext.getPortletResponse().

4.5. JSF and Inter-Portlet Communication

The PortletFaces Bridge supports Portlet 2.0 IPC using the JSR 329 approach for supporting Portlet 2.0 Events and also Portlet 2.0 Public Render Parameters. It could be argued that the Portlet 2.0 approaches for IPC have a common drawback in that they can lead to a potentially disruptive end-user experience. This is because they cause either an HTTP GET or HTTP POST which results in a full page refresh. Technologies such as ICEfaces Ajax Push can be used to solve this problem. Refer to the Ajax Push IPC section of this document for more details.

Example Portlets at portletfaces.org

Visit http://www.portletfaces.org/portletfaces-bridge/examples for example portlets that demonstrate how to use each of these various approaches to IPC.

4.5.1. Portlet 2.0 Public Render Parameters

As discussed in Chapter 1, the Public Render Parameters technique provides a way for portlets to share data by setting public/shared parameter names in a URL controlled by the portal. While the benefit of this approach is that it is relatively easy to implement, the drawback is that only small amounts of data can be shared. Typically the kind of data that is shared is simply the value of a database primary key. As required by the Portlet 2.0 standard, Public Render Parameters must be declared in the WEB-INF/portlet.xml descriptor.

Example 4.9. Specifying Supported Public Render Parameters in the WEB-INF/portlet.xml descriptor

```
<portlet>
 <portlet-name>customersPortlet</portlet-name>
 ...
 <supported-public-render-parameter>selectedCustomerId</supported-
public-render-parameter>
</portlet>
<portlet>
 <portlet-name>bookingsPortlet</portlet-name>
 ...
 <supported-public-render-parameter>selectedCustomerId</supported-
public-render-parameter>
</portlet>
<public-render-parameter>
 <identifier>selectedCustomerId</identifier>
 <qname
 xmlns:x="http://portletfaces.org/pub-render-
params">x:selectedCustomerId</qname>
</public-render-parameter>
```

The JSR 329 standard defines a mechanism by which developers can use Portlet 2.0 Public Render Parameters for IPC in a way that is more natural to JSF development. Section 5.3.2 requires the bridge to inject the public render parameters into the Model concern of the MVC design pattern (as in JSF model managed-beans) after RESTORE_VIEW phase completes. This is accomplished by evaluating the EL expressions found in the <model-el>...</model-el> section of the WEB-INF/faces-config.xml descriptor.

Example 4.10. Specifying Public Render Parameters in the WEB-INF/faces-config.xml descriptor

```
<faces-config>
 <application>
  <application-extension>
   <bridge:public-parameter-mappings>
    <bridge:public-parameter-mapping>
     <parameter>selectedCustomerId</parameter>

 <model-
el>#{customersPortlet:customersModelBean.selectedCustomerId}</model-
el>

 <model-el>#{bookingsPortlet:bookingsModelBean.selectedCustomerId}</
model-el>
    </bridge:public-parameter-mapping>
   </bridge:public-parameter-mappings>
  </application-extension>
 </application>
</faces-config>
```

Section 5.3.2 of the JSR 329 standard also requires that if a bridgePublicRenderParameterHandler has been registered in the WEB-INF/portlet.xml descriptor, then the handler must be invoked so that it can perform any processing that might be necessary.

Example 4.11. Specifying a bridgePublicRenderParameterHandler in the WEB-INF/portlet.xml descriptor

```
<!-- Optional bridgePublicRenderParameterHandler -->
<init-param>
 <name>javax.portlet.faces.bridgePublicRenderParameterHandler</name>
 <value>org.portletfaces.example.handler.CustomerSelectedHandler</
value>
</init-param>
```

Example 4.12. bridgePublicRenderParameterHandler Java Code

```
package org.portletfaces.example.handler;

import javax.faces.context.FacesContext;

import org.portletfaces.bridge.BridgePublicRenderParameterHandler;

public class CustomerSelectedHandler implements
 BridgePublicRenderParameterHandler {

 public void processUpdates(FacesContext facesContext) {
  // Here is where you would perform any necessary processing of
 public render parameters
 }

}
```

4.5.2. Portlet 2.0 Events

As discussed in Chapter 1, the Server-Side Events technique provides a way for portlets to share data using an event-listener design. When using this form of IPC, the portlet container acts as broker and distributes events and payload (data) to portlets. One requirement of this approach is that the payload must implement the java.io.Serializable interface since it might be sent to a portlet in another WAR running in a different classloader.
As required by the Portlet 2.0 standard, Events must be declared in the WEB-INF/portlet.xml descriptor.

