**TOPIC 1 :- Setting up the development environment**

Know the capabilities of two development environments for developing portlet applications for WebSphere Portal, IBM® Rational® Application Developer and IBM WebSphere® Portlet Factory.

Rational Application Developer includes tools designed to help you develop portlet applications for WebSphere Portal. The portlet tools provide the following capabilities:

* Portlet project support for the standard portlet API and the IBM portlet API.
* Web perspective views and editors for developing portlets.
* Portlet project wizard to create basic portlets, Faces portlets, and Struts portlets.
* Editing and validation of the portlet deployment descriptor (portlet.xml).
* Testing and debugging of portlets within the workbench using the WebSphere Portal Test Environment.
* Testing and debugging of portlets on a remote machine using the WebSphere Portal Server Attach.
* Visual tooling to insert portlet programming objects into JSP files, using Page Designer.
* Portlet sample applications, available in the Samples Gallery.
* Educational tutorials, available in the Tutorials Gallery.

Before using Rational Application Developer, determine which configuration best suits your requirements. See the Rational Application Developer documentation for complete installation and configuration instructions on installing WebSphere Portal for use with Rational Application Developer.

WebSphere Portlet Factory: Its features include:

* Tight Integration with WebSphere Portal (auto deploy portlets, Click-2-Action, People Awareness, SSO).
* Plug-in to Eclipse and IBM Rational Application Developer.
* Multi-page, complex portlets without coding.
* SOA delivery/robust integration capabilities with existing applications.
* Unlimited customization without the maintenance overhead.
* Business user configuration.
* Rapid iteration and change.
* Flexible deployment options.

**TOPIC 2 :- Creating a simple portlet**

View the basic steps for creating a simple portlet, that include writing the portlet code, compiling java source, creating the JAR file, writing the portlet descriptors, setting up the WAR file directory structure, and packaging and deploying portlets.

Before you begin developing portlets, you should setup an environment that makes the tasks of writing, compiling, and testing portlets easier. Rational Application Developer includes a test environment that you can use to run and debug your portlets without having to manually deploy them to the server. You can setup the run time environment for debugging portlets on the local development machine or on a remote server. Refer to the documentation for Rational Application Developer for complete setup instructions.

Rational Application Developer provides wizards to help you build, test, and deploy portlets using all of the APIs and related classes and interfaces available in the portlet run time environment. You can also build portlets using your own development environment and tools. If you are not using wizards to develop portlets, the following topics describe the mechanics of building a simple portlet.

* [Writing the portlet code](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbscfg.html#wpsbscfg__hello_world)
* [Compiling Java source](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbscfg.html#wpsbscfg__compile)
* [Creating the JAR file](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbscfg.html#wpsbscfg__jarfile)
* [Writing the portlet descriptors](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbscfg.html#wpsbscfg__samp_pdd)
* [Setting up the WAR file directory structure](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbscfg.html#wpsbscfg__par)
* [Packaging and deploying portlets](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbscfg.html#wpsbscfg__wpspar)
* [IBM portlet API examples for Hello World](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbscfg.html#wpsbscfg__ibmhwex)

**Writing the portlet code**

The Hello World portlet provides an introduction to writing your first portlet. The portlet is provided along with the source in the IBM Portlet Samples package, which is available from the portlet catalog by searching for navcode 1WP10017Z. See [Sample portlets](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/jsrsamp.html) for more information. Hello World provides the fewest methods required for a portlet. It uses the portlet response object to write simple output directly to the portal page.

*Figure 1. Example: Java source for Hello World portlet (standard portlet API)*

package com.ibm.wps.samples.jsr;

import javax.portlet.\*;

import java.io.\*;

public class HelloWorld extends GenericPortlet {

public void init (PortletConfig portletConfig) throws UnavailableException, PortletException

{

super.init(portletConfig);

}

public void doView(RenderRequest request, RenderResponse response)

throws PortletException, IOException

{

// set return content type

response.setContentType("text/html");

PrintWriter writer = response.getWriter();

writer.println("<p class='wpsPortletText'>Hello Portal World!</p>");

}

}

If you are familiar with writing to the IBM portlet API, here are some of the key differences between this standard API example and the corresponding [Figure 6](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbscfg.html#wpsbscfg__ibmhwportlet).

| **API element** | **IBM portlet API** | **Java Portlet Specification** |
| --- | --- | --- |
| import statements | org.apache.jetspeed.portlet | javax.portlet |
| Portlet class | *PortletAdapter* | *GenericPortlet* which also can throw a *PortletException* |
| Request object | *PortletRequest* | *RenderRequest* |
| Response object | *PortletResponse* | *RenderResponse* |

Also notice that the content type must be set in the response.

**Compiling Java source**

Compile your Java source files. To do this, proceed as follows: Before you compile your Java source, set the CLASSPATH for the compiler to find the JAR files for any portlet packages that your portlet uses by running the following command:

* **UNIX:** ./setupcmdLine.sh
* **i5/OS:** setupcmdLine.sh
* **Windows:** setupcmdLine.bat

The following JAR files should be set in the CLASSPATH to compile portlets:

**Standard portlets**

| **Jar file** | **Purpose** |
| --- | --- |
| portletapi\_20.jar | This file complies with the Java Portlet Specification Version 2.0. |
| public\_api.jar | Use this file if you use services from the Public **API** javadoc package. |
| public\_api.jar + public\_spi.jar | Use this file if you use services from the Public **SPI** javadoc package. |

These files are located in the directory [*PortalServer\_root*](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/reference/wpsdirstr.html#wpsdirstr__wp_root)/doc/compile. The JavaDocs are available via [developerWorks: http://www.ibm.com/developerworks/websphere/library/specs/0608\_wp6javadoc.html](http://www.ibm.com/developerworks/websphere/library/specs/0608_wp6javadoc.html).

**IBM portlets**

| **Jar file** | **Purpose** |
| --- | --- |
| wp.pe.api.legacy.jar | IBM portlet API |
| wp.portletservices.api.legacy.jar | Portlet services |
| wp.pe.rt.api.jar | Portlet menus |

Then, compile the portlet using the fully qualified path to the Java portlet source.

appserver\java\bin\javac -classpath %WAS\_CLASSPATH%;path\_to\portletapi\_20.jar

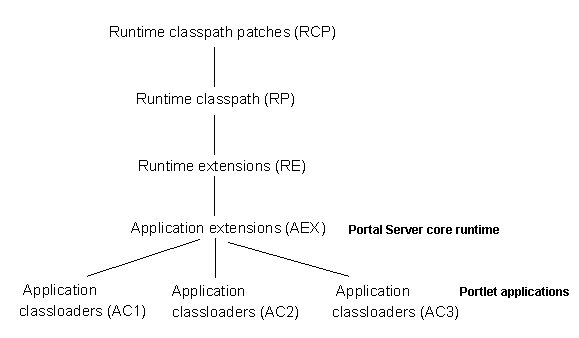
com.ibm.wps.samples.jsr.HelloWorld.java

appserver\java\bin\javac -classpath %WAS\_CLASSPATH%;path\_to\portletapi\_20.jar

com.ibm.wps.samples.v4.HelloWorld.java

**Loading classes for portlets**

WebSphere Portal classloading follows the WebSphere Application Server hierarchy for classpaths and search orders. A particular classloader can reference other classes as long as the other classes can be loaded by the same classloader or any of its ancestors, but not its children. The graphic illustrates where WebSphere Portal and portlet applications fit into the classloading hierarchy.



As illustrated, WebSphere Portal is an application extension (AEX) under WebSphere Application Server. Consequently, the WebSphere Portal core classes are in the classpath [*PortalServer\_root*](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/reference/wpsdirstr.html#wpsdirstr__wp_root)/shared/app. If an installed portlet application includes a classloader, the portlet application classloader is an application classloader (ACx) under WebSphere Portal.

If you suspect a classloading problem, ensure that the required classes are in the appropriate classpath according to the classloading hierarchy.

**Creating the JAR file**

Next, the portlet must be packaged in the JAR file format. To create a JAR file with the name HelloWorld.jar, enter the following command:

jar -cf HelloWorld.jar HelloWorld.class

Refer to the JDK documentation for more information about the JAR command.

**Writing the portlet descriptors**

The following samples can be packaged with the Hello World portlet.

**Web application deployment descriptor for standard portlets:**

According to the Java Portlet Specification, only Web resources that are not portlets should be declared in the web.xml. However, the following properties should be set to correspond to the portlet descriptor:

* **<description/>**

describes the portlet application.

* **<display-name/>**

indicates the portlet application name.

* **<security-role/>**

indicates the portlet application security role mapping. Omit this tag if the portlet does not use this feature.

*Figure 2. Example: Web application deployment descriptor for Hello World (standard portlets)*

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"

"http://java.sun.com/dtd/web-app\_2\_3.dtd">

<web-app id="WebApp">

<display-name>Hello World (JSR)</display-name>

<description>Basic JSR 168 portlet</description>

</web-app>

If you are familiar with the web.xml for IBM portlets, the key difference between this example and the corresponding [Figure 7](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbscfg.html#wpsbscfg__ibmhwweb) is the required <servlet/> element in the web.xml for IBM portlets.

**Standard portlet deployment descriptor:**

The following shows the minimum elements required for the standard portlet deployment descriptor.

*Figure 3. Example: Portlet deployment descriptor for Hello World (standard)*

<?xml version="1.0" encoding="UTF-8"?>

<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app\_2\_0.xsd" version="2.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app\_2\_0.xsd

http://java.sun.com/xml/ns/portlet/portlet-app\_2\_0.xsd">

<portlet>

<portlet-name>HelloWorld portlet name</portlet-name>

<display-name>Hello World portlet (JSR)</display-name>

<display-name xml:lang="en">Hello World portlet (JSR)</display-name>

<portlet-class>com.ibm.wps.samples.jsr.jsrHelloWorld</portlet-class>

<supports>

<mime-type>text/html</mime-type>

<portlet-mode>view</portlet-mode>

</supports>

<supported-locale>en</supported-locale>

<portlet-info>

<title>Hello World (JSR)</title>

</portlet-info>

</portlet>

</portlet-app>

There are number of differences between the elements in this example and the corresponding [Figure 8](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbscfg.html#wpsbscfg__ibmhwpdd). In addition, the standard portlet descriptor is defined by an XML schema and does not require a DTD.

**Setting up the WAR file directory structure**

Before you package your portlet, the class files and resources must be arranged in the WAR file directory structure described here. A portlet application exists as a structured hierarchy of directories.

**Note:** Do not package .tld files for tag libraries that are provided by the portal or application server installation with the portlet application WAR file. This includes the IBM portlet API tags, JSR 168 and 286 portlet tags and the Java Standard Tag Library (JSTL).

**/**

The root directory of the portlet file structure.

**/images**

Location for any images the required by the portlet.

**/WEB-INF**

Location for all protected resources. The /WEB-INF directory stores the portlet descriptor document and all of the run time executable JAR files and classes that the packaged portlet requires.

The portlet information directory is not part of the public document tree of the application. Files that reside in /WEB-INF are not served directly to a client.

**/WEB-INF/lib**

Location for storing portlet JAR files.

**/WEB-INF/jsp**

Location for JSP files. This is a suggested path name. Your JSPs can be packaged in any location. JSPs that are included inside the portlet markup should be placed under the /WEB-INF directory. You should only place them outside the /WEB-INF directory if you create direct links to them.

**/WEB-INF/classes**

Location for portlet class files. Individual class files should be stored in a directory structure within /WEB-INF/classes that reflects the class package. For example, the class HelloWorld.class, in package com.ibm.wps.samples, would be stored in /WEB-INF/classes/com/ibm/wps/samples/HelloWorld.class.

**/META-INF**

Location for the manifest file, manifest.mf and the Java 2 security file, was.policy (if present). The manifest is in the standard JAR file format as defined by the Java 1.3 specification. The Java 2 security policy file is used to allow a portlet to perform operations that might be restricted if Java 2 security is enabled. The contents of the /META-INF directory is not served to clients.

**Note:** The application server searches for security policy files in the location of the enterprise application archive rather than the Web application archive. Therefore, the portal server copies was.policy from the *appname*.war/META-INF directory to the generated *appname.ear*/META-INF directory during deployment of a portlet WAR file.

**Packaging and deploying portlets**

To deploy a portlet and run it on the server, it must be packaged in the form of a *Web application ARchive* or *WAR* file. The WAR file format contains the Java classes and resources that make up one or more portlets in a portlet application. The resources can be images, JSP files, [Writing the portlet descriptors](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbscfg.html#wpsbscfg__samp_pdd), and property files containing translated message text. Packaging portlet classes, resources, and descriptive information in a single file makes distribution and deployment of portlets easier.

WebSphere Portal includes an administrative portlet for installing, uninstalling, and updating portlets. Portlets contained in WAR files have the advantage of being dynamically downloaded and installed. The portal administrator can download a WAR file from the Internet and then use the portal administration interface to install the portlet to WebSphere Portal. After installation, the portlet is ready for use and does not require the server to be restarted. To package your portlet in a WAR file, you can use the JAR utility to [package the portlet into a WAR file](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbscfg.html#wpsbscfg__package).

**Note:** Because Windows limits the maximum path length to 260 characters, the name of the WAR file must be less than 25 characters. On a portal server running on Windows, installing a WAR file with a name that is more than 25 characters will result in an error.

**Packaging a portlet and resources into a WAR file**

Any JAR utility may be used to build a WAR file. Below are examples of how to use the JAR utility provided by WebSphere Application Server.

* To create a WAR file with the name HelloWorld.war and include all of the files in the /WEB-INF and /images directories:
* jar -cf HelloWorld.war images WEB-INF
* To update an existing WAR file, HelloWorld.war with a revised portlet descriptor:
* jar -uf HelloWorld.war WEB-INF/portlet.xml
* To extract the portlet descriptor from the WAR file, HelloWorld.war :
* jar -xf HelloWorld.war WEB-INF/portlet.xml
* To extract all files from an existing WAR file, HelloWorld.war:
* jar -xf HelloWorld.war

After the WAR file is created, it can be installed to WebSphere Portal as described in [Portal administration portlets](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/admin/adpltadm.html).

**Preparing the portlet application for installation**

To facilitate deployment of portlet applications and complex portlets, you can provide a portlet configuration file that can be invoked by the XML configuration interface (XMLAccess). The XML configuration interface allows the portlet developer to specify places, pages, themes, skins, supported markups and clients, and other settings for a portlet application. This is especially useful for portlets that use messaging because these portlets have to be placed on the same page. For more information, see [The XML configuration interface](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/admin/admxmlai.html). There is also some helpful information about XMLAccess in the [IBM WebSphere Portal Zone](http://www7b.software.ibm.com/wsdd/zones/portal/).

When constructing XMLAccess scripts for use in installing standard portlets, use the following values:

* *uid* attribute for the <web-app> element:

Use the *uid* attribute of the <portlet-app/> subelement with a .webmod suffix. As described subsequently in this topic, the *uid* attribute of the <portlet-app/> subelement is dependent on the presence of the *id* attribute of the <portlet-app/> element from the portlet.xml.

* uid attribute for the <portlet-app> element:

Use the id attribute of the <portlet-app/> element from the portlet.xml. If this value has not been specified, specify the WAR file name of the portlet application in its place. For portlet updates, the WAR file name must be the original name of the WAR file used to install the portlet application. That is, the WAR file name can be changed, but the uid must indicate the original uid used during portlet installation.

* *name* attribute for the <portlet> element:

Use the content of the <portlet-name/> element from the portlet.xml.

* *referenceid* attribute of the <servlet> element:

Use the content of the <portlet-name/> element from the portlet.xml appended with the .servlet suffix.

For example, a portlet application might use a portlet descriptor as follows:

*Figure 4. Example <portlet-app/> element in portlet descriptor*

<portlet-app

xmlns="http://java.sun.com/xml/ns/portlet/portlet-app\_1\_0.xsd"

version="1.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation=

"http://java.sun.com/xml/ns/portlet/portlet-app\_1\_0.xsd

http://java.sun.com/xml/ns/portlet/portlet-app\_1\_0.xsd">

<portlet>

<portlet-name>External Bookmarks</portlet-name>

...

In this example, there is no id attribute provided on the <portlet-app/> element. Therefore, the <portlet-app/> element of the XMLAccess script would use the WAR file name, as follows:

*Figure 5. Example <web-app/> element in XMLAccess*

<web-app action="update" active="true" uid="bookmarks.war.webmod">

<url>file:///$server\_root$/installableApps/bookmarks.war</url>

<servlet action="update"

active="true"

referenceid="External Bookmarks.servlet"/>

<portlet-app action="update" active="true" uid="bookmarks.war">

<portlet action="update" active="true" name="External Bookmarks">

</portlet-app>

</web-app>

**IBM portlet API examples for Hello World**

*Figure 6. Example: Java source for Hello World portlet (IBM)*

package com.ibm.wps.samples.v4;

import org.apache.jetspeed.portlet.\*;

import java.io.\*;

public class HelloWorld extends PortletAdapter {

public void init (PortletConfig portletConfig) throws UnavailableException

{

super.init(portletConfig);

}

public void doView(PortletRequest request, PortletResponse response)

throws PortletException, IOException

{

PrintWriter writer = response.getWriter();

writer.println("<p class='wpsPortletText'>Hello Portal World!</p>");

}

}

*Figure 7. Example: Web application deployment descriptor (IBM)*

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"

"http://java.sun.com/dtd/web-app\_2\_3.dtd">

<web-app id="WebApp">

<display-name>HelloWorld</display-name>

<servlet id="com.ibm.wps.samples.HelloWorld.a0ae41f2d3c1001710b7b313e1a97">

<servlet-name>HelloWorld</servlet-name>

<servlet-class>com.ibm.wps.samples.v4.HelloWorld</servlet-class>

</servlet>

<servlet-mapping

id="ServletMapping\_com.ibm.wps.samples.HelloWorld.a0ae41f2d3c1001710b7b313e1a97">

<servlet-name>HelloWorld</servlet-name>

<url-pattern>/HelloWorld/\*</url-pattern>

</servlet-mapping>

</web-app>

As described in [Deployment descriptors](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsptdesc.html), the *href* attribute of the <portlet/> element references the servlet ID from Web deployment descriptor.

**Note:** The portlet deployment descriptor references the portlet\_1.1.dtd, which portal server finds in the [*was\_profile\_root*](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/reference/wpsdirstr.html#wpsdirstr__was_profile_root)/installedApps/*hostname*/wps.ear/wps.war/dtd directory. Do not package the DTD with the portlet application WAR file.

*Figure 8. Example: Portlet deployment descriptor (IBM)*

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE portlet-app-def PUBLIC "-//IBM//DTD Portlet Application 1.1//EN"

"portlet\_1.1.dtd">

<portlet-app-def>

<portlet-app uid="com.ibm.wps.samples.HelloWorld.a0ae41f2d3c1001710b7b313e1a97"

major-version="1" minor-version="0">

<portlet-app-name>HelloWorld application</portlet-app-name>

<portlet id="com.ibm.wps.samples.HelloWorld"

href="WEB-INF/web.xml#com.ibm.wps.samples.HelloWorld.a0ae41f2d3c1001710b7b313e1a97"

major-version="1" minor-version="0">

<portlet-name>HelloWorld portlet</portlet-name>

<cache>

<expires>0</expires>

<shared>NO</shared>

</cache>

<allows>

<maximized/>

<minimized/>

</allows>

<supports>

<markup name="html">

<view/>

</markup>

</supports>

</portlet>

</portlet-app>

<concrete-portlet-app uid="com.ibm.wps.samples.HelloWorld.313e1a97f47.2">

<portlet-app-name>HelloWorld application</portlet-app-name>

<concrete-portlet href="#com.ibm.wps.samples.HelloWorld">

<portlet-name>HelloWorld portlet</portlet-name>

<default-locale>en</default-locale>

<language locale="en">

<title>HelloWorld portlet</title>

<title-short></title-short>

<description></description>

<keywords></keywords>

</language>

</concrete-portlet>

</concrete-portlet-app>

</portlet-app-def>

# TOPIC 3 :- Generating output

View how portlets use JSPs to generate markup, create URLs to portlet resources, support multiple devices, markups, and languages in IBM portlets, and make use of JSTL.

In the previous example, the [Hello World](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbscfg.html) portlet provided markup by using a Java PrintWriter. However, most portlets generate output using JSPs. One exception to this is when the portlet has to transform XML source. In this case, the portlet can use XSLT to generate markup. The following sections describe how JSPs are used by portlets.

* [Using JSPs to generate markup](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsoutput.html#wpsbsoutput__helloWorld_jsps)
* [Creating URLs to portlet resources](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsoutput.html#wpsbsoutput__encodeURL)
* [Supporting multiple devices, markups, and languages in IBM portlets](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsoutput.html#wpsbsoutput__md_capability)
* [Using JSTL in portlet JSPs](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsoutput.html#wpsbsoutput__portlet_jstl)
* [IBM portlet API examples for Hello JSP](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsoutput.html#wpsbsoutput__ibmhjex)

**Using JSPs to generate markup**

To separate portlet output from the main functionality of the portlet, use a JSP. Below is the JSP for the view page of the jsrHelloJSP.war sample. All sample portlets are available from the portlet catalog by searching for navcode 1WP10017Z. See [Sample portlets](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/jsrsamp.html) for more information.

*Figure 1. Example: JSP for Hello JSP portlet (standard)*

<%@ taglib uri="http://java.sun.com/portlet" prefix="portletAPI" %>

<%@ page session="false"%>

<p class="portlet-font">Hello JSP!</p>

If you are familiar with writing to the IBM portlet API, here are some of the key differences between this standard portlet example and the corresponding [Figure 7](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsoutput.html#wpsbsoutput__ibmhjjsp).

* As with the [Hello World portlet](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbscfg.html#wpsbscfg__hwportlet), the content type must be set in the Java source using the setContentType() method.
* The taglib directive points to the URI for the JSP tag library for the Portlet API Specification. In this example, however, no JSP tags are used.
* Style classes from the WSRP specification are used. See the [Markup guidelines](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpspar.html) for more information.

Separate JSPs would exist to provide the user interface for supporting any additional portlet modes, such as edit or help. The basic portlet wizard in Rational Application Developer allows you to create a portlet that provides JSPs for some of the other modes in which the portlet can be invoked.

The following shows the doView() method provided in the jsrHelloJSP sample.

*Figure 2. Example: Java source for Hello JSP portlet (standard)*

package com.ibm.wps.samples.jsr;

import javax.portlet.\*;

import java.io.\*;

public class HelloJSP extends GenericPortlet {

public void init(PortletConfig portletConfig) throws UnavailableException, PortletException

{

super.init(portletConfig);

}

public void doView(RenderRequest request, RenderResponse response)

throws PortletException, IOException {

// set return content type

response.setContentType("text/html");

PortletContext context = getPortletConfig().getPortletContext();

context.getRequestDispatcher("/jsp/View.jsp").include( request, response);

}

}

If you are familiar with writing to the IBM portlet API, here are some of the key differences between this standard portlet example and the corresponding [Figure 8](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsoutput.html#wpsbsoutput__ibmhjjava) are as follows:

* IBM portlet JSPs are included by using PortletContext.include() method. Standard portlet JSPs are included by a request dispatcher's include() method.
* The MIME type of the output returned in the response must be set before including the JSP. IBM portlets declare the MIME type in the page directive of the JSP.

There are several points to keep in mind when writing your JSPs:

1. For consistency in portal look and feel, use the portlet's class specifications in the WSRP specification.
2. Be sure to include the appropriate tag library to obtain the needed functionality in your JSPs.
3. Become familiar with the guidelines and best practices for portlet markup. For example, all named elements must be namespace-encoded, using the <portletAPI:encodeNamespace> tag, to avoid clashes with other named elements on the portal page.
4. Portlet JSPs cannot link directly to resources within the portlet's WAR directory structure. See [Creating URLs to portlet resources](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsoutput.html#wpsbsoutput__encodeURL) for examples.

See the [Markup guidelines](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpspar.html) for more information about good JSP coding practices.

**Creating URLs to portlet resources**

Portlet JSPs cannot link directly to content (for example, images, applets, other JSPs, or other resources) within the portlet's WAR directory structure. Instead, they have to use the services of the portlet container to create portlet URLs from which the content can be accessed. Use the encodeURL() method of the PortletResponse to access content within the portlet WAR structure. The following examples are used in the View World portlet samples.

**ibmViewWorld.war**

*Figure 3. IBM portlet JSP with image file*

<img src='<%=portletResponse.encodeURL("images/earth.jpg")%>'

alt="Earth" />

**jsrViewWorld.war**

For standard portlets, you also have to add the context path of the portlet from the request:

*Figure 4. Standard portlet JSP with image file*

<img src='<%=renderResponse.encodeURL(renderRequest.getContextPath() +

"/images/earth.jpg")%>'

alt="Earth" />

The String returned by the encodeURL() method returns the relative URL of the content, without the host name and domain information.

**Multimedia example**

The following example shows how an audio file can be included in a JSP for a standard portlet.

*Figure 5. Standard portlet JSP with multimedia file*

<object

classid='<%=renderResponse.encodeURL(renderRequest.getContextPath() +

"/audio/WakeUpSong.mp3")%>'

type="audio/wav" width="300" height="18">

<param name="controls" value="smallconsole" valuetype="data">

<param name="autostart" value="true" valuetype="data">

<param name="controller" value="true" valuetype="data">

</object>

**Applet example**

The following example shows how an applet can be included in an IBM portlet JSP.

*Figure 6. Standard portlet JSP with applet*

<applet codebase='<%=response.encodeURL("applet")%>'

code="MyApplet.class" width="150" height="150">

<param name="timeout" value="3600">

<param name="border" value="5">

<param name="font" value="TimesRoman|BOLD|18">

<param name="bgcolor" value="ffffff">

</applet>

**Supporting multiple devices, markups, and languages in IBM portlets**

WebSphere Portal supports PC browsers, i-mode and WAP phones, plus the ability to create and add support for other devices and clients, markup types, and languages. The challenge with supporting multiple devices, markups, and languages is to render content differently depending on the characteristics of the client, which is provided in the portlet request. For example, one browser may accept HTML 4.0 while another supports XHMTL 1.0. One WAP device might support four lines with 25 characters while another phone has its own PDA-style interface.

For IBM portlets, the aggregation component of WebSphere Portal allows you to package JSPs and resources in directories that match the client with the request, reducing the amount of programming you have to do to support multiple client types. JSPs that contain mostly text, such as help JSPs, can be translated directly rather than storing strings in a resource bundle. JSPs that do not use resource bundles need to be stored in the appropriate location. When a portlet uses a JSP for rendering of the portlet's content, the portal searches for and selects the proper JSP based on the client type (including browser), markup language, and locale indicated in the request. To include a JSP in a portlet, you must use the appropriate method:

// JSR 168 include

context.getRequestDispatcher("/jsp/View.jsp").include( request, response);

// IBM API include

context.include("/jsp/View.jsp", request, response);

To support multiple markup types and locales, the portlet's JSP's must be packaged in the WAR file using the following directory structure:

*jsp\_path*/*markup\_type* /*language* \_*region*/*client*/*jspname*.jsp

Where

**jsp\_path**

a path defined by the developer. For example, JSPs can be located in the root of the WAR file or in a jsp directory. However, this path must not include *mime-type*/*language*\_*region*\_*variant* . The include() method already locates the correct JSP also in those directories.

**markup\_type**

is either html, wml, or chtml .

**language**

is the language for this JSP, for example, en, ja, or de .

**region**

is the country, region, or territory of the JSP, for example, US, UK, or CA.

**client**

is the type of device. For example, it could indicate a JSP with browser-specific markup, such as ie or ns4. Manage Clients help describes how clients are identified by the portal server.

For example, if the client is using Internet Explorer 5 with language properties are set to English (United States), the method include(/mypath/mytemplate.jsp, portletRequest, portletResponse) would cause the portal server to look for the JSP in the following order.

1. /mypath/html/ie5/en\_US/mytemplate.jsp
2. /mypath/html/ie5/en/mytemplate.jsp
3. /mypath/html/ie5/mytemplate.jsp
4. /mypath/html/en\_US/mytemplate.jsp
5. /mypath/html/en/mytemplate.jsp
6. /mypath/html/mytemplate.jsp
7. /mypath/mytemplate.jsp

**Note:** Standard portlets must provide their own logic for handling requests from multiple locales and markup types.

Content accessed by a portlet JSP (for example, images or HTML pages) using ServletResponse.encodeURL() are not found by portal aggregation. To provide different versions of these resources based on client type, markup, and locale, the portlet must use PortletContext.include().

**Using JSTL in portlet JSPs**

The following example shows how to use JSTL to retrieve translated Strings from a resource bundle in your JSPs.

* The JSTL tag library is included at the beginning of the JSP.
* The <fmt:setBundle/> tag indicates the resource bundle to use.
* The <fmt:message/> tag indicates the key to look for in the resource bundle.
* For the image source, the encodeURL() method is used in a Java scriptlet.

<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>

...

<fmt:setBundle basename="nls.reminder"/>

...

<img border='0'

src='<%=response.encodeURL("task\_add.gif")%>'

title='<fmt:message key="add\_reminder"/>'

alt='<fmt:message key="add\_reminder"/>'/>

The JARs required to implement JSTL tags are included with the portal server. You should not package these JARs in your portlet's WAR file.

For more information about JSTL tags, see [JavaServer Pages Standard Tag Library](http://java.sun.com/products/jsp/jstl) and [JSP Standard Tag Library 1.1 implementation](http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html).

**IBM portlet API examples for Hello JSP**

The following examples are from the ibmHelloJSP.war sample.

*Figure 7. Example: JSP for Hello JSP portlet (IBM)*

<%@ taglib uri="/WEB-INF/tld/portlet.tld" prefix="portletAPI" %>

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" session="false" %>

<p class="portlet-font">Hello JSP!</p>

*Figure 8. Example: JSP for Hello JSP portlet (IBM)*

The PortletContext.include() method is used to invoke the JSP for the view mode.

package com.ibm.wps.samples.v4;

import org.apache.jetspeed.portlet.\*;

import java.io.\*;

public class HelloJSP extends PortletAdapter {

public void init(PortletConfig portletConfig) throws UnavailableException

{

super.init(portletConfig);

}

public void doView(PortletRequest request, PortletResponse response)

throws PortletException, IOException {

PortletContext context = getPortletConfig().getContext();

context.include("/jsp/View.jsp", request, response);

}

}

# TOPIC 4:- Storing user settings

Using a sample portlet as an example, learn how user settings can be stored and retrieved in portlets.

This topic describes how user settings can be stored and retrieved in portlets. Examples in this topic build off of the HelloJSP samples described in [Generating output](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsoutput.html). All sample portlets are available from the portlet catalog by searching for navcode 1WP10017Z. See [Sample portlets](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/jsrsamp.html) for more information.

For standard portlets, user settings are stored, set, and retrieved using the *PortletPreferences* object, which represents name/value pairs in <preference/> elements of the portlet deployment descriptor. Preferences can be marked read-only, in which case they can only be modified by someone with administrative privileges, typically using a configure mode. Otherwise, the portlets can be modified in any portlet mode.

In the following example, the userName preference is provided, but set to a null value. This preference is not marked read-only, so it can be changed by any user with edit access to the portlet.

<portlet-preferences>

<preference>

<name>userName</name>

<value></value>

</preference>

</portlet-preferences>

The portlet retrieves a reference to an instance of PortletPreferences by calling the getPreferences() method of the PortletRequest object. The store() method saves the information PortletData. The Java Portlet Specification allows the portlet to define a preference validator in the portlet deployment descriptor. If one is defined, then the validator is called before the store operation is performed.

The following example from jsrHelloUser.war builds on the [jsrHelloJSP.war](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsoutput.html) sample by adding the doEdit() and processAction() methods to allow the user to edit preferences. The getValue() method allows the portlet developer to designate a default value if the preference has not been set.