Example 4.13. Specifying Supported Events in the WEB-INF/portlet.xml descriptor

```
<portlet>
 <portlet-name>customersPortlet</portlet-name>
 ...
 <supported-processing-event>
  <qname
 xmlns:x="http://portletfaces.org/events">x:ipc.customerEdited</
qname>
 </supported-processing-event>
 <supported-publishing-event>
  <qname
 xmlns:x="http://portletfaces.org/events">x:ipc.customerSelected</
qname>
 </supported-publishing-event>
</portlet>
<portlet>
 <portlet-name>bookingsPortlet</portlet-name>
 ...
 <supported-processing-event>
  <qname
 xmlns:x="http://portletfaces.org/events">x:ipc.customerSelected</
qname>
 </supported-processing-event>
 <supported-publishing-event>
  <qname
 xmlns:x="http://portletfaces.org/events">x:ipc.customerEdited</
qname>
 </supported-publishing-event>
</portlet>
<event-definition>
 <qname
 xmlns:x="http://portletfaces.org/events">x:ipc.customerEdited</
qname>
 <value-type>org.portletfaces.example.dto.Customer</value-type>
</event-definition>
<event-definition>
 <qname
 xmlns:x="http://portletfaces.org/events">x:ipc.customerSelected</
qname>
 <value-type>org.portletfaces.example.dto.Customer</value-type>
</event-definition>
```

Section 5.2.5 of the JSR 329 standard requires that if a BridgeEventHandler has been registered in the WEB-INF/portlet.xml descriptor, then the handler must be invoked so that it can perform any processing that might be necessary.

Example 4.14. Specifying a BridgeEventHandler in the WEB-INF/portlet.xml descriptor

```
<!-- Optional bridgeEventHandler -->
<init-param>
 <name>javax.portlet.faces.bridgeEventHandler</name>
 <value>org.portletfaces.example.event.CustomerEditedEventHandler</
value>
</init-param>
```

Example 4.15. BridgeEventHandler Java Code

```
package org.portletfaces.example.event;

import javax.el.ELContext;
import javax.el.ValueExpression;
import javax.faces.context.FacesContext;
import javax.portlet.Event;

import org.portletfaces.bridge.BridgeEventHandler;
import org.portletfaces.bridge.event.EventNavigationResult;

public class CustomerEditedEventHandler implements
 BridgeEventHandler {

 public EventNavigationResult handleEvent(FacesContext facesContext,
 Event event) {
  EventNavigationResult eventNavigationResult = null;
  String eventQName = event.getQName().toString();

  if
 (eventQName.equals("{http://portletfaces.org/
events}ipc.customerEdited")) {
   Customer customer = (Customer) event.getValue();
   getCustomerService(facesContext).save(customer);
   System.err.println("Received event ipc.customerEdited");
   System.err.println("customerId=" + customer.getCustomerId());
  }

  return eventNavigationResult;
 }
}
```

4.5.3. Portlet 2.0 Shared Session Scope

Perhaps the most natural approach for a JSF developer to try for IPC is to specify session scope on a JSF managed-bean. Surprisingly, this approach doesn't work. To understand the reason why, it is necessary to discuss the fact that the Portlet 1.0 and 2.0 standards make a distinction between two kinds of session scopes: javax.portlet.PortletSession.APPLICATION_SCOPE

and javax.portlet.PortletSession.PORTLET_SCOPE. The former can be used for sharing data between portlets packaged in the same WAR, but the latter cannot. The reason why JSF session scope can't be used to share data between portlets is because all JSF portlet bridges use PortletSession.PORTLET_SCOPE.