*Figure 1. Example: Java source for Hello User portlet (standard)*

package com.ibm.wps.samples.jsr;

import javax.portlet.\*;

import java.io.\*;

public class HelloUser extends GenericPortlet {

public void init(PortletConfig portletConfig) throws UnavailableException, PortletException

{

super.init(portletConfig);

// The default user name is obtained from the portlet configuration parameters

}

public void doView(RenderRequest request, RenderResponse response)

throws PortletException, IOException {

// set return content type

response.setContentType("text/html");

// Get the user's preferred name to display from preferences

// If no value is available, set it to "User"

String displayName = request.getPreferences().getValue("userName", "User");

// Add the display string to the portlet request to make it accessible by the view JSP

request.setAttribute("username", displayName);

PortletContext context = getPortletContext();

context.getRequestDispatcher("/jsp/View.jsp").include( request, response);

}

public void doEdit(RenderRequest request, RenderResponse response)

throws PortletException, IOException{

response.setContentType("text/html");

// Create the return URL for the cancel link of the edit page

PortletURL cancelUrl = response.createRenderURL();

cancelUrl.setPortletMode(PortletMode.VIEW);

// Preserve the Cancel URL in the request to make it accessible by the edit JSP

request.setAttribute("cancelUrl",cancelUrl.toString());

// For the "Save" button the return URI must include the "Save" action

PortletURL saveUrl = response.createActionURL();

saveUrl.setPortletMode(PortletMode.VIEW);

saveUrl.setParameter("save","save");

// Preserve the Save URL in the request to make it accessible by the edit JSP

request.setAttribute("saveUrl",saveUrl.toString());

String displayName = request.getPreferences().getValue("userName", "User");

request.setAttribute("username", displayName);

String jspName = getPortletConfig().getInitParameter("jspEdit");

PortletRequestDispatcher rd = getPortletContext().getRequestDispatcher("/jsp/Edit.jsp");

rd.include(request,response);

}

public void processAction(ActionRequest request, ActionResponse response)

throws PortletException, IOException{

PortletContext context = getPortletContext();

try{

String save = request.getParameter("save");

if (save != null){

PortletPreferences prefs = request.getPreferences();

prefs.setValue("userName",request.getParameter("username"));

prefs.store();

}

}

catch ( IOException ioe ){

context.log("An IO error occurred when trying to save the name.");

}

catch ( PortletException pe ) {

context.log("A portlet exception was thrown when trying to save the name.");

}

}

}

For IBM portlets, user settings are saved, retrieved, or deleted using the [PortletData](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsptcfgob.html) object. Portlets can store values in the PortletData object only when the portlet is in edit mode. If the portlet is on a group page, then information saved in PortletData is available to all users of the portlet. The portlet retrieves a reference to an instance of PortletData by calling the getData() method of the PortletRequest object. The store() method saves the information PortletData. Only data of the Java String type can be saved in the *PortletData* object.

PortletData is used in the ibmHelloUser.war sample to allow users to edit and save their names in which they are addressed in the greeting. This sample builds on the ibmHelloJSP sample by adding the doEdit() and the actionPerformed() methods, thus allowing users to edit the greeting and save their preferences.

* The default greeting to display is obtained from the [portlet deployment descriptor](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsptdesc.html) . The following example shows the portion of this descriptor with the configuration parameter defaultUserName set to "User".

*Figure 2. Example: Portlet deployment descriptor for ibmHelloUser*

<config-param>

<param-name>defaultUserName</param-name>

<param-value>User</param-value>

</config-param>

Configuration data that is set by the <init-param> tags are read-only and maintained for all users and every concrete portlet derived from the portlet. If you need to allow different configurations for each concrete portlet, then the data should be set in the <concrete-portlet> tag of the portlet deployment descriptor.

* The doView() method receives control prior to the standard display for this portlet. The *PortletData* object is accessed to obtain the string to display. If the user has not yet specified a string to display, the default string will be used. The string is stored in the *PortletRequest* object to make it available to the JSP that generates the view markup for this portlet (viewJSP).
* The doEdit() method receives control prior to the display of the edit page for this portlet. A return URI is created and passed to the JSP for edit mode using the *PortletRequest* object. The save action is included in the return URI using the addAction() method. The portal passes control to the *ActionListener* upon processing the save action. The *ActionListener* can preserve the user entered "edit" information in the persistent storage. See [Action events](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsptevents.html) for more information about *ActionListener* and portlet actions.

*Figure 3. Example: Java source for Hello User portlet (IBM)*

package com.ibm.wps.samples;

import org.apache.jetspeed.portlet.\*;

import org.apache.jetspeed.portlet.event.\*;

import java.io.\*;

public class HelloUser extends PortletAdapter implements ActionListener{

private String defaultString;

private String displayName;

public void doView(PortletRequest request, PortletResponse response)

throws PortletException, IOException

{

//Get the user's name to display from persistent storage

displayName = (String) request.getData().getAttribute("userName");

defaultString = request.getPortletSettings().getAttribute("defaultUserName");

// If user's preferred name is not found, display the default

if (displayName == null) {

displayName = defaultString; // set default string

}

// Add the display string to the portlet request to make it accessible by the view JSP

request.setAttribute("userName", displayName);

getPortletConfig().getContext().include("/jsp/View.jsp", request, response);

}

public void doEdit(PortletRequest request, PortletResponse response)

throws PortletException, IOException

{

// Create the return URI for the cancel link of the edit page

PortletURI cancelURI = response.createReturnURI();

// Preserve the Cancel URI in the request to make it accessible by the edit JSP

request.setAttribute("cancelURI", cancelURI.toString());

// For the "Save" button the return URI must include the "Save" action

// so the Action Listener for this portlet will be invoked

PortletURI saveURI = response.createReturnURI();

saveURI.addAction("save");

// Preserve the Save URI in the request to make it accessible by the edit JSP

request.setAttribute("saveURI", saveURI.toString());

//Get the user's name to display from persistent storage

displayName = (String) request.getData().getAttribute("userName");

defaultString = request.getPortletSettings().getAttribute("defaultUserName");

// If user's preferred name is not found, display the default

if (displayName == null) {

displayName = defaultString; // none found, set default string

}

// Add the display string to the request to make it accessible by the edit JSP

// as an initial value of the input field on the edit form

request.setAttribute("userName", displayName);

getPortletConfig().getContext().include("/jsp/Edit.jsp", request, response);

}

public void actionPerformed(ActionEvent event)

{

String action = event.getActionString();

PortletLog log = getPortletLog();

// If this is a save action, then see if the user specified a name

if ( action!=null ) {

if ( action.equals("save") ) {

PortletRequest request = event.getRequest();

PortletData portData = request.getData();

String userName = request.getParameter("userName");

try {

// Save the name specified by the user

if ( userName != null ) {

portData.setAttribute("userName", userName);

portData.store();

}

} catch ( AccessDeniedException ade ) {

} catch ( IOException ioe ) {

log.error( "<i><b>Couldn't write the user data to " );

log.error( "persistence because an I/O Error occurred.</b></i>" );

}

}

}

}

}

As mentioned previously, an *ActionListener* is implemented by HelloUser to process the save action. The user enters a name on the edit page and the actionPerformed() method of the *ActionListener* obtains the user-specified string from the *PortletRequest* object for storing in the user's persistent storage. The *ActionListener* is invoked prior to returning to the doView() method of the portlet, thus if the user failed to enter a name, the *ActionListener* can force the portlet to remain in edit mode, waiting for input from the user.

# TOPIC 5:- Preference layers and portlet modes

Get an overview of portlet configuration preference layers and portlet modes to best implement the different layers on which preferences for portlet views can be configured.

**Overview of portlet configuration preference layers and portlet modes**

Portlets that are written to comply with the Standard Portlet API access their configuration by using the PortletPreferences interface from the standard portlet API. Internally, WebSphere Portal keeps multiple levels of portlet configuration that are aggregated to form the representation of the portlets for the users:

| *Table 1. Portlet configuration preference layers and portlet modes* | | |
| --- | --- | --- |
| **Type of configuration** | **Description** | **Portlet mode** |
| Deployment descriptor preferences | They are associated with a deployed portlet, represented by a <portlet> tag in portlet.xml. They apply to all occurrences of that portlet on all pages for all users. |  |
| Administrator preferences | They are associated with a portlet definition, that is a particular copy of a portlet. They apply to *all* occurrences of that portlet definition on *all* pages for *all* users. The administration portlets allow you to create multiple copies of the same portlet with different configurations on the administrator level. | config |
| Shared preferences | They are associated with *a particular occurrence* of a portlet definition on a page and apply to *all* users who view the portlet on that page | edit\_defaults |
| Personalized preferences | They are associated with *a single user* and apply only to that particular user who views the portlet on the page. | edit, view, help |

For details about the tag for the edit\_defaults mode refer to [<portal-skin/> tags](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/dgn_ptlskin.html) under the tag [<portal-skin:portletEditDefaults/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/dgn_ptlskin.html#dgn_ptlskin__portleteditdefault).

**Accessing preferences**

WebSphere Portal merges the different preference keys and values that are defined on these levels into a single PortletPreferences object. When the same preference key is defined on multiple levels with different values, the most specific value (from bottom to top in the above list), takes precedence and values on more general levels are not available. This allows you to set up general default values for preferences that can then be customized by individual users. Preference values can be locked for user customization by declaring them read-only in the deployment descriptor. Read-only values can only be modified on the administrator level but can never be overridden in lower levels. An example:

1. Administrator **A** configures a database viewer portlet so that it displays the **Fish** view in the **Animals** database and shows **10** lines per page. The database setting is locked (read-only) so that users may customize the portlet but cannot use it to view a different database.
2. Editor **E** places the portlet on a page and decides that the **Mammals** view is a more interesting default.
3. User **U1** personalizes the portlet to display the **Birds** view with **20** lines per page instead of the defaults given by the administrator and the editor.
4. User **U2** personalizes only the number of displayed lines with a value of **55**, so user **U2** still sees the **Mammals** view.
5. Other users who view the same page will still see the **Mammals** view with **10** lines per page as set by the administrator and the editor.
6. Editor **E** now changes the default view to **Reptiles**. This does not affect **U1**, but only **U2** and other users that have not personalized that particular preference key.
7. Finally, **U1** resets the personalization of the database view. As a result, the shared default, which is now **Reptiles**, is shown again. The number of displayed lines still remains at the personalized value of **20**.

**Modifying preferences**

A preference is modified by using setValue. This normally occurs at the **personalized** layer and therefore affects only the current user. WebSphere Portal uses two special custom modes from the set of predefined custom modes in the Java Portlet Specification to allow setting up the more general preference levels:

* The edit\_defaults custom portlet mode is used to work directly on the shared preferences. In this case the personalized preferences level is not available.
* Similarly, the config mode is used to read and modify the administrator level of preferences.
* The deployment descriptor level of preferences can only change when the portlet is redeployed with a modified *portlet.xml*. It cannot be modified by portlet code.

| *Table 2. Portlet modes and preferences* | |
| --- | --- |
| **Portlet mode** | **Layer to which preference changes are written** |
| config | Administrator preferences |
| edit\_defaults | Shared preferences |
| All other modes: edit, view, help | Personalized preferences |

If a portlet wants to use these custom modes, they must be declared with <custom-portlet-mode> tags in the deployment descriptor. Entering one of these modes requires special access permissions. For details about access permissions refer to the appropriate sections. Portlets can enter these modes using the standard APIs for setting a portlet mode. These APIs are ActionResponse.setPortletMode and PortletURL.setPortletMode. However, normally the portlets do not need to do so, because the portal normally provides buttons in the portlet skin that allow to switch to these modes, if they are supported by the portlet. The administrator level of preferences for a particular portlet can also be modified without actually invoking the portlet itself by using the portlet management portlet.

Each call to PortletPreferences.store persists those values at the current preference level that were set by using setValue. Even if the value that is being set is identical to the currently applicable value, which has been set by the more general level, the new value is now stored at the current level and never inherited. That means, if you want to make use of the preferences hierarchy, so that later changes on a more general level are available (“shine through”) to the more specific levels, you must take care to set and store only those preferences that should not be inherited from the more general levels. Resetting of preferences as on the example above is done programmatically by using the reset method. This means that the value of the preference key that is reset will now be read from the next more general preference layer.

**Notes:**

1. The reset of a preference is fundamentally different from setValue(null): the latter will hide any default value that was set on a more general level.
2. If a preference is reset at all levels and has no value defined in portlet.xml, the default value passed to the getValue call will be returned.

**Programming recommendations**

The separation of preferences into different layers is mostly transparent to the portlet programmer. In the code, you access the aggregated preferences by using the PortletPreferences object. The portal will automatically select the appropriate values for the current user and portlet mode. The only place where the distinction of the preference layers becomes visible in a portlet is in the configuration views, that is in the portlet modes edit, edit\_defaults, and config. The different preferences and the different modes in which they are presented determine which part of the portlet behavior can be customized by end users and which part can only customized by editors or administrators.

In practice, it is often difficult for the programmer to anticipate how a portlet will be used in a particular environment. In many cases it should rather be decided by an administrator which of the preferences can be customized and at which level they should be customized. Therefore, program your portlets in such a way that these decisions are not determined by the code but, as far as possible, by the deployment descriptor. This makes it easier for the administrator later to determine to which extent and on which levels preferences can be customized.

**Note:** This can often be achieved by using the same code (and JSPs) for implementing all three configuration views, edit, edit\_defaults, and config, and simply deactivating or hiding those configuration settings that cannot be modified in the current mode. Settings that can only be modified by administrators (config mode) can be determined by the read-only state that is defined in portlet.xml. There is no standard API that can be used to indicate which preferences should only be modified in the shared configuration and which preferences should be locked for end-users. If this is required, we recommend to keep a special (invisible) preference locked-pref-keys that holds a list of those preference keys that can only be modified in edit\_defaults or config mode.

**Code examples:**

1. Define the preference names that should be locked for the end user in the portlet.xml:
2. <preference>
3. <name>locked-pref-keys</name>
4. <value>allowed-folders</value>
5. <value>folder</value>

</preference>

1. Verify that the preference is writable before allowing the user to change a preference value:
2. public boolean isWritable(String prefName) {
3. PortletMode mode = request.getPortletMode();
4. if (mode.equals(CONFIG\_MODE))
5. return true;
6. PortletPreferences prefs = request.getPreferences();
7. if (prefs.isReadOnly(prefName))
8. return false;
9. if (! mode.equals(EDIT\_DEFAULTS\_MODE)) {
10. String[] lockedKeys = prefs.getValues("locked-pref-keys", null);
11. if (lockedKeys != null && Arrays.asList(lockedKeys).contains
12. (prefName))
13. return false;
14. }
16. return true;

}

As mentioned above, for the normal display of your portlet, you do not need to distinguish whether a particular preference value is an administrator or a user setting. The portal will automatically select the appropriate values for the current user and portlet mode. Nevertheless, you need to consider the merging behavior for the different preference levels particularly in two scenarios:

* When you use multi-valued preferences, for example to store lists, no merging of values takes place; if the preference is personalized, the shared settings are simply hidden.

Example: If you want to implement a scenario where, for example, an administrator defines a part of a bookmark list that all users must see, and users can personalize the list by adding or removing their own entries, you need to use separate keys for administrator and user preferences and merge the values explicitly in the code.

* When values for multiple preference keys depend on each other, you can run into problems if one of the keys is personalized while another is taken from a shared preference level. If the more general level is now changed, the new combination of shared and personalized values may become invalid. In the example above, the possible values for the database view obviously depend on the selected database. Therefore, if the administrator levels of preferences is changed from the **Fish** view of the **Animals** database to a **Plants** database and a **Trees** view, all the specializations for the database view become invalid: preferences for user **U2** would still indicate the **Reptiles** view, but now on a **Plants** database.

There are multiple options to resolve the second problem; it depends on the logic of your portlet which of them is most appropriate:

1. You can always set all the dependent preference keys in a single operation by setting all of them explicitly before the same call to store.
2. You can even merge the dependent preference keys into a single string value. This is what you should do for combined values, for example x-y coordinates. This implies that no merging of preference levels can take place, therefore you cannot use this approach if you want to support a use case scenario in which administrator changes can be partially overridden by users.
3. In other cases, it is more appropriate to accept the possibility of invalid preference combinations and just provide suitable error handling; for the example above, the portlet should simply be prepared to handle the fact that the specified database view may be invalid and reset that preference key to go back to a - hopefully valid - default that was set on a more general level.
4. Another option to resolve this problem is the use of a PreferenceValidator to make sure that only valid preference combinations are available to the portlet. The following section [Preference Validators](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/adprflyrpltmd.html#adprflyrpltmd__prefval) describes how to do this.

**Preference Validators**

The Java Portlet Specification defines how a preference validator is defined in a portlet. If it is present, tools can use it to make sure that only valid preference combinations are saved for a portlet. WebSphere Portal uses a preference validator not only to validate preferences that are stored, but also to make sure that a change on a more general level has not invalidated the current view. Whenever a portlet retrieves its preferences in a particular context, the available combination of preference levels is validated, if necessary, and preference levels that have become invalid in combination with more general levels are ignored: they are not available to the portlet but are retained in the database. Therefore they can become available again when the conflicting preferences on the more general level are updated to be compatible again. This way you can isolate your validation checking and let the portal take care of providing the display logic with valid data. To verify whether the selective hiding of preference levels works as expected, you can enable traces for the following trace string:

com.ibm.wps.pe.pc.std.core.impl.WPPreferencesHierarchyImpl=low=enabled

This will generate trace logs for validation exceptions that occur during preference reads.

**Note:** Keep the following restrictions in mind when implementing a preference validator:

1. A failed validation on preference access causes the complete conflicting level to be hidden. If you want to be more selective about how preferences are made consistent, you need to use the following approach with self-contained preference validators.
2. Preference validators are intended to be used also by standalone tools. Therefore they should be self-contained and not require a portal run time environment, network, or back end access etc. This means that a preference validator would not be suitable for the example above, that is for checking that the specified document really exists in the document database.

**Compatibility with portlets that do not support the edit\_defaults portlet mode**

The Standard Portlet API supports the standard portlet modes edit, view, and help. Generic portlets that are written to comply with the Standard Portlet API support those portlet modes. But as they are not aware of the preference hierarchy implemented in WebSphere Portal, they usually do not support the custom portlet mode edit\_defaults. Instead, they only support the standard portlet mode edit for customization. The same applies to portlets that were written to comply with the Standard Portlet API for a previous version of WebSphere Portal, in which the edit\_defaults mode was not yet supported. Preference values that are set in any of the standard portlet modes are never shared between users, in accordance with the behavior assumed by the Java Portlet Specification, because they are always stored at the personalized preference level. Unfortunately, this behavior precludes even the required use case where an Editor wants to set up a read-only page for all users with the appropriate shared portlet configuration. Normally this would be done in the edit\_defaults mode, but as the portlet does not support that mode, this is not possible.

To enable editing of shared preferences on such portlets, WebSphere Portal provides a special compatibility mode, the edit\_defaults\_compatibility portlet mode. Like any other custom portlet mode, this mode must be explicitly enabled in the portlet deployment descriptor, but other than that, the portlet does not require any special coding. To the portlet, the compatibility mode will just appear as the standard edit mode that it already supports, but the API implementation will read and store the shared preference level instead. As the portlet code does not know about the compatibility mode, entering that mode is only possible from the icon or pull-down menu selection option that is provided on the portlet title bar. As the compatibility mode is mapped to edit mode for the portlet, a portlet that supports this mode must also support edit mode for the same content types in the portlet deployment descriptor. As an example, consider the following code snippet:

<supports>

<mime-type>text/html</mime-type>

<portlet-mode>edit</portlet-mode>

<portlet-mode>edit\_defaults\_compatibility</portlet-mode>

. . . . .

</supports>

In the standard portlet modes, such as edit, a portlet that supports the compatibility mode will still read and store personalized preferences only, and changes are not visible to other users. There is, however, a slight difference to the normal aggregation of preference layers: When a user personalizes the portlet, the preferences are stored in a private copy and not on a personalized layer. No merging takes place between shared and personalized preferences, and any subsequent changes on the shared preferences are not visible to a user who has personalized the portlet, no matter which preference keys are affected. This arrangement as been made to avoid issues with inconsistencies between dependent preference values, as portlet code might not be prepared to handle unexpected combinations of preference values resulting from the merging of shared and personalized levels (as described under [Accessing preferences](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/adprflyrpltmd.html#adprflyrpltmd__accpref)).

As an example, assume that the database viewer portlet mentioned under [Accessing preferences](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/adprflyrpltmd.html#adprflyrpltmd__accpref) was written without support for config and edit\_defaults modes, and allows you to customize all settings in the edit mode only. To set up the portlet on a shared page without requiring each user to customize it, its deployment descriptor has been modified to support the edit\_defaults\_compatibility mode. Editor **E** sets up the portlet on a shared page by using the compatibility mode to display the **Mammals** view from the **Animals** database with **10** lines per page. User **U** personalizes the portlet in edit mode to show **20** lines instead. At this point, a private copy of all preference keys is created, so if editor **E** changes the shared preferences to display the **Reptiles** view instead, **U** will continue to see the **Mammals** view when viewing the page.

This behavior is almost identical to Version 5.1 of WebSphere Portal, where the explicit distinction between shared and personalized preference levels, along with the support for the edit\_defaults mode, did not yet exist. The only difference is that **E** must now explicitly enable and use the edit\_defaults\_compatibility mode for changing shared preferences, while in Version 5.1 the same behavior was implicitly chosen in edit mode, because **E** has the necessary access permissions to change the shared configuration visible to all users.

**Note:** The modes edit\_defaults and edit\_defaults\_compatibility mode are mutually exclusive. A portlet can support only one of them at a time.

**TOPIC 6 :-**

**Message and trace logging**

Portlets can write message and trace information to log files, which are maintained in the [*wp\_profile\_root*](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/reference/wpsdirstr.html#wpsdirstr__wp_profile_root)/logs/WebSphere\_Portal directory. The log files help the portal administrator investigate portlet errors and special conditions and help the portlet developer test and debug portlets.

The PortletContext object in the standard portlet API includes a log() method. The [Hello User example](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbspers.html) shows how to log a message when exceptions are caught.

*Figure 1. Logging error messages in Hello User example*

public void processAction(ActionRequest request, ActionResponse response)

throws PortletException, IOException{

PortletContext context = getPortletContext();

try{

String save = request.getParameter("save");

if (save != null){

PortletPreferences prefs = request.getPreferences();

prefs.setValue("userName",request.getParameter("username"));

prefs.store();

}

}

catch ( IOException ioe ){

context.log("An IO error occurred when trying to save the name.");

}

catch ( PortletException pe ) {

context.log("A portlet exception was thrown when trying to save the name.");

}

}

The IBM Portlet API provides the *PortletLog* class, which has methods to write specific types of message and trace information to the logs.

* debug() - for trace information
* info() - for informational messages.
* error() - for error messages.
* warn() - for warning messages.

If an IBM portlet needs to access the portlet log multiple times in a method, it is good idea to assign the log reference to a variable, for example:

private PortletLog myLogRef = getPortletLog();

Since logging operations are expensive, the *PortletLog* class of the IBM Portlet API provides methods to determine if logging is enabled for a given level. The portlet should write to the log of a given level only if the log is tracking messages of that level. For example:

if( getPortletLog().isDebugEnabled() )

{

myLogRef.debug("Warning, Portlet Resources are low!");

}

**Enabling logging for portlets**

Portlet logging is enabled by setting the following trace strings in the log.properties file.

**Standard portlets**

javax.portlet.Portlet=all

**Note:** The Java Portlet Specification defines that log statements from the JSR portlets are also passed on to the ServletContext.log() method, which will additionally write them to System.out in the default configuration.

**IBM portlets**

org.apache.jetspeed.portlet.PortletLog=all

# TOPIC 7 :- Caching portlet output

Portlet output can be cached at the local application server or using a remote proxy server.

**Local cache settings**

If fragment caching is enabled on the application server, the local cache holds the complete output of the portlet by portlet window and portlet state. If the portlet is included on a page, and the cache contains valid markup for the requested portlet state, the portlet code is not called, but cached content is returned. An action or event on the portlet invalidates all cached content for the affected portlet window.

To enable local caching, check the Enable Portlet Fragment Cache option in the administrative console for the WebSphere Application Server. Refer to the WebSphere Application Server information center for detailed instructions.

The portlet indicates how long, in seconds, its output should be cached in the portlet deployment descriptor.

**Standard portlet cache settings**

<expiration-cache>300</expiration-cache>

<cache-scope>private</cache-scope>

**IBM portlet cache settings**

<cache>

<expires>300</expires>

<shared>no</shared>

</cache>

A value of -1 indicates that the portlet cache never expires. A value of 0 indicates that the portlet is never cached, which is also the behavior if the portlet descriptor does not provide cache settings. Both standard and IBM portlet deployment descriptors can also specify a cache scope that indicates whether cached content is public (shared) or per user. Cache settings for standard portlets can be modified in the portal administration.

By more precise detail, IBM portlets are cached via servlet caching. When you enable fragment caching, servlet caching is also enabled, as fragment caching requires servlet caching..

**Modifying the local cache at run time**

For standard portlets, the portlet window can modify the expiration time at run time using the CacheControl object, as follows:

renderResponse.getCacheControl().setExpirationTime(3000);

**Note:** Modifying the cache scope at run time is restricted.

The [getLastModified() method](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsptelem.html) of the IBM portlet API enables the portlet developer to inform the container when the current cache entry for the portlet should be invalidated, and therefore the portlet's content should be refreshed. The following example shows how a portlet that caches its output can change its content immediately to provide additional output. In the portlet's getLastModified() method, the time stamp is set to an attribute in the portlet's session.

*Figure 1. getLastModified() example*

public long getLastModified(PortletRequest request) {

PortletSession session = request.getPortletSession(false);

if(session == null) {

return System.currentTimeMillis();

}

if (session != null) {

Long lastModified = (Long) session.getAttribute(LAST\_MODIFIED);

if (lastModified != null) {

return lastModified.longValue();

}

}

return -1;

}

**Remote cache settings**

In an environment where a remote proxy server is used for caching, standard portlets can indicate cache settings, which are used by the portal server to determine how the page is cached on a remote proxy server. Refer to the section about how to tune your portal for more information on how remote caching is determined for the page. IBM portlets cannot provide individual cache preferences for the remote cache.

Standard portlets specify cache expiration and cache scope for remote caching in the deployment descriptor, in the same way as noted above under "local cache". These cache settings can be modified in the portal administration

# TOPIC 8 :- Accessing remote systems

The portlet API provides a ContentAccessService, which allows portlets to access remote systems or content from remote URLs, including URLs located on the other side of a proxy server. You should always use this service to access remote content if you cannot be certain whether a firewall will be present after the portlet is deployed into a production environment.

The following topics describe how portlets can use the content access service.

* [Standard portlet example](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsbackend.html#wpsbsbackend__jsr)
* [IBM example](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsbackend.html#wpsbsbackend__ibm)
* [getMarkup and character sets](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsbackend.html#wpsbsbackend__getmarkup)

See [Portlet services](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsptservice.html) for an overview of portlet services and how to access them.

**Standard portlet example**

The following example shows how a standard portlet uses the content access service to setup an HTTP input stream to a remote system. In this example, the method receives the URL to the backend system as a String (urlS). If the service cannot be found by JNDI, this sample writes a message to the log. However, standard portlets should always provide equivalent functionality when the portlet service cannot be located.

*Figure 1. Example: ContentAccessService*

import com.ibm.portal.portlet.service.\*;

import com.ibm.portal.portlet.service.contentaccess.\*;

...

private PortletServiceHome casHome = null;

...

public void init(PortletConfing config)

{

javax.naming.Context ctx = new javax.naming.InitialContext();

try {

Object home =

ctx.lookup("portletservice/com.ibm.portal.portlet.service.contentaccess.ContentAccessService");

if (home != null)

casHome = (PortletServiceHome) home;

} catch(javax.naming.NameNotFoundException ex) {

// if service is not available, write to the log

config.getPortletContext().log("Content Access Service is not available");

}

}

protected InputStream getHTTPInputStream(String urlS, PortletRequest request,

PortletResponse response) throws IOException {

getPortletConfig().getPortletContext().log("getHTTPInputStream: " + urlS);

if (casHome != null)

{

ContentAccessService contentAccessservice = (ContentAccessService) casHome.getPortletService

(ContentAccessService.class);

// content access service handles local and remote URLs transparently

return contentAccessservice.getInputStream( urlS, request, response );

} else

{

// no content access service available

return (new URL(urlS).openStream);

}

}

**Tip:** When accessing remote internet resources directly, the content is simply taken and written to the portlet's output. This means that all relative links to other pages or images will be broken. This can be solved by parsing the content or use some enhanced browser portlet.

The *ContentAccessService* can also open an SSL connection to remote applications. The code is the same as for nonsecure connections (see [Figure 1](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsbackend.html#wpsbsbackend__ctacc_example)) except that you specify a secure protocol (HTTPS) in the URL. However, the portal must be configured to support SSL in PortletServiceRegistryService.properties.

**IBM example**

The following example shows how a portlet written to the IBM portlet API uses the content access service to setup an HTTP input stream to a remote system. This example is similar to the standard example, except that the portlet uses the PortletContext.getService() method to retrieve the portlet service instead of JNDI.

*Figure 2. Content access service in IBM portlet*

import org.apache.jetspeed.portlet.service.ContentAccessService;

...

protected InputStream getHTTPInputStream(String urlS, PortletRequest request,

PortletResponse response)

throws MalformedURLException, IOException {

if(getPortletLog().isInfoEnabled()) {

getPortletLog().info("getHTTPInputStream: " + urlS);

}

try {

PortletContext context = getPortletConfig().getContext();

ContentAccessService contentAccessservice =

(ContentAccessService)context.getService(org.apache.jetspeed.portlet.service.

ContentAccessService.class);

} catch(PortletServiceException e) {

getPortletLog().error("MyPortlet: ContentAccessPortletService error: ",e);

throw new IOException("ContentAccessPortletService Error: " + e);

}

}

**getMarkup and character sets**

The getMarkup() and include() methods in ContentAccessService require a Web site to return the character set used to encode the content; this information is normally included in the HTTP response headers. However, sometimes Web sites do not specify the character set. In this case, ContentAccessService assumes the content is encoded with UTF-8. However, if the Web site contains non-English content (for example, double-byte character languages such as Japanese and Korean), content will not render properly.

Developers can work around this problem by using the getInputStream() method, which handles the character set conversion manually. For example:

InputStream stream =

contentAccessService.getInputStream(url, portletRequest, portletResponse);

Reader reader = new InputStreamReader(stream, mySpecifiedCharset);

# TOPIC 9 :- Portlet authentication

For resources protected by the portal, IBM® WebSphere® Portal uses CORBA credentials and an encrypted LTPA cookie to authenticate users. However, for backend systems that require their own authentication, portlets need to provide some form of authentication to access these remote applications. In order to provide a single sign-on user experience, portlets must be able to store and retrieve user credentials for their particular associated application and use those credentials to log in on behalf of the user. WebSphere Portal supports the use of a credential vault where users and administrators can safely store credentials for authentication. Portlets written to extract the user's credentials from the vault can hide the login challenge from the user.

The credential vault provides exactly this functionality. Portlets can use it through the credential vault portlet service (*CredentialVaultService*). See [Portlet services](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsptservice.html) for an overview of portlet services and how to access them. The following sections provide more information about the credential vault.

* [Credential vault organization](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsadvdev.html#wpsadvdev__vault_org)
* [Vault segments](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsadvdev.html#wpsadvdev__v_segments)
* [Vault slots](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsadvdev.html#wpsadvdev__cred_slots)
* [Credential objects](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsadvdev.html#wpsadvdev__cred_objects)
* [Credential vault usage scenarios](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsadvdev.html#wpsadvdev__use_scenarios)
* [Credential vault samples](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsadvdev.html#wpsadvdev__samples)
* [Changing the credential vault encryption](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsadvdev.html#wpsadvdev__encrypt)

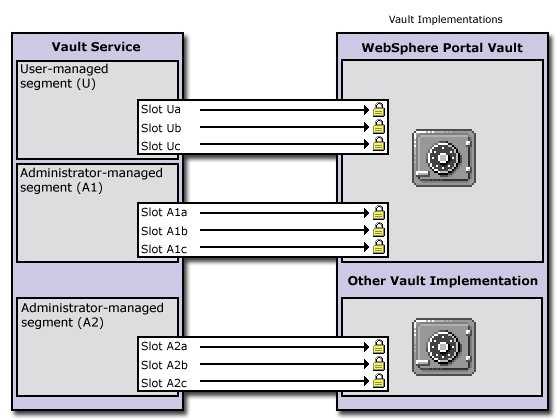
**Credential vault organization**

The portal server's credential vault is organized as follows:

* The portal administrator can partition the vault into several *vault segments*. Vault segments can be created and configured only by portal administrators.
* A vault segment contains one or more *vault slots*. Vault slots are the "drawers" where portlets store and retrieve a user's credentials. Each slot holds one credential.
* A vault slot is linked to a *resource* in a *vault implementation*.
  + A vault implementation is the place where users credentials are actually stored. Examples of vault implementations include the default database vault or the Tivoli Access Manager lock box.
  + The resource within the vault implementation corresponds to an application or backend system that requires its own authentication. Examples of resources include Lotus Notes, personnel records, or a bank account.