In order to share data with PortletSession.APPLICATION_SCOPE, the JSF portlet developer can place a JSF model managed-bean in request scope and use the getter/setter as a layer of abstraction.

Example 4.16. Shared Scope Model Managed-Bean - Part I

```
/**
 * This class is a request-scoped JSF managed-bean that has
 * a getter and setter that serves as a layer of abstraction
 * over PortletSession.APPLICATION_SCOPE
 */
@RequestScoped(name="sharedScopeModelBean")
public class SharedScopeModelBean {

 public static final String
  SHARED_STRING_KEY = “sharedStringKey”;

 public String getSharedString() {
  return PortletSessionUtil.getSharedSessionAttribute(
   SHARED_STRING_KEY);
 }

 public void setSharedString(String value) {
  PortletSessionUtil.setSharedSessionAttribute(
   SHARED_STRING_KEY, value);
 }
}

public class PortletSessionUtil {

 public static Object getSharedSessionAttribute(
  String key) {

  FacesContext facesContext =
   FacesContext.getCurrentInstance();

  ExternalContext externalContext =
   facesContext.getExternalContext();

  PortletSession portletSession =
   (PortletSession) externalContext().getSession(false);

  return portletSession.getAttribute(
   key, PortletSession.APPLICATION_SCOPE);
 }
```

Example 4.17. Shared Scope Model Managed Bean - Part II

```
 public static void setSharedSessionAttribute(
  String key, Object value) {

  FacesContext facesContext =
   FacesContext.getCurrentInstance();

  ExternalContext externalContext =
   facesContext.getExternalContext();

  PortletSession portletSession =
   (PortletSession)externalContext().getSession(false);

  portletSession.setAttribute(
   key, value, PortletSession.APPLICATION_SCOPE);
 }
}
```

Chapter 5. JSF Component Tags

5.1. Overview

Although the JSR 329 standard does not define any JSF components that implementations are required to provide, the PortletFaces Bridge comes with a handful of components that are helpful during JSF portlet development.

5.2. Bridge UIComponent Tags

The PortletFaces Bridge provides the following bridge-specific UIComponent tags as part of its component suite.

Table 5.1. UIComponent Tags

Tag	Description
`bridge:inputFile`	Renders an HTML `<input type="file" />` tag which provides file upload capability.

5.2.1. The bridge:inputFile tag

The `bridge:inputFile` tag renders an HTML `<input type="file" />` tag which enables file upload capability.

Dependency on JARs from apache.org

Usage of this tag requires the Apache `commons-fileupload` and `commons-io` dependencies. See the Example JSF2 Portlet at portletfaces.org for more details.

Table 5.2. Attributes

Attribute	Type	Description	Required
id	String	The identifier of the component	false
binding	org.portletfaces.bridge.component.HtmlInputFile	An EL expression that represents a JavaBean proeprty for the corresponding HtmlInputFile component in the runtime component tree.	false
rendered	Boolean	Boolean flag indicating whether or not this component is to be rendered during the RENDER_RESPONSE phase of the JSF lifecycle. The default value is “true”.	false

Example 5.1. Example usage of bridge:inputFile tag

```
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
 xmlns:bridge="http://portletfaces.org/bridge"
 xmlns:f="http://java.sun.com/jsf/core"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:ui="http://java.sun.com/jsf/facelets">
 <h:head />
 <h:body">
  <h:form>
   <bridge:inputFile binding="#{backingBean.attachment1}" />
   <h:commandButton
 actionListener="#{backingBean.uploadAttachments}"
    value="Submit" />
  </h:form>
 </h:body>
</f:view>
```

Example 5.2. Backing Bean Java Code

```
@ManagedBean(name = "backingBean")
@ViewScoped
public class BackingBean implements Serializable {

 private transient HtmlInputFile attachment1;
 public void uploadAttachments(ActionEvent actionEvent) {

  UploadedFile uploadedFile1 = attachment1.getUploadedFile();
  System.err.println("Uploaded file:" + uploadedFile1.getName());
 }
}
```

5.3. Portlet 2.0 UIComponent Tags

The PortletFaces Bridge provides the following portlet UIComponent tags as part of its component suite.

JSF 2 Facelets

Although JSP tags are provided by the portlet container implementation, the PortletFaces Bridge provides these tags in order to support their usage within Facelets.

Table 5.3. UIComponent Tags

Tag	Description
`portlet:actionURL`	If the var attribute is present, introduces an EL variable that contains a javax.portlet.ActionURL adequate for postbacks. Otherwise, the URL is written to the response.
`portlet:namespace`	If the var attribute is present, introduces an EL variable that contains a the portlet namespace. Otherwise, the namespace is written to the response.
`portlet:param`	Provides the ability to add a request parameter name=value pair when nested inside portlet:actionURL, portletRenderURL, or portlet:resourceURL tags.
`portlet:renderURL`	If the var attribute is present, introduces an EL variable that contains a javax.portlet.PortletURL adequate for rendering. Otherwise, the URL is written to the response.
`portlet:resourceURL`	If the var attribute is present, introduces an EL variable that contains a javax.portlet.ResourceURL adequate for rendering. Otherwise, the URL is written to the response.

5.3.1. The portlet:actionURL tag

If the var attribute is present, the `portlet:actionURL` tag introduces an EL variable that contains a javax.portlet.ActionURL adequate for postbacks. Otherwise, the URL is written to the response.

Table 5.4. Attributes

Attribute	Type	Description	Required
id	String	The identifier of the component	false
rendered	Boolean	Boolean flag indicating whether or not this component is to be rendered during the RENDER_RESPONSE phase of the JSF lifecycle. The default value is "true".	false
var	String	Specifies the name of a variable that will be introduced into the EL.	false

Example 5.3. Example usage of portlet:actionURL tag

```
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
 xmlns:f="http://java.sun.com/jsf/core"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:portlet="http://java.sun.com/portlet_2_0"
 xmlns:ui="http://java.sun.com/jsf/facelets">
 <h:head />
 <h:body">
  <h:form>
   <portlet:actionURL var="myActionURL" >
    <portlet:param name="foo" value="1234" />
   </portlet:actionURL>
   <h:outputText var="actionURL=#{myActionURL}" />
  </h:form>
 </h:body>
</f:view>
```

5.3.2. The portlet:namespace tag

If the var attribute is present, the `portlet:namespace` tag introduces an EL variable that contains the portlet namespace. Otherwise, the namespace is written to the response.

Table 5.5. Attributes

Attribute	Type	Description	Required
id	String	The identifier of the component	false
rendered	Boolean	Boolean flag indicating whether or not this component is to be rendered during the RENDER_RESPONSE phase of the JSF lifecycle. The default value is “true”.	false
var	String	Specifies the name of a variable that will be introduced into the EL.	false

Example 5.4. Example usage of portlet:actionURL tag

```
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
 xmlns:f="http://java.sun.com/jsf/core"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:portlet="http://java.sun.com/portlet_2_0"
 xmlns:ui="http://java.sun.com/jsf/facelets">
 <h:head />
 <h:body">
  <h:form>
   <portlet:namespace var="mynamespace" />
   <h:outputText var="namespace=#{mynamespace}" />
  </h:form>
 </h:body>
</f:view>
```

5.3.3. The portlet:param tag

The `portlet:param` tag provides the ability to add a request parameter name=value pair when nested inside portlet:actionURL, portletRenderURL, or portlet:resourceURL tags.

Table 5.6. Attributes

Attribute	Type	Description	Required
id	String	The identifier of the component	false
name	String	The name of the parameter.	true
rendered	Boolean	Boolean flag indicating whether or not this component is to be rendered during the RENDER_RESPONSE phase of the JSF lifecycle. The default value is “true”.	false
value	String	The value of the parameter.	true

Example 5.5. Example usage of portlet:actionURL tag

```
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
 xmlns:f="http://java.sun.com/jsf/core"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:portlet="http://java.sun.com/portlet_2_0"
 xmlns:ui="http://java.sun.com/jsf/facelets">
 <h:head />
 <h:body">
  <h:form>
   <portlet:actionURL>
    <portlet:param name="foo" value="1234" />
   </portlet:actionURL>
  </h:form>
 </h:body>
</f:view>
```

5.3.4. The portlet:renderURL tag

If the var attribute is present, the `portlet:renderURL` tag introduces an EL variable that contains a javax.portlet.PortletURL adequate for rendering. Otherwise, the URL is written to the response.