**Vault segments**

A vault segment is flagged to be either administrator-managed or user-managed. While portlets (on behalf of a portal user) can set and retrieve credentials in both types of segments, they are permitted to create vault slots only in user-managed vault segments. The following figure shows how administrator-managed vault segments can be distributed among different vault implementations. There is only one user-managed vault segment, and it resides in the default-customization vault provided by WebSphere Portal.



**Vault slots**

The credential vault provided by WebSphere Portal distinguishes between four different types of vault slots:

* Vault Slots in administrator-managed vault segments:
  + A **system slot** stores system credentials where the actual secret is shared among all users and portlets. It is a **shared slot** that belongs to an administrative segment. This slot type can be created through the Credential Vault administrative portlet, by adding a new slot in an a administrator-managed vault segment and mark it as shared.
  + An **administrative slot** allows each user to store a secret for an administrator-defined resource (for example, Lotus Notes). It is an **unshared slot** that belongs to an administrative segment. This slot type can be created through the Credential Vault administrative portlet, by adding a new slot in an a administrator-managed vault segment and do not mark it as shared.
* Vault Slots in user-managed vault segments:
  + A **shared user slot** stores user credentials that are shared among the user's portlets. It is a **shared slot** that belongs to the user segment. This slot type can be created through the Credential Vault portlet service by using the appropriate method and setting the argument as **true** for shared slot.
  + A **portlet private slot** stores user credentials that are not shared among portlets. It is an **unshared slot** that belongs to the user segment. This slot type can be created through the Credential Vault portlet service by using the appropriate method and setting the argument as **false** for shared slot.

The following tables shows the constraints applicable for the available vault slot types:

| **Vault Slot Type** | **Segment type** | **Shared** | **Creation through** | **Secret Sharing** |
| --- | --- | --- | --- | --- |
| **System Slot** | administrator-managed | true | Credential Vault administrative portlet | one secret per system - shared among all users and portlets |
| **Administrative Slot** | administrator-managed | false | Credential Vault administrative portlet | one secret per user - shared among all of user's portlets |
| **Shared User Slot** | user-managed | true | Credential Vault portlet service | one secret per user - shared among all of user's portlets |
| **Portlet Private Slot** | user-managed | false | Credential Vault portlet service | one secret per user and portlet entity - not shared among portlets |

**Credential objects**

The credential vault portlet service returns credentials in form of credential objects. The following are the base interface classes for all credential objects:

**Standard portlet API**

com.ibm.portal.portlet.service.credentialvault.credentials.Credential

**IBM Portlet API**

com.ibm.wps.portletservice.credentialvault.credentials.Credential

WebSphere Portal differentiates between passive and active credential objects.

* Passive credential objects are containers for the credential's secret. Portlets that use passive credentials need to extract the secret from the credential and do all the authentication communication with the backend itself.

*Example: Passive credential object use (pseudo code)*

Object userSecret = credential.getUserSecret();

< portlet connects to backend system authenticates using the user's secret >

< portlet can use the connection to communicate with the backend application >

< portlet takes care of logging at the backend >

* Active credential objects hide the credential's secret from the portlet; there is no way of extracting it out of the credential. In return, active credential objects offer business methods that take care of all the authentication.

*Figure 1. Example: Active credential object use (pseudo code)*

// log into the backend system

credential.login()

// get an authenticated connection to work with

URLConnection = credential.getAuthenticatedConnection();

// log out at the back end

credential.logout();

Active credential objects allow portlets to trigger authentication to remote servers using standard mechanisms such as basic authentication, HTTP form-based authentication, or POP3 authentication, without obtaining the credential secrets. They can ask the portal to authenticate on their behalf and then use already authenticated connections. From a security point of view the portlets never access the credential secrets and thus there is no risk a portlet could violate any security rules like, for example, storing the secret on the portlet session. While there might not always be an appropriate active credential class available, it is the preferred type of credential object to use.

All credential types that are available within the portal are registered in a credential type registry. WebSphere Portal provides a small set of credential types, but additional credential objects can be registered in this registry.

The ActiveCredential and PassiveCredential interfaces inherit from the Credential base interface. The following topics describe the different types of active and passive credential objects provided by WebSphere Portal:

**Passive Credential Objects:**

**SimplePassive**

This credential object stores secrets in the form of serializable Java objects. As the vault service does currently not support binary large object (BLOB) secrets, this is intended for future use only.

**UserPasswordPassive**

Credential object that stores secrets in the form of userid/password pairs.

**JaasSubjectPassive**

Credential object that stores secrets in form of *javax.security.auth.Subject* objects. Again, this kind of secret cannot currently be stored by the vault service. It is used as a transient credential that is not persisted but taken from the user's session.

**BinaryPassiveCredential**

Credential object that stores secrets in the form of a byte array.

**Active Credential Objects:**

**BinaryCredential**

This credential stores secrets in binary form.

**UserPasswordCredential**

The following credentials store the user's secret as a userid/password pair.

**HttpBasicAuth**

This credential object stores userid/password secrets and supports HTTP Basic Authentication.

**HttpFormBasedAuth**

This credential object stores userid/password secrets and supports HTTP Form-Based Authentication.

**JavaMail**

A credential object that stores userid/password pairs and leverages the authentication functionality of the javax.mail API.

**JaasSubjectCredential**

The following credentials store the user's secret as a JAAS Subject. These credentials are not persisted in the Credential Vault but created from the user's JAAS Subject associated with the current session.

**LtpaToken or LtpaToken2**

A credential object for authenticating with a backend system that is within the same WebSphere Application Server single sign-on domain as the portal.

**Note:** LtpaToken2 is only available if attribute propagation is enabled.

**Computer Associates *e*Trust SiteMinder Token**

A credential object for authenticating with a backend system that is within the same *e*Trust SiteMinder single sign-on domain as the portal. This credential should be used when *e*Trust SiteMinder is used as an authentication proxy for the portal.

**WebSealToken**

A credential object for authenticating with a backend system that is within the same WebSEAL single sign-on domain as the portal. This credential should be used when a WebSEAL authentication proxy is used by the portal.

**SSMTokenCredential**

A credential object for authenticating with a back-end system that is within the same Openwave SSM single sign-on domain as the portal. This credential should be used when Openwave SSM is the authentication proxy for the portal authentication. The credential needs to be provided in form of the portal's JAAS Subject containing a private credential that holds the user's SSMCredential.

**Storing credential objects in the PortletSession**

Credential objects do not implement java.io.Serializable - they cannot simply be stored in the PortletSession. This is for security reasons. Because the credential classes store the actual credential secret as one of their private attributes, the secret could be found by anyone who has access to the application server session database.

However, you can store a credential object in the PortletSession as long as you ensure sure that it is not serialized in a cluster setup. One way of doing this would be to define a credential container class that stores the actual credential object as a transient member. This container object can then be stored in the PortletSession without any problems, you only have to make sure to check whether the credential object got lost during serialization and in this case retrieve it from the vault again.

*Figure 2. Example: Credential object session container*

import com.ibm.portal.portlet.service.credentialvault.credentials.Credential;

public class CredentialSessionContainer implements java.io.Serializable

{

private transient Credential credential = null;

public void setCredential(Credential cred) {this.credential = cred;}

public Credential getCredential() {return credential;}

public boolean hasCredential() {return credential != null;}

}

**Credential vault usage scenarios**

Portlets that need a credential to complete their service have two options:

1. Use an existing slot that has been defined by the portal administrator in an administrator-managed vault segment.
2. Create a new slot in the user-managed vault segment.

The option you choose depends on how the portlet will be used. Generally, the best solution hides the technical details of the credential vault from users. Additionally, Active Credentials of the type JaasSubjectCredential provide the portlet with the ability to access resources within the same single sign-on domain as the portal. The following are some example scenarios for the use of slots.

**Intranet Lotus Notes mail portlet**

A company has an intranet employee portal. Each portal user has a Lotus Notes mail server account and a Lotus Notes mail portlet will be deployed and pre-configured on one of the employee's default portal pages.

Design solution:

The Notes mail portlet needs to store the user's Notes password. As most users will actually use this portlet, the administrator needs to create a "Lotus Notes Credential Slot" for it through the Credential Vault administrative portlet. Using the portlet's configure mode, the administrator sets the vault slot ID for all portlet entries. The portlet allows users to set their personal Notes password in edit mode. The portlet can store each user's password in the credential vault.

If the company would use a Lotus Domino server within the same single sign-on domain as portal, it would also be possible to use an LTPAToken Credential based on the user's JAAS Subject. This credential offers access to the domino server through an authenticated connection that reuses the user's LTPA token.

**Stock of inventory portlet**

A company's buying department runs a portal that integrates different legacy applications. One of these applications is a mainframe ordering application that directly connects to the systems of the suppliers. Several employees use the ordering portlet. However, the mainframe application is secured by a single system ID; it does not support several user accounts.

Design solution: The ordering portlet needs to access the ordering application under the system ID. The administrator configures the vault slot ID during portlet deployment. The portal administrator therefore creates a vault slot in an administrator-managed vault segment and marks it as a system credential. The administrator uses the **Credential vault** portlet to store the ordering system ID and password in this slot. The buying department's employees do not have to be concerned with credentials at all.

**Internet Mail federating POP3 portlet**

An Internet community portal offers, among other features, a mail-federating portlet that can be used by a portal user to collect mail from a number of POP3 mail accounts.

Design solution: The mail federating portlet is just another feature of the community portal and thus is likely to be used only by some of the portal users. Furthermore, it is not clear from the outset how many mail accounts a user wants to federate. Therefore, it does not make sense for the portal administrator to create a vault slot for this portlet. Instead, the portlet provides users with a comfortable configuration in edit mode. Users can add as many POP3 mailboxes as necessary. The portlet creates a vault slot for each of the user's mailboxes in the user-managed vault segment.

Theoretically, a user could configure two instances of the portlet on a page, one for business accounts and one for private mail accounts. Therefore, and because it most likely doesn't make sense to share the user's mail credentials with other portlets, the portlet created vault slots are better marked as portlet private.

**Credential vault samples**

This section contains sample code for using the credential vault service. See the Portlet API Javadoc for further information about the methods of the CredentialVaultService.

**Standard portlet example from Rational Application Developer**

When you use the portlet wizard in Rational Application Developer to create portlets that use the credential vault portlet service, the wizard generates a SecretManager class for you that handles common tasks. The session bean generated by the wizard includes getters and setters for the secret type and vault slot name. The following examples show code that could be generated for you, depending on your selections, when you create a JSR 168 portlet project.

**Note:** This code is only a demonstration of how you can use the credential vault. Any code that is generated must be further customized for your specific application requirements.

1. **Retrieve the credential vault portlet service.**

This code is placed in the init() method of the SecretManager and is called from the init() of the main portlet class. See [Accessing portlet services](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsaccpserv.html) for general information about how portlet services are retrieved using JNDI.

*Figure 3. Retrieving the credential vault service*

public static void init(PortletConfig config) throws PortletException {

try {

if( vaultService == null ) {

Context ctx = new InitialContext();

PortletServiceHome cvsHome =

(PortletServiceHome)ctx.lookup("portletservice/com.ibm.portal.portlet.

service.credentialvault.CredentialVaultService");

if (cvsHome != null) {

vaultService = (CredentialVaultService)cvsHome.getPortletService

(CredentialVaultService.class);

}

}

} catch (Exception e) {

throw(new PortletException("Error on init()", e));

}

}

1. **Set the credential.**

The portlet's processAction() method gets the USERID and PASSWORD parameters on the action request from the edit JSP. If both parameters are not null, then these are used to set the credential.

*Figure 4. Getting the credentials on the action request*

if( request.getParameter(USER\_SUBMIT) != null ) {

// Set userId/password text in the credential vault

PortletSessionBean sessionBean = getSessionBean(request);

if( sessionBean!=null ) {

String userID = request.getParameter(USERID);

String password = request.getParameter(PASSWORD);

// save only if both parameters are set

if(userID!=null && password!=null && !userID.trim().equals("") &&

!password.trim().equals("")) {

try {

SecretManager.setCredential(request,sessionBean,userID,password);

}

catch (Exception e) {

//Exception Handling

}

}

}

}

The setCredential() method in the SecretManager class determines if the portlet can write to the slot and if the slot ID has content. If so, it uses the setCredentialSecretUserPassword() method of the credential vault service to set the credential to the slot.

*Figure 5. Setting the credential*

public static boolean setCredential(PortletRequest portletRequest,

PortletSessionBean sessionBean,

String userID,

String password) throws PortletException {

try {

if( isWritable(sessionBean) ) {

String slotId = getSlotId(portletRequest,sessionBean,true); // create

slot if necessary

if( slotId != null ) {

vaultService.setCredentialSecretUserPassword(slotId,userID,password.

toCharArray(),portletRequest);

return true;

}

}

}

catch( CredentialVaultException e) {

//Exception Handling

}

return false;

}

1. **Get a slot for storing the user's credentials.**

The setCredential() method calls this method to either create a new slot or use an existing, accessible slot.

* + The slot name is set as a portlet preference so that it can be changed by the administrator to any slots created using the portal administration interface.
  + For a portlet private slot, this method creates a new slot if the slot name from the portlet preferences is null. The createNewSlot() method is another custom method of SecretManager that uses to createCredentialSlot() method of the credential vault service. This method can only create non-system slots and returns a CredentialSlotConfig object. The slotID of the new slot is stored in the portlet preferences and is used to update and retrieve the portlet private credentials.
  + For a shared slot, this method searches for a shared resource name from accessible slots. If a shared slot is not available, it creates a new slot.

*Figure 6. Creating or using an existing slot*

private static String getSlotId(PortletRequest portletRequest,

PortletSessionBean sessionBean,

boolean bCreate) throws PortletException {

String slotId = null;

String slotName = sessionBean.getVaultSlotName();

switch( sessionBean.getSecretType() ) {

case SECRET\_PORTLET\_PRIVATE\_SLOT:

PortletPreferences prefs = portletRequest.getPreferences();

String prefsKey = ".slot."+portletRequest.getRemoteUser()+".

"+slotName;

slotId = prefs.getValue(prefsKey,null);

if( slotId==null && bCreate ) {

slotId = createNewSlot(portletRequest,slotName, true); //

create private slot

if( slotId != null ) {

try {

prefs.setValue(prefsKey,slotId);

prefs.store();

}

catch( Exception e ) {

throw(new PortletException("Error on PortletPreferences.

store()", e));

}

}

}

break;

case SECRET\_SHARED\_SLOT:

try {

Iterator it = vaultService.getAccessibleSlots(portletRequest);

while( it.hasNext() ) {

CredentialSlotConfig config = (CredentialSlotConfig)it.next() ;

//searches for shared resource name

if( config.getResourceName().startsWith(slotName ) ) {

slotId = config.getSlotId();

break;

}

}

if( slotId==null && bCreate )

slotId = createNewSlot(portletRequest,slotName,false);

// create shared slot

}

catch( CredentialVaultException e) {

// exception handling goes here

}

break;

default:

slotId = slotName;

break;

}

return slotId;

}

1. **Retrieving the credential**

Both the doView() and doEdit method call getCredential() from the SecretManager. For the case of passive credentials, this portlet sets the USERID and PASSWORD returned from this method as attributes on the request.

*Figure 7. Setting the passive credential*

StringBuffer userId = new StringBuffer("");

StringBuffer password = new StringBuffer("");

try {

SecretManager.getCredential(request,sessionBean,userId, password);

}

catch( Exception e ) {

getPortletContext().log("Exception on SecretManager.getCredential(): "+e.getMessage());

}

request.setAttribute(USERID,userId.toString());

request.setAttribute(PASSWORD,password.toString());

The getCredential() method of SecretManager handles UserPasswordPassiveCredential types. The type of credential is set as an Integer in the portlet preferences. The getCredential() method of the credential vault service is used to obtain the credentials stored in the slot.

*Figure 8. getCredential() method*

public static void getCredential(PortletRequest portletRequest,

PortletSessionBean sessionBean,

StringBuffer userid,

StringBuffer password) throws PortletException {

try {

String slotId = getSlotId(portletRequest,sessionBean,false);

if( slotId != null ) {

UserPasswordPassiveCredential credential =

(UserPasswordPassiveCredential)vaultService.getCredential

(slotId,CredentialTypes.USER\_PASSWORD\_PASSIVE,new

HashMap(),portletRequest);

if( credential != null) {

userid.append(credential.getUserId());

password.append(String.valueOf(credential.getPassword()));

}

}

}

catch( CredentialVaultException e) {

// exception handling goes here

}

}

**IBM example that reads a credential from the vault**

The following is an example of an IBM portlet that uses a userid/password credential to log into a Web application that is secured with an HTTP form-based login page. The example shows how the CredentialVaultService is used to read a credential from the vault and how to use this credential. It does not show how in the portlet's edit mode the portlet queries the user for his/her secret and stores it in the vault. This example is incomplete and should not be pasted into working code.

*Figure 9. Example: Portlet using the CredentialVault PortletService with a HttpFormBasedAuthCredential*

import org.apache.jetspeed.portlet.service.PortletServiceException;

import com.ibm.portal.portlet.service.credentialvault.CredentialVaultService;

import com.ibm.portal.portlet.service.credentialvault.CredentialSecretNotSetException;

import com.ibm.portal.portlet.service.credentialvault.credentials.

HttpFormBasedAuthCredential;

...

public void doView (PortletRequest request, PortletResponse response)

throws PortletException, IOException

{

// get output stream and write out the results

PrintWriter writer = response.getWriter();

// get the CredentialVault PortletService

PortletContext context = this.getPortletConfig().getContext();

CredentialVaultService service =

(CredentialVaultService) context.getService(CredentialVaultService.class);

// retrieve slotId from persistent portlet data

String slotId = (String) request.getData().getAttribute("VaultTestSlotID");

if (slotId == null) {

writer.println("<h2>Credential not found. Please set it in the edit mode! </h2>");

return;

}

// bundle the credential config data mostly based on the backend application's login

form

HashMap config = new HashMap();

config.put( HttpFormBasedAuthCredential.KEY\_USERID\_ATTRIBUTE\_NAME, "j\_userid");

config.put( HttpFormBasedAuthCredential.KEY\_PASSWORD\_ATTRIBUTE\_NAME, "j\_password");

config.put( HttpFormBasedAuthCredential.KEY\_LOGIN\_URL, "EAI.yourco.com/myapp/

j\_security\_check");

config.put( HttpFormBasedAuthCredential.KEY\_LOGOUT\_URL,"EAI.yourco.com/myapp/

quit.jsp");

config.put( HttpFormBasedAuthCredential.KEY\_USE\_AUTH\_COOKIES, Boolean.TRUE);

List formData= new ArrayList();

formData.add("action=Login");

// add the POST data that the form's login button will generate (name=value)

config.put( HttpFormBasedAuthCredential.KEY\_FORM\_DATA, formData);

// get the actual credential from the credential vault

HttpFormBasedAuthCredential credential;

try {

credential = (HttpFormBasedAuthCredential)

service.getCredential(slotId, "HttpFormBasedAuth", config, request);

} catch (PortletServiceException serviceExc) {

writer.println("<h2>Credential vault error, please contact your admin! </h2>");

return;

} catch (CredentialSecretNotSetException userExc) {

writer.println("<h2>Credential not set. Please set it in the edit mode! </h2>");

return;

}

try {

// use credential object to log in at the backend server

credential.login();

// get an authenticated connection to the backend server and send the actual request

connection = credential.getAuthenticatedConnection("EAI.yourco.com/request.jsp");

// Work with the connection: send an HTTP GET or POST and evaluate the response

[...] // your business code

// use credential object to log out at the backend server

credential.logout();

} catch (IOException exc) {

writer.println("<h2>Single-sign-on error, login at backend failed! </h2>");

return;

}

// get output stream and write out the results

PrintWriter writer = response.getWriter();

}

**Changing the credential vault encryption**

WebSphere Portal supports plugging in different vault adapters for the storage and retrieval of credentials. The default vault adapter that ships with WebSphere Portal stores user credentials in the portal configuration database. By default, the passwords are only obfuscated, but not encrypted.

# TOPIC 10 :- Client profile information (CC/PP) in portlets

The Portal provides a standard API named "CC/PP" for accessing client profiles; learn how the client profile can be accessed through a request attribute. Learn about the attributes and components that are supported by the default profile implementation in the portal.

For each incoming request, the portal tries to determine the client that issued the request. It is important to determine the client in order to know its capabilities and respond properly. The portal database contains a repository of client profiles, providing all relevant information about the supported clients. The Portal uses JSR 188, which provides a standard API named "CC/PP" for accessing client profiles.

The Java Portlet Specification allows portlets to access the client profile through a request attribute. See following code snippet, how to get the profile in a portlet and to access an profile attribute.

*Figure 1. Accessing an attribute of the client profile*

import javax.ccpp.Attribute;

import javax.ccpp.Profile;

.........

Profile clientProfile =

(Profile) portletRequest.getAttribute(PortletRequest.CCPP\_PROFILE);

String vendor = null;

if (clientProfile != null) {

Attribute attribute = clientProfile.getAttribute("Vendor");

if (attribute != null) {

vendor = attribute.toString();

}

}

The following attributes and components are supported by the default profile implementation in the portal. The proprietary attributes supported by the portal default implementation, but not part of the UAPROF dictionary, are all collected in the component ProcessingInstructions.

The following table lists all attributes required for WebSphere Portal to work properly.

| **Name** | **Component** | **Type** | **Description** | **Example Value** |
| --- | --- | --- | --- | --- |
| Vendor | HardwarePlatform | Literal | Name of the vendor manufacturing the browser or device | "Netscape" |
| BrowserName | BrowserUA | Literal | Name of the browser user agent associated with the current request | "Navigator" |
| BrowserVersion | BrowserUA | Literal | Version of the browser | "6.x" |
| MarkupName | ProcessingInstructions | Literal | Markup that the browser or device accepts | "html" |
| MarkupVersion | ProcessingInstructions | Literal | Version of the markup | "ns6" |
| CcppAccept | SoftwarePlatform | Literal | List of content types the device supports | "text/html" |
| HtmlVersion | BrowserUA | Literal | Version of Hyper Text Markup Language (HTML) supported by the browser | "4.0" |
| HtmlCSS | BrowserUA | Boolean | Indicates whether cascaded style sheets can be used with this browser |  |
| FramesCapable | BrowserUA | Boolean | Indicates whether the browser is capable of displaying frames |  |
| IFramesCapable | BrowserUA | Boolean | Indicates whether the browser is capable of displaying inline frames |  |
| JavaAppletEnabled | BrowserUA | Boolean | Indicates whether the browser supports Java applets |  |
| JavaScriptEnabled | BrowserUA | Boolean | Indicates whether the browser supports JavaScript |  |
| TablesCapable | BrowserUA | Boolean | Indicates whether the browser is capable of displaying tables |  |
| HtmlNestedTable | BrowserUA | Boolean | Indicates whether the browser is capable of displaying nested tables |  |
| WmlVersion | WapCharacteristics | Literal | Wireless Markup Language (WML) version supported by the device | "1.1" |
| WmlTable | WapCharacteristics | Literal | Indicates whether the WML device is capable of displaying tables |  |
| FragmentIdentifier | ProcessingInstructions | Boolean | Indicates whether the browser or device supports fragment identifiers |  |

**PORTLET SERVICES**

# TOPIC 1:- Accessing portlet services

Using an example, learn how a standard portlet can retrieve and use a sample portlet service. Accessing a portlet service requires a JNDI lookup for a *PortletServiceHome*. To use the portlet service, you retrieve a service object from the home, cast it to the service-specific interface and invoke service methods.

This section describes how a portlet can invoke a portlet service. See [Portlet services](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsptservice.html) for a general overview of portlet services. The following example shows how a standard portlet can retrieve and use a sample portlet service. The service implementation and deployment is explained in [Creating your own portlet service](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsservice.html).

Accessing a portlet service requires a JNDI lookup for a *PortletServiceHome*. As this is may be a rather expensive operation, you should do it in the init() method of the portlet and store the returned object in an instance variable:

import javax.portlet.\*;

import com.ibm.portal.portlet.service.\*;

private PortletServiceHome helloServiceHome = null;

...

public void init(PortletConfig config)

{

javax.naming.Context ctx = new javax.naming.InitialContext();

try {

Object home =

ctx.lookup("portletservice/sample.portletservice.HelloService");

if (home != null)

helloServiceHome = (PortletServiceHome) home;

} catch(javax.naming.NameNotFoundException ex) {

// we can do without the service, if it is not available

config.getPortletContext().log("No hello service available");

}

}

The example is written to gracefully handle a situation where the service is not available because the service is optional for the functionality of the portlet. If possible, you should write your portlets so that they are portable across portal installations, whether they support a given portlet service or not. Such portlets can still run on portals that have not support for portlet services at all and do not even provide the IBM-specifc *PortletServiceHome* class, because this class is only loaded when the service is actually registered in JNDI.

To use the portlet service, you retrieve a service object from the home, cast it to the service-specific interface and invoke service methods:

public void doView(RenderRequest request, RenderResponse response)

{

if (helloServiceHome != null)

{

HelloService service =

(HelloService) helloServiceHome.getPortletService(HelloService.class);

service.sayHello(request, response);

}

//... do other stuff

}

Note that, while it is good practice to store the *PortletServiceHome* object in an instance variable, you must not store the actual service object, because references to service objects may not be held for longer than a single request.

# TOPIC 2:- Creating your own portlet service

Write a portlet service by defining the interface, writing the service implementation, making the service accessible for IBM portlets, and registering the service.

Writing a portlet service consists of four steps:

1. [Defining the interface](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsservice.html#wpsbsservice__def_service)
2. [Writing the service implementation](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsservice.html#wpsbsservice__wri_service)
3. [Making the service accessible for IBM portlets](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsservice.html#wpsbsservice__conversion)
4. [Registering the service](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsservice.html#wpsbsservice__reg_service)

The service provider interfaces can be used to write portlet services for IBM portlets as well as for Standard portlets. The IBM Portlet API has been deprecated for WebSphere Portal Version 6.1, but is still supported. No new functionality will be added and it is recommended that you use the Standard portlet API. For backward-compatibility, the service provider interfaces from the org.apache.jetspeed.portlet.service.spi packages are supported. For writing new portlets services, however, you should use only the service provider interfaces described in this text.

**Defining the interface**

This step is not required if you want to implement your service against an existing interface. Defining a portlet service interface requires the same careful considerations as defining any public API interface. A portlet service interface must extend the PortletService interface defined in the com.ibm.portal.portlet.service package. The following is an example interface for the HelloWorldService .

*Figure 1. Extending the PortletService interface*

package sample.portletservice;

import java.io.IOException;

import javax.portlet.RenderRequest;

import javax.portlet.RenderResponse;

import com.ibm.portal.portlet.service.PortletService;

public interface HelloService extends PortletService

{

/\*\* print a nice greeting \*/

public void sayHello(RenderRequest request, RenderResponse response)

throws IOException;

}

**Writing the service implementation**

The service implementation must implement the PortletServiceProvider interface of the com.ibm.portal.portlet.service.spi package to be able to make use of the portlet service life cycle methods in addition to your service interface. The PortletServiceConfig parameter of the init() method allows you, for example, to access the configuration of the service (see [Registering the service](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbsservice.html#wpsbsservice__reg_service) for more information).

*Figure 2. Implementing the PortletServiceProvider interface*

package sample.portletservice;

import java.io.IOException;

import java.io.PrintWriter;

import java.util.prefs.Preferences;

import javax.portlet.RenderRequest;

import javax.portlet.RenderResponse;

import com.ibm.portal.portlet.service.spi.PortletServiceProvider;

public class HelloServiceImpl

implements HelloService, PortletServiceProvider {

private String message;

// called by the portal when the service is initialized

public void init(Preferences servicePreferences) {

// read the message from the configuration, default is "Hello"

message = servicePreferences.get("message", "Hello");

}

public void sayHello(RenderRequest request, RenderResponse response)

throws IOException {

String user = request.getRemoteUser();

if (user == null)

// no user logged in

user = "Stranger";

PrintWriter out = response.getWriter();

out.print(message);

out.print(", ");

out.print(user);

}

}

**Making the service accessible for IBM portlets**

This step is optional. If you want your portlet service to be available for IBM portlets, you need to create an additional service interface that extends org.apache.jetspeed.portlet.service.PortletService and provides the same functionality.

*Figure 3. Extending the PortletService interface for IBM portlets*

package sample.portletservice;

import java.io.IOException;

import org.apache.jetspeed.portlet.PortletRequest;

import org.apache.jetspeed.portlet.PortletResponse;

import org.apache.jetspeed.portlet.service.PortletService;

public interface HelloServiceIBM extends PortletService {

/\*\* print a nice greeting \*/

public void sayHello(PortletRequest request, PortletResponse response) throws IOException;

}

You can have a single implementation that is registered for both interfaces and implements both. If the service methods take arguments that are classes or interfaces from the portlet API, the method signatures are different for the two service interfaces. You can still use a common implementation for both interfaces by using the APIConverterFactory class of the com.ibm.portal.portlet.apiconvert package. This class includes methods that wrap objects from the IBM portlet API, such as the PortletRequest and PortletSession, and implement the corresponding standard portlet API objects on the service side, so that you can re-use your service implementation for standard portlets.

*Figure 4. Using the APIConverterFactory class*

package sample.portletservice;

import java.io.IOException;

import java.io.PrintWriter;

import java.util.prefs.Preferences;

import javax.portlet.RenderRequest;

import javax.portlet.RenderResponse;

import org.apache.jetspeed.portlet.PortletRequest;

import org.apache.jetspeed.portlet.PortletResponse;

import com.ibm.portal.portlet.apiconvert.APIConverterFactory;

import com.ibm.portal.portlet.service.spi.PortletServiceProvider;

public class HelloServiceImpl2

implements HelloService, HelloServiceIBM, PortletServiceProvider {

private String message;

// called by the portal when the service is initialized

public void init(Preferences servicePreferences) {

// read the message from the configuration, default is "Hello"

message = servicePreferences.get("message", "Hello");

}

public void sayHello(RenderRequest request, RenderResponse response)

throws IOException {

String user = request.getRemoteUser();

if (user == null)

// no user logged in

user = "Stranger";

PrintWriter out = response.getWriter();

out.print(message);

out.print(", ");

out.print(user);

}

public void sayHello(PortletRequest request, PortletResponse response)

throws IOException {

sayHello (APIConverterFactory.getInstance().getRenderRequest(request),

APIConverterFactory.getInstance().getRenderResponse(response));

}

}

**Registering the service**

1. Put all service interface and implementation classes into a JAR file.
2. Place the JAR file in the [*PortalServer\_root*](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/reference/wpsdirstr.html#wpsdirstr__wp_root)/shared/app directory.
3. Register the new portlet service with the WP PortletServiceRegistryService resource environment provider in the WebSphere Application Server administration console.
   * Create an entry to register the implementation in the JNDI directory. The name for this entry is jndi.*service\_interface* and the value is *service\_implementation*. The fully qualified service interface name can then be used to lookup the service.
   * If you provide a service interface for IBM portlets, create an entry to register the implementation for this interface as well. The name for this entry is *service\_interface* and the value is *service\_implementation*. Services that are registered this way are not retrieved from JNDI, but from the PortletContext interface of the IBM portlet API.
   * *Optional:* Provide configuration parameters for the implementation. The name for this entry is *service\_implementation*.parameter and the value is the desired parameter value.
4. Restart WebSphere Portal to activate the new settings.