Table 5.7. Attributes

Attribute	Type	Description	Required
id	String	The identifier of the component	false
rendered	Boolean	Boolean flag indicating whether or not this component is to be rendered during the RENDER_RESPONSE phase of the JSF lifecycle. The default value is “true”.	false
var	String	Specifies the name of a variable that will be introduced into the EL.	false

Example 5.6. Example usage of portlet:renderURL tag

```
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
 xmlns:f="http://java.sun.com/jsf/core"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:portlet="http://java.sun.com/portlet_2_0"
 xmlns:ui="http://java.sun.com/jsf/facelets">
 <h:head />
 <h:body">
  <h:form>
   <portlet:renderURL var="myRenderURL">
    <portlet:param name="foo" value="1234" />
   </portlet:renderURL>
   <h:outputText var="actionURL=#{myRenderURL}" />
  </h:form>
 </h:body>
</f:view>
```

5.3.5. The portlet:resourceURL tag

If the var attribute is present, the `portlet:resourceURL` tag introduces an EL variable that contains a javax.portlet.ActionURL adequate for obtaining resources. Otherwise, the URL is written to the response.

Table 5.8. Attributes

Attribute	Type	Description	Required
id	String	The identifier of the component	false
rendered	Boolean	Boolean flag indicating whether or not this component is to be rendered during the RENDER_RESPONSE phase of the JSF lifecycle. The default value is “true”.	false
var	String	Specifies the name of a variable that will be introduced into the EL.	false

Example 5.7. Example usage of portlet:resourceURL tag

```
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
 xmlns:f="http://java.sun.com/jsf/core"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:portlet="http://java.sun.com/portlet_2_0"
 xmlns:ui="http://java.sun.com/jsf/facelets">
 <h:head />
 <h:body">
  <h:form>
   <portlet:resourceURL var="myResourceURL">
    <portlet:param name="foo" value="1234" />
   </portlet:resourceURL>
   <h:outputText var="actionURL=#{myResourceURL}" />
  </h:form>
 </h:body>
</f:view>
```

Chapter 6. Liferay Portal

6.1. Overview

Liferay Portal is an open source portal server that implements the Portlet 2.0 standard. The project home page can be found at http://www.liferay.com.

While the PortletFaces Bridge is theoretically compatible with any portal that implements the Portlet 2.0 standard, it has been carefully tested for use with Liferay Portal 5.2 and Liferay Portal 6.0 and has several optimizations that provide increased performance within Liferay.

6.2. JavaScript Concerns

When any JSF 2 portlet dynamically is added to a portal page at runtime by the end-user, the JSF 2 standard jsf.js JavaScript code will not be executed unless there is a full page refresh.

As a workaround, Liferay Portal provides configuration parameters that allow the developer to specify that a full page refresh is required. Doing this ensures that JSF 2 is properly initialized. The required parameters, render-weight and ajaxable, are specified in the WEB-INF/liferay-portlet.xml configuration file

Example 6.1. Specifying that a full page refresh should take place after the JSF 2 portlet is first added to the portal page

```
<liferay-portlet-app>
 <portlet>
  <portlet-name>my_portlet</portlet-name>
  <instanceable>false</instanceable>
  <render-weight>1</render-weight>
  <ajaxable>false</ajaxable>
 </portlet>
</liferay-portlet-app>
```

Chapter 7. ICEfaces 2 Portlet Development

7.1. Overview

ICEfaces 2 is an open source extension to JSF that enables developers with Java EE application skills to build Ajax-powered Rich Internet Applications (RIA) without writing any JavaScript code. The product contains a robust suite of Ajax-enabled JSF UI components, and also supports a broad array of Java application servers, IDEs, third party components, and JavaScript effect libraries. The project home page can be found at http://www.icefaces.org.

ICEfaces 2 introduces automatic-Ajax into JSF 2 applications, which makes it a particularly good choice for developing RIA portlets. Consider a portal page that contains two portlets: Portlet A and Portlet B. When submitting a form in Portlet A, an HTTP POST takes place and the entire portal page is refreshed. This can result in a disruptive end-user experience if the user had entered data in Portlet B prior to submitting Portlet A. ICEfaces 2 allows you to combat this disruptive experience with RIA features in your portlets.

ICEfaces 2 can be used in a variety of portal products including Liferay Portal. Since July of 2007, Liferay, Inc. and ICEsoft Technologies, Inc. have had a technology partnership in place to support customers that want to develop and deploy ICEfaces portlets within Liferay Portal.

7.2. ICEfaces Direct-To-DOM RenderKit

Rather than writing markup directly to the response, ICEfaces 2 components render themselves into a server-side Document Object Model (DOM) via the Direct-to-DOM (D2D) RenderKit. When a JSF view is requested for the first time, the markup inside the server-side DOM is delivered to the browser as part of the response. As the user interacts with the UI of the portlet, ICEfaces transparently submits user actions via Ajax and executes the JSF lifecycle. When the RENDER_RESPONSE phase of the JSF lifecycle completes, ICEfaces will compare the previous server-side DOM with the latest server-side DOM and send the incremental page updates back to the browser via the ICEfaces Ajax Bridge.

This approach is sometimes referred to as “dom-diffing” and provides the following benefits for portlets:

- The end user is immediately presented with form validation failures for visited fields when pressing the tab key

- When a navigation-rule fires and a new JSF view is to be rendered, ICEfaces will render the new JSF view by performing a complete update of the markup contained in the affected portlet, rather than causing the entire browser page to reload
- ICEfaces portlets will not disturb other portlets on the same portal page

7.3. ICEfaces Ajax Push and Inter-Portlet Communication

While the Portlet 2.0 standard defines techniques for performing inter-portlet communication (IPC), they cause either an HTTP GET or HTTP POST which results in a full page reload and a disruptive end-user experience. ICEfaces provides a natural way for portlets to perform IPC via ICEfaces Ajax Push.

ICEfaces pioneered Ajax Push, which is sometimes referred to as Reverse Ajax or Comet. The technology provides the ability for server-initiated events to cause incremental page updates to be sent to the browser. With ICEfaces Ajax Push, developers can create collaborative and dynamic enterprise applications like never before.

Because the mechanism facilitates asynchronous updates from the server to the client, interaction with one ICEfaces portlet can trigger communication with other ICEfaces portlets by changing values in JSF backing and model managed-beans. This mechanism is not restricted to IPC among portlets on the same portal page in a single browser, but can include updating other browsers that are interacting with the same portal page. The result is not just inter-portlet communication, but inter-portlet, inter-browser communication. Additionally, ICEfaces Ajax Push solves the potentially disruptive end-user experience associated with the Portlet 2.0 standard IPC techniques.