In the following example, HelloService is the name of the portlet service, HelloServiceIBM is the name of the interface for IBM portlets, and the message configuration parameter is set with the value Greetings.

**Note:** The colon (**:**) used in previous versions of WebSphere Portal to designate JNDI entries with jndi: is not supported for resource environment providers. Use jndi. instead.

To register this portlet service, add the following property names and values to the PortletServiceRegistryService:

|  |  |
| --- | --- |
| **Property name** | **Value** |
| jndi.sample.portletservice.HelloService | sample.portletservice.HelloServiceImpl2 |
| sample.portletservice.HelloServiceIBM | sample.portletservice.HelloServiceImpl2 |
| sample.portletservice.HelloServiceImpl2.message | Greetings |

**Tip:** To check whether your service has been registered successfully, use the application server *dumpNamespace* tool. The following command, executed from the [*was\_profile\_root*](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/reference/wpsdirstr.html#wpsdirstr__was_profile_root)/bin directory, will list all portlet service entries in JNDI:

dumpNamespace -port bootstrap\_port -root server -startAt portletservice

Using the application server's administrative console, you can find the bootstrap port of your portal server in the "End Points" section of the settings for the server "WebSphere\_Portal".

**STRUTS PORTLET FRAMEWORK**

# TOPIC 1:- Changes to Struts application code

Learn about the main differences between a servlet-based Struts application and a Struts application created for the portal environment.

* [Comparison of servlets and portlets](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__serv_compare)
* [Saving information for rendering the view](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__render_proc)
* [Response object](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__pseudo)
* [sendError() processing](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__senderr)
* [Saving information in a bean](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__save_bean)
* [Forwards and redirects](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__forwards)
* [Customizing Commands](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__custcmd)
* [Adding commands through the command factory](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__factory)
* [API enhancements](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__api_enh)
* [Accessing portlet objects](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__access_specs)
* [Sending a message](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__struts_msg)

**Comparison of servlets and portlets**

There are two main differences between the portlet and servlet environment that affects a Struts application in WebSphere Portal.

1. **Action processing and rendering**

All servlet processing occurs during the service() method. The Struts rendering of the page is usually immediately preceded by action processing; they are essentially part of one step. The request and response object are passed to the service() method, which writes the resulting output to the response object. The servlet-based Struts RequestProcessor is designed to complete the Struts action processing and eventually forward, based on the URI, to a JSP or another Struts action.

Portlet processing, however, is implemented in two phases, an event phase and a render phase. Action processing is performed prior to rendering the display view. Only the request object is passed to the portlet during the event phase. When a Struts application is migrated to the portlet environment, some of the information that was available during the event phase, namely the request parameters, is no longer available during the render phase. Additionally, since rendering methods, such as doView(), can be called when the portlet page is refreshed without a new event occurring for that portlet, all information required to render the page must be available every time that method is called.

**Note:** The struts action is not invoked when moving between modes (view, edit, and configure). Only the service method is called (doView, doEdit) when switching modes. To cause the struts action to fire, place a call to that action in these methods.

The Struts Portlet Framework provides a RequestProcessor that creates an IViewCommand. IViewCommand encapsulates the information so that the command object can be rendered at a later time, during the rendering method. IViewCommand and IViewCommandFactory are discussed in [Saving information for rendering the view](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__render_proc).

1. **URI construction**

URIs are constructed differently for portlets than for servlets. The portlet creates the URI programmatically using the PortletResponse object. The Struts Portlet Framework has modified the tags in Struts so that they create portal links. The Struts link tags behave the same as they do in the servlet environment, but the URL is a portlet URL and the Struts URL is passed as a parameter on the URL. See [Creating portlet URIs](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrjsps.html) for more information.

For general information about portlet action processing, see [Portlet events](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsptevents.html).

**Saving information for rendering the view**

A command pattern can be used to encapsulate the rendering of the view, and the information required during this rendering (See *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides for more information about the command design pattern). The pattern is implemented using the IViewCommand interface.

During the action processing, information needed to render the view needs to be saved, including the path to the page to be displayed. Also, for a JSP in a Struts application, the associated ActionForm is normally needed in order to fill in the page. So, any ActionForm attributes need to be saved along with the path to the page to render. The attributes that are saved with a command should be limited because, since the action processing and view rendering steps involve separate requests, the command is saved in session. Since the session information may need to be serialized in a high availability environment, it is important to minimize the amount of data stored in the session wherever possible. Thus, the ActionForm is typically needed, this is saved by default.

In doView() processing, nothing needs to be known about the IViewCommand except that it should be executed. The execute method of the IViewCommand receives the request and response objects as parameters. As part of the processing, the previously saved attributes are populated into the request object that is passed in on the execute method.

Also, during the render phase, an execution context object (ViewCommandExecutionContext) is passed in on the execute call, which provides additional objects to help with the actual execution. You can save additional information with the view command and provide additional objects and information in the command execution context.

**Note:** For more information about IViewCommand, refer to the Javadoc for the *com.ibm.portal.struts.command* package. The product Javadoc is located at [*PortalServer\_root*](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/reference/wpsdirstr.html#wpsdirstr__wp_root)/doc/Javadoc/api\_docs/ directory. Also see [Using the command factory](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__factory).

**Updating ActionForm in the render phase**

Some Struts applications may need to refresh the beans for the render phase every time the portlet renders. If the user is interacting with the portlet and clicking action links then the portlet will have an action phase and then followed by a render phase. The typical scenario is the case where an action phase does not precede the render phase. For example, if a user is interacting with another portlet on the page, then only that portlet has an action phase. The other portlets on the page may need to refresh the data to be rendered. The Struts action that implements the IStrutsPrepareRender action can read data from the model and update the ActionForm bean associated with the JSP through the information available through the ActionMapping. The page controller action is executed in the render phase of the portal server and therefore cannot change portlet state or write to the model. It also cannot interact with other portlets using Property Broker or portlet messaging. The collaboration between portlets must occur during the action phase of the portal server.

The IStrutsPrepareRender interface is way to indicate that a Struts action should be called in the render phase of portal, for the specific task of refreshing the data to be displayed. When the Struts Portlet Framework Request Processor detects a Struts Action that implements the IStrutsPrepareRender in the action phase, the request processing is stopped and the ViewCommandFactory is called to create a WpsStrutsViewActionCommand object. The WpsStrutsViewActionCommand object is then executed in the render phase of the portal, at which time the request processor will be called again to execute the Struts action. A Struts action implementing IStrutsPrepareRender may forward to another Struts action that implements the Prepare-Render interface or to the JSP that renders the view. To see an example of how to use the IStrutsPrepareRender interface, see the Stock Quote portlet or Clock portlet shipped with WebSphere Portal.

**Clock Portlet Sample:**

The Clock portlet is a simple portlet that displays the system time. The typical implementation of this portlet would refresh the time when the refresh button is clicked, but would display the saved time if the clock portlet was refreshed without the refresh button clicked. This example demonstrates how this portlet can be implemented so that it refreshes the time every time the portlet is rendered, regardless if the refresh button is clicked. The GetTimeAction Struts action simply performs the task of obtaining the current time and updating the ActionForm bean. This action is a candidate for a Prepared Render Struts action because it does not change the portlet state, only refreshes the Struts ActionForm bean. A Struts Action is specified to be a Prepare Render Struts Action by implementing the IStrutsPrepareRender interface. This allows the portlet to properly refresh the Struts ActionForm bean with the current time while in the render phase. The sample for the legacy container is named SPFLegacyClock.war, and the sample for the standard container is called SPFStandardClock.war. Both clock samples are shipped with WebSphere Portal.

public class GetTimeAction extends Action

implements IStrutsPrepareRender

{

public ActionForward execute(ActionMapping mapping, ActionForm form,

HttpServletRequest request,

HttpServletResponse response)

throws Exception

{

ActionForward forward = mapping.findForward("success");

if (form instanceof ClockBean) {

ClockBean clock = (ClockBean) form;

Date current = new Date();

String timeString =

DateFormat.getTimeInstance(DateFormat.LONG).format (current);

clock.setTime( timeString );

}

return forward;

}

}

**Stock Portlet Sample:**

The Stock Quote portlet is a Struts application that allows configuring companies in the edit portlet mode, and then the Stock Quotes for the portlet are displayed in the view mode. The quotes are supplied by a StockService, which returns random numbers, but the StockQuoteService could be replaced by a real service. The Stock Quote sample also uses a Struts Action that implements the IStrutsPrerpareRender so the stock quotes are refreshed even if a request phase does not precede the render phase.

public class RefreshAction extends Action implements IStrutsPrepareRender

{

// ------------------------------------------------ Instance Variables

/\*\*

\* The Log instance for this application.

\*/

private Log log =

LogFactory.getLog(this.getClass());

// ------------------------------------------------- Public Methods

/\*\*

\* This action will refresh the stock quotes and then forwards to an

\* ActionForward for displaying the quotes.

\*

\* @param mapping The ActionMapping used to select this instance

\* @param form The optional ActionForm bean for this request (if any)

\* @param request The HTTP request we are processing

\* @param response The HTTP response we are creating

\*

\* @exception Exception if the application business logic throws

\* an exception

\*/

public ActionForward execute(ActionMapping mapping,

ActionForm form,

HttpServletRequest request,

HttpServletResponse response)

throws Exception

{

HttpSession httpSession = request.getSession();

//---------------------------------------------------------

// Get the SubscribedNewsgroupsBean. This should have been

// put in the session earlier

//---------------------------------------------------------

SubscribedCompaniesBean subscribedCompaniesBean =

( SubscribedCompaniesBean )

httpSession.getAttribute( Constants.SUBSCRIBED\_COMPANIES );

ActionForward actionForward = mapping.findForward("welcome");

if ( subscribedCompaniesBean != null )

{

log.debug("Refresh quotes");

Collection companies = subscribedCompaniesBean.getCompanies();

if ( companies != null )

{

StockQuoteService stockQuoteService =

new StockQuoteService();

Iterator iterator = companies.iterator();

while ( iterator.hasNext() )

{

CompanyBean companyBean = ( CompanyBean ) iterator.next();

try

{

log.debug("Get quote for company " +

companyBean.getSymbol() );

float quote =

stockQuoteService.getQuote(companyBean.getSymbol());

companyBean.setQuote( Float.toString( quote) );

}

catch ( Exception ex )

{

companyBean.setQuote( "---");

}

subscribedCompaniesBean.addCompany(companyBean.getName(),

companyBean );

}

}

}

return actionForward;

}

}

**Response object**

During portlet action processing the response object is not available. The methods called during Struts processing expect both a request and response. To address the need for a response object, the Struts Portlet framework creates a temporary one. Other than for calling [sendError()](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__senderr), you should not write to the response object. Instead your Action should return an ActionForward object and let the Struts RequestProcessor complete the doForward set.

**sendError() processing**

The typical Struts application has no need to access the response object during the Action processing. However, when an application checks and finds some state information invalid during action processing, it is not unusual for a Struts application to report the error using the response.sendError() method. The Struts Portlet Framework intercepts the calls to sendError() and saves the error information in the session. During the later view render phase, this error information is found, and the error information is displayed in lieu of displaying other content for the portlet.

**Overriding Error Response Formatter**

The Struts Portlet Framework provides default formatting for error responses. For standard portlets, the default error response formatter is com.ibm.portal.struts.plugins.DefaultErrorResponseFormatter. For IBM portlets, it is com.ibm.wps.portlets.struts.plugins.DefaultErrorResponseFormatter. This formatter can be overridden from within a Struts Plug-in. You can provide your own Plug-in which sets your own subclass of DefaultErrorResponseFormatter. The SPFLegacyPlugins.war and SPFStandardPlugins.war are examples of how to implement a custom error response formatter.

**Saving information in a bean**

A typical Struts application does not write directly to the response object and instead forwards to a JSP. The JSPs should not need access to the original event information, request parameters, which initially led to the current rendering. Instead, action processing should store information in a bean from which the JSP extracts the information. As long as the bean is available during the rendering, the rendering can be repeated. Since beans, such as ActionForm, are usually stored in the original request attributes, the request attributes from the original request processing must be saved in the iViewCommand and made available when doView() is invoked.

**Forwards and redirects**

Although you can write portlets using Struts, there are some aspects of the servlet environment that you cannot do in a Struts portlet. For example, Struts provides support for redirects and forwards to other servlets. These are provided because they are functions generally available to servlets. However, these capabilities are not available in portlets because WebSphere Portal does not support forwards or redirects.

The Struts Action object returns an ActionForward object, which contains a path to be either forwarded or redirected to. WebSphere Portal does not support forward operations because the response object has already been committed. If the path is not an action, include the page using the PortletContext.include() method. If the path is an action, then a forward to the Action is simulated. Forwards to other actions can be handled by recursively sending the new Action URI through the Struts base processing of an Action. Redirects are treated in the same way as a forward. This can be implemented using PortletApiUtils class. The example below demonstrates how to implement a forward to a Struts Action from an Action or a custom tag.

PortletApiUtils portletUtils = PortletApiUtils.getInstance();

if (portletUtils != null)

{

portletUtils.forward( page, request );

}

else

{

pageContext.forward( page, request );

}

The PortletApiUtils.getInstance() method is called to see if a PortletApiUtils instance is available. If a non null value is returned, then the execution is inside of WebSphere Portal. In this case, the portletApiUtils.foward() method is used instead of the PageContext.forward().

The issue with forwards affects tag handlers as well. In tag handlers, it is possible to access the PageContext object and invoke the forward method. An alternative method to the PageContext.forward() is available via the *PortletApiUtils* class that will provide a mechanism to emulate the forward functionality.

The tags in the PortalStrutsExample were modified to use the *PortletApiUtil* class instead of the PageContext.forward. The tags shipped with the Struts example are CheckLogonTag, LinkSubscriptionTag, and LinkUserTag. Here is a code sample of the change.

PortletApiUtils portletUtils = PortletApiUtils.getInstance();

if (portletUtils != null)

{

// when run in a portlet use this execution path.

Object pResponse = portletUtils.getPortletResponse( request );

Object portletURI =

portletUtils.createPortletURIWithStrutsURL(pResponse, url.toString() );

// don't need to call response.encodeURL,

// the portletURI.toString takes care of that.

results.append(portletURI.toString() );

}

else

{

// when run as a servlet, execute this path

results.append(response.encodeURL(url.toString()));

}

If a forward is required in a JSP, then the logic forward tag is the suggested solution. The forward tag has been modified to use the PortletApiUtils forward implementation, and is the preferred method for a forward from a JSP. The PortletApiUtils can be obtained in a JSP through java code, but obtaining the Struts ModuleConfig to prefix the path is problematic. The logic forward tag handles these issues.

For further information, see [Creating link tags](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrlink.html).

**Customizing Commands**

The section [Comparison of servlets and portlets](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__serv_compare) describes two phase processing of the portal server and the need for the IViewCommand objects that are created in the actionPerformed() phase. The Struts Portlet Framework ships several different commands that handle dynamic content, static content, XML data and error information from response.sendError(). This section describes some of the customizations that can be applied to a command, and also how to implement a command factory for creating new commands.

**Struts View command**

The Struts View command is the base class for commands in the Struts Portlet Framework.

* **Standard portlets:** com.ibm.portal.struts.command.StrutsViewCommand
* **IBM portlets:** com.ibm.wps.portlets.struts.WpsStrutsViewCommand

This class provides the foundation for saving the required information, including attributes, so the command can be rendered at a later time and multiple times. The Struts View command limits the number of attributes saved in the command object because these objects are saved in session. The command object needs to be available any time the portlet is refreshed so the command is saved in session. There is one command per portlet mode supported.

When the command object is created, Struts View command saves the request attributes that are configured to be saved. When the command is executed, it restores these attributes in the request object. An anticipated change to a command is the need for additional request attributes to be stored with the command. The Struts View command has two static methods that allow adding additional request attributes to save so this change can be made easily. The new attributes can be request attributes with a specific name, or attributes of a specific type.

If the rendering step requires additional attributes, then The Struts View command can be requested to save those attributes.

**Struts View Jsp command**

The Struts View Jsp command is the most commonly used command in the Struts Portlet Framework.

* **Standard portlets:** com.ibm.portal.struts.command.StrutsViewJspCommand
* **IBM portlets:** com.ibm.wps.portlets.struts.WpsStrutsViewJspCommand

The dynamic content rendered in the JSP most likely needs request attributes in the render step. The request attributes that are saved and made available in the render step are:

**Attributes with the name**

* Globals.MODULE\_KEY
* Globals.MESSAGES\_KEY
* Globals.ERROR\_KEY

**Attributes of type**

ActionForm

**Adding request attributes stored with a command**

The Struts application should call the Struts View command to add static methods before any IViewCommand objects are created. There are two ways to implement this customization of the Struts View command.

* Subclass WpsStrutsPortlet and implement the init() method. The WpsStrutsPortlet subclass then uses the following WpsStrutsViewCommand static methods.
* addAttributeNameToSave(String attributeName)

addAttributeTypeToSave(Class attributeType)

* Use a Struts plugin. The Struts Plugin is initialized when the configuration for the Struts module is read. The plugin can be used to add the attributes that are needed by a command. Here is an example plugin definition from the Struts configuration file.
* <plug-in
* className="com.ibm.struts.sample.plugins.ExampleAddAttributePlugin">

</plug-in>

Here is an example of the plugin's init() method:

/\*\*

\* Add attributes to save.

\*

\* @param config The ModuleConfig for our owning modules

\* @param servlet The ActionServlet

\*

\* @exception ServletException if we cannot configure

\* ourselves correctly

\*/

public void init(ActionServlet actionServlet, ModuleConfig config)

throws ServletException

{

// Call StrutsViewCommand.addAttributeNameToSave for each

// request attribute we wish to make available to the code

// that renders our page. We can either specify the name

// of the attribute or the class

// type of attributes to save.

StrutsViewCommand.addAttributeNameToSave("testAttribute");

}

**Accessing request attributes**

All request attributes that are set by the Struts action will be available when the *WpsStrutsViewCommand* is rendered the first time without the need to take these additional steps to save them. When a user interacts with a portlet, this defines a single request and the same request object is available to the actionPerformed() method, where the Struts action is executed, and the service() method, where the rendering occurs. If the portlet is requested to refresh the view at a later time, the attributes might not be available. The refresh may be because the user is interacting with another portlet and the request object will not have the expected attributes. If the additional attributes are required during the refresh at a later time, then *WpsStrutsViewCommand* should be modified to save these attributes as described above. Also, the rendering step should not rely directly on request parameters. Request parameters should be processed in the Action. However, request parameters can be saved automatically by Struts in the ActionForm. The ActionForm objects are available to the render phase.

**Adding commands through the command factory**

The Struts Portlet Framework uses the ViewCommandFactory object to create the IViewCommand objects for representing the view to render based on the given input information. The view command object typically contains the URL of the object (usually a JSP) creating the output for the portlet and other information needed for the rendering.

* For standard portlets, the default view command factory class is com.ibm.portal.struts.plugins.ViewCommandFactory.
* For IBM portlets, the default view command factory class is com.ibm.wps.portlets.struts.plugins.ViewCommandFactory.

The command object needs to contain the information to render the output properly. For a JSP within a Struts application, this is typically a form bean. It is possible that a particular portlet needs additional information in order to do the rendering. In such a case, you would need to add code to cause the additional information to be added to the command. The likely approach would be to override the saveAttributes() method in your subclass of the *IViewCommand* class. If the given URI (or perhaps type of URI) was one that would require additional information for rendering, then a new type of *WpsStrutsViewCommand* would need to be created which contained the needed information. The saveAttributes() method of this subclass of IViewCommand could then save the additional objects to be placed in the request object as attributes when the command was executed. To cause an existing request attribute to be saved, the saveAttribute method can be used to provide the request object and name of the attribute.

**Note:** The SPFLegacyPlugins.war and SPFStandardPlugins.war are examples of how to implement a custom ViewCommandFactory.

**Controlling where commands are stored using the Command Manager Factory**

The CommandManagerFactory allows a Struts application to specify where the IViewCommand objects are stored. The Command Manager is pluggable through the Command Manager Factory interface.

* **Standard portlet container:** The Command Factory plugin is typically configured in the portlet deployment descriptor.
* **IBM portlet container:** The Command Factory plugin is configured in the web deployment descriptor.

The default Command Manager Factory is the SessionCommandManagerFactory which stores the IViewCommands in the portlet session. The SessionCommandManagerFactory is the default command manger factory and recommended for portlets.

**Note:** The SPFLegacyCommandManger.war and SPFStandardCommandManager.war are examples of how to implement a custom CommandManagerFactory.

**API enhancements**

Ideally, portal-specific requirements should be minimized when writing Struts Applications. There are certain areas that need to be surfaced to the developer, like tag modifications and forwards. For those developing Struts applications specifically for the portlet environment, a class is provided to hide some of the servlet details that do not map well to execution in the portal server environment.

**StrutsAction**

The StrutsAction class provides an execute method that is passed portlet objects instead of servlet objects so a cast is not required to access portal-specific objects. The signatures for functions in the StrutsAction class are below:

**Standard portlets**

public ActionForward execute(ActionMapping mapping,

ActionForm form, PortletRequest request) throws Exception;

public ActionForward execute(ActionMapping mapping,

ActionForm form, ActionRequest request,

ActionResponse response) throws Exception;

public ActionForward execute(ActionMapping mapping,

ActionForm form, RenderRequest request,

RenderResponse response) throws Exception;

public void sendError(ActionRequest request,

int sc, String msg) throws IOException;

public void sendError(RenderRequest request,

int sc, String msg) throws IOException;

public void sendError(ActionRequest request, int sc )

throws IOException;

public void sendError(RenderRequest request,

int sc ) throws IOException;

**IBM portlets**

public ActionForward execute(ActionMapping mapping,

ActionForm form, PortletRequest request)

throws Exception;

public ActionForward execute(ActionMapping mapping,

ActionForm form, PortletRequest request,

PortletResponse response) throws Exception;

public void sendError(PortletRequest portletRequest,

int sc, String msg) throws IOException;

public void sendError(PortletRequest portletRequest,

int sc) throws IOException;

**LookupDispatchAction and DispatchAction**

The WpsLookupDispatchAction and WpsDispatchAction classes offer the same functionality as their Jakarta LookupDispatchAction and DispatchAction counterparts.

* **Standard portlets**
* com.ibm.portal.struts.action.LookupDispatchAction

com.ibm.portal.struts.action.DispatchAction

* **IBM portlets**
* com.ibm.wps.struts.action.WpsLookupDispatchAction

com.ibm.wps.struts.action.WpsDispatchAction

DispatchAction extends StrutsAction, so the StrutsAction methods can be used for the sendError() implementation.

In the following getKeyMethodMap() example, the add and delete buttons each have their own methods for processing.

protected Map getKeyMethodMap() {

Map map = new HashMap();

map.put("button.add", "add");

map.put("button.delete", "delete");

return map;

}

The LookupDispatchAction subclass implements the following two methods.

public ActionForward add(ActionMapping mapping,

ActionForm form, PortletRequest request )

throws IOException, ServletException {

// do add

return mapping.findForward("success");

}

public ActionForward delete(ActionMapping mapping,

ActionForm form, PortletRequest request )

throws IOException, ServletException {

\* // do delete

\* return mapping.findForward("success");

\* }

\*

The Struts Portlet Framework implementation of LookupDispatchAction is not passed a response object and the request object is a PortletRequest. These two classes are a convenience for applications that are intended for the portal environment only.

**Accessing portlet objects**

There will be situations when a Struts Action needs to access objects in the Portlet API. For example, the Struts Action might need to access the *PortletRequest* object to get to *PortletSettings* or *PortletPreferences*, depending on the container. The PortletApiUtils should be used to obtain the PortletRequest object from the HttpServletRequest object.

PortletApiUtils portletUtils = PortletApiUtils.getUtilsInstance();

if (portletUtils != null)

{

PortletRequest portletRequest = (PortletRequest)

portletUtils.getPortletRequest( request );

}

The Struts Action class also makes the ActionServlet class available. When the Struts application is authored with the Struts Portlet Framework, the ActionServlet class is actually the WPActionServlet. For more information see the Javadoc for:

* **Standard portlet container:** com.ibm.portal.struts.portlet.WpActionServlet class information.
* **IBM portlet container:** com.ibm.wps.portlet.struts.WpsActionServlet class information.

**Sending a message**

The Struts examples use the Property Broker for sending messages from one portlet to another. For more information about the property broker, see the property broker programming model section of the [Concepts of cooperative portlets](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsc2acnc.html) section.

The Struts Portlet Framework ships the SPFLegacyMultipleModules, and SPFStandardMultiplModules as examples of using the property broker and Struts Portlet Framework

**TOPIC 2:- Changes to Struts JSPs**

The JSPs for Struts applications in the portal environment have to be modified to adapt to the way the portal server expects portlet URIs to be created. There are some changes to the tag library for HTML markup and additional tag libraries have been added to support cHTML and WML markup.

* [Creating portlet URIs](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrjsps.html" \l "wpstrjsps__return)
  + [Examples](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrjsps.html#wpstrjsps__retrun_ex)
* [Style sheets](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrjsps.html#wpstrjsps__style)
* [Markup support](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrjsps.html#wpstrjsps__markup)
  + [Using the cHTML tags](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrjsps.html#wpstrjsps__chtml)
  + [Using the WML tags](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrjsps.html#wpstrjsps__wml)

**Creating portlet URIs**

The Struts application paths, both to actions and to pages, must be sent and retrieved using portlet URIs. Portlet URIs have a specific format. A special API is used to generate the URI and add the desired information to be passed to the portlet. If portlet URIs were not used, control would not get passed to the correct portlet. Thus, the portlet URIs must be used to get control passed to the correct portlet, with the additional path information needed by the Struts application made available. The Struts tags have been modified to automatically provide this needed functionality.

Struts Action mappings are defined in terms of paths. The name and location of page objects (for example, JSPs) are also defined using paths. Thus, although portlets have their own form of URI, it is still necessary to associate the Struts path with an action sent to a portlet and to retrieve that Struts path when the portlet action is handled. The Struts URI is passed as a parameter so the Struts request processor can process the action. If the link is not an action then the portlet processes the information and create the appropriate IViewCommand.

Typically a Struts application passes parameters on such a path using the query string on the HTTP URL. Often the actions containing these paths are generated from tags provided by Struts. The most obvious examples of these are the tags for the HTML elements <link> and <form>. For IBM Struts portlets, the *urlType* attribute must be included in the <html:form>, <html:link> and <html:rewrite> tags to support the different types of URIs provided by the IBM Portlet API. This attribute can take the following values:

* return - Creates a portlet return URL indicating the caller of the portlet. For example, this URL can be useful for creating a form action that can take a user back to view mode from edit mode.
* standard - Creates a standard portlet URL.

**Examples**

**<html:rewrite>**

<link rel="stylesheet" type="text/css"

href="<html:rewrite page='/assets/styles/base.css'/>">

**<html:form> (IBM portlets)**

<html:form action="/logon" focus="username" urltype="return">

...

</html:form>

**<html:form> (Standard portlets)**

<html:form action="/saveConfiguration.do" portletMode="view">

**<html:link> (IBM portlets)**

<html:link page="/tour.do" urltype="standard">

<bean:message key="index.tour"/>

</html:link>

**Style sheets**

Many existing Struts application use the rewrite tag to create a link element for a cascading style sheet. This is not the documented intention of the rewrite tag, which is supposed to create the same path as the link tag without the <a> element. Since the Struts Portlet Framework had to modify how links are created, the rewrite tag required some customizations to be used to create link elements for style sheets. The rewrite tag will create the same path as the link tag, except when the page or forward reference is to a CSS file. In the case where a CSS file is referenced, the rewrite tag will use the Jakarta Struts implementation, which results in a path to the CSS file. Here are examples of how to create link elements for style sheets using the Struts Portlet Framework.

**Using a forward**

<link rel="stylesheet" type="text/css"

href="<html:rewrite forward='baseStyle'/>">

**Using a page**

<link rel="stylesheet" type="text/css"

href="<html:rewrite page='/basestyle.css'/>">

**Using the portlet tags**

<%@ taglib uri="/WEB-INF/tld/portlet.tld" prefix="api" %>

<api:init />

<link rel="stylesheet" type="text/css"

href="<%= portletResponse.encodeURL( basestyle.css ) %>">

**Markup support**

The Struts tag library has been modified to support the additional markup languages supported by WebSphere Portal. For HTML, the tags that create links have been modified to support portlet URIs. See [Creating portlet URIs](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrjsps.html#wpstrjsps__return) for details. Also, be aware of the restrictions for HTML output described in [Markup guidelines](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpspar.html).

There might be a cases where the JSPs for a Struts application need to run in both the servlet and portlet environment. For this reason, page level tags are implemented in tag libraries. The Struts application can use them in its JSPs, but the tags will not generate markup when executed within WebSphere Portal.

You should also refrain from setting color, and fonts. The portal server supports skins and themes that give the page a consistent look and feel. The JSP should be authored so it adheres to the conventions of the theme by using the appropriate style sheet.

**Using the cHTML tags**

The use of the cHTML tags is similar to the use of the HTML tags. The name of the cHTML tag library file is struts-chtml.tld. The following is an example of the cHTML taglib definition.

<%@ taglib uri="/WEB-INF/struts-chtml.tld" prefix="chtml" %>

**Using the WML tags**

The WML tags are a new addition for creating a user experience in WAP devices. The use of the WML tags is similar to the use of the Struts HTML tags. Use the following directive to make these tags available to a JSP.

<%@ taglib uri="/WEB-INF/struts-wml.tld" prefix="wml" %>

The use of the WML tags provided with this distribution is similar to the use of Struts HTML tags. The name of the WML tag library file is struts-wml.tld. The following is an example of the WML version of index.jsp .

<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>

<%@ taglib uri="/WEB-INF/struts-wml.tld" prefix="wml" %>

<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>

<p>

<wml:link page="/editRegistration.do?action=Create">

<bean:message key="index.registration"/>

</wml:link>

<br/>

<wml:link page="/logon.jsp">

<bean:message key="index.logon"/>

</wml:link>

<br/>

<wml:link page="/tour.do">

<bean:message key="index.tour"/>

</wml:link>

</p>

See [General tips for portlet output](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpspar.html) and the [WML markup guide](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpspar.html) for general guidelines about providing WML markup.

WML does not have a <form> element like HTML. Struts, however, uses the <form> tag for the scoping of parameters and supporting form beans. For that reason, the WML implementation also includes a <form> tag to support some of the Struts features in WML. The <form> tag in the WML taglib takes an action as an attribute. The following is an example of a form in WML.

*Figure 1. Example form for WML*

<wml:form action="/logon">

<do type="options" label="send">

<wml:go method="post">

<postfield name="username" value="$username"/>

<postfield name="password" value="$password"/>

</wml:go>

</do>

<bean:message key="prompt.username"/><wml:text property="username"/>

<bean:message key="prompt.password"/>

<wml:password property="password" size="16" maxlength="16"/>

</p>

</wml:form>

The JSPs for the HTML Struts example have been modified for the WML markup. This example also demonstrates the use of the WML taglib. These JSPs can be found in the /wml/view directory of the expanded PortalStrutsExample.war file.

The tags in the Struts portlet WML tag library can throw a JspException at run time. An error page can be declared using the <%@ page %> directive to detect the exception and supply information about the error condition. The actual exception is passed as an attribute in the request object with the key org.apache.struts.action.EXCEPTION.

The following is a brief description of each tag. See [Detailed descriptions of the Struts WML tags](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html) for more information.

| *Table 1. Summary of WML tags for Struts portlets* | |
| --- | --- |
| **Tag** | **Description** |
| [<wml:cancel/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html#wpsstrtags__wml_cancel) | Renders a WML <postfield> element with a value of cancel. |
| [<wml:card/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html#wpsstrtags__wml_card) | Renders a WML card element. |
| [<wml:errors/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html#wpsstrtags__wml_errors) | Retrieves the set of error messages from the request object with the default key of Action.ERROR\_KEY or the value specified by attribute name. |
| [<wml:form/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html#wpsstrtags__wml_form) | Does not render any markup, but it is used to scope beans and transactions. |
| [<wml:go/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html#wpsstrtags__wml_go) | Renders a WML <go> element. |
| [<wml:head/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html#wpsstrtags__wml_head) | Renders a WML <head> element with language attributes