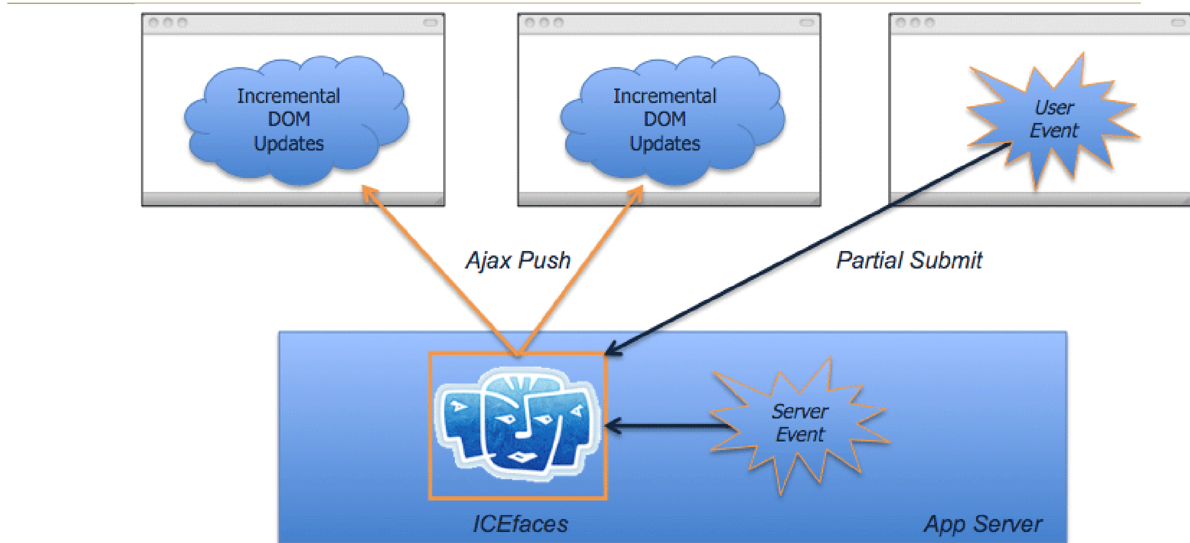


Figure 7.1. Illustration of IPC with ICEfaces Ajax Push

The following is a list of guidelines for achieving IPC with ICEfaces Ajax Push

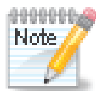
- Package the portlets that need to communicate in the same WAR.
- In order to share data between portlets, use application-scoped beans or request-scoped beans that store data in PortletSession.APPLICATION_SCOPE.
- Use the ICEfaces Ajax Push SessionRenderer to trigger client updates when the shared data changes.

Example 7.1. Chat Portlet Managed-Bean

```
/*
* This is a file named ChatRoomManagedBean.java
* that is registered as a JSF managed-bean in
* application scope. It maintains a chat log
* that participates in ICEfaces Ajax Push using
* the SessionRenderer.
*/
@ManagedBean(name = "chatRoomManagedBean")
@RequestScoped
public class ChatRoomManagedBean {

 private String messageText;

 private List<String> messages = new ArrayList<String>();

 private static final String
  AJAX_PUSH_GROUP_NAME = "chatRoom";

 public ChatRoomsModel() {
  SessionRenderer.addCurrentSession(
   AJAX_PUSH_GROUP_NAME);
 }

 @PreDestroy
 public void preDestroy() throws Exception {
  SessionRenderer.removeCurrentSession(
   AJAX_PUSH_GROUP_NAME);
 }

 public void addMessage(ActionEvent actionEvent) {
  messages.add(messageText);
  SessionRenderer.render(AJAX_PUSH_GROUP_NAME);
 }

 public List<String> getMessages() {
  return messages;
 }

 public String getMessageText() {
  return messageText;
 }

 public void setMessageText(String messageText) {
  this.messageText = messageText;
 }
}
```

Example 7.2. Chat Portlet Facelet View

```
<!-- This is a Facelet view named chatRoom.xhtml -->
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
 xmlns:f="http://java.sun.com/jsf/core"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:ui="http://java.sun.com/jsf/facelets">
 <h:head />
 <h:body styleClass="example-icefaces-portlet">
  <h:form>
   <h:dataTable
    value="#{chatRoomManagedBean.messages}"
    var="message">
    <h:column>
     <h:outputText value="#{message}" />
    </h:column>
   </h:dataTable>
   <h:inputText
    value="#{chatRoomManagedBean.messageText}" />
   <h:commandButton
    actionListener="#{chatRoomManagedBean.addMessage}" />
  </h:form>
 </h:body>
</f:view>
```



Example Portlets at portletfaces.org

The portletfaces.org website hosts a variety open source demonstration portlets that focus on ICEfaces portlets for Liferay Portal. Specifically, there is a portlet featuring an ICEfaces chat room that integrates with Liferay Portal's online users. When a user signs-in to Liferay Portal, a server-initiated event triggers ICEfaces Ajax Push so that other users that are online become aware of online presence. Additionally, when the user clicks on a “Chat” icon, ICEfaces Ajax Push is used for IPC to begin a new chat room in a Chat Portlet. For more information, visit : http://www.portletfaces.org/projects/portletfaces-bridge/examples and http://www.portletfaces.org/projects/liferayfaces/examples

7.4. ICEfaces Themes and Portal Themes

The ICEfaces Component Suite fully supports consistent component styling via a set of predefined CSS style classes and associated images. Changing the component styles for a web application developed with the ICEfaces Component Suite is as simple as changing the style sheet used. ICEfaces ships with a set of predefined style sheets are available to be used as-is, or

customized to meet the specific requirements of the application. There are five predefined ICEfaces style sheets included, two of which are designed to be used inside a portlet container:

- rime.css
- rime-portlet.css
- xp.css
- xp-portlet.css
- royale.css

Example 7.3. Specifying the portlet-compatible version of the ICEfaces "XP" theme

```
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
 xmlns:f="http://java.sun.com/jsf/core"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:ui="http://java.sun.com/jsf/facelets">
 <h:head>
 <link href="#{request.contextPath}/xmlhttp/css/xp/xp-portlet.css"
 rel="stylesheet"
  type="text/css" />
 </h:head>
 <h:body styleClass="example-icefaces-portlet">
  <h:form>
   ...
  </h:form>
 </h:body>
</f:view>
```

The Portlet 1.0 and 2.0 standards document a set of common CSS class names that should be applied to specific page elements in order to integrate with the portlet container's theme mechanism. When running in a portlet container, ICEfaces 1.8 compatibility components will automatically render the following subset of Portlet 1.0 CSS class names where appropriate

- portlet-form-button
- portlet-form-field
- portlet-form-input-field
- portlet-form-label

- portlet-menu
- portlet-menu-cascade-item
- portlet-menu-item
- portlet-menu-item-hover
- portlet-section-alternate
- portlet-section-body
- portlet-section-footer
- portlet-section-header
- portlet-msg-alert
- portlet-msg-error
- portlet-msg-info

To disable this feature, developers can specify the com.icesoft.faces.portlet.renderStyles context parameter in the WEB-INF/web.xml configuration file and set its value to false.

Example 7.4. Disabling automatic rendering of Portlet 1.0 / 2.0 standard CSS class names in the WEB-INF/web.xml configuration file

```
<context-param>
 <param-name>
  com.icesoft.faces.portlet.renderStyles
 </param-name>
 <param-value>false</param-value>
</context-param>
```

7.5. ICEfaces Themes and Liferay Themes

Liferay Portal supports styling for Portlet 1.0 and 2.0 standard CSS class names as well as a set of vendor-specific CSS class names within the context of a Liferay theme. However, since Liferay themes do not contain styling for the ICEfaces Component Suite, it is necessary to select an ICEfaces style sheet that is visually compatible with the Liferay theme.

On some occasions, it becomes necessary to override some of the styling in a Liferay theme in order to make it more visually compatible with an

ICEfaces portlet. For example, Liferay themes typically render spans of class portlet-msg-error with a margin that has too much space to be placed alongside a rendered ice:inputText component tag.

Example 7.5. Overriding styling in a Liferay theme from within a Facelet view so that rendered output from ice:messages and ice:message have a more narrow margin

```
<?xml version="1.0" encoding="UTF-8"?>
<f:view xmlns="http://www.w3.org/1999/xhtml"
 xmlns:f="http://java.sun.com/jsf/core"
 xmlns:h="http://java.sun.com/jsf/html"
 xmlns:ui="http://java.sun.com/jsf/facelets">
 <h:head>
 <link href="#{request.contextPath}/xmlhttp/css/xp/xp-portlet.css"
 rel="stylesheet"
  type="text/css" />
 <link href="#{request.contextPath}/css/liferay-theme-override.css"
 rel="stylesheet"
  type="text/css" />
 </h:head>
 <h:body styleClass="example-icefaces-portlet">
  <h:form>
   <h:messages />
   ...
  </h:form>
 </h:body>
</f:view>

/*
 * This is a separate file named liferay-theme-override.css
 */
.example-icefaces-portlet .portlet-msg-error {
 margin: 1px 0px 0px 0px;
 padding: 1px 5px 1px 24px;
}
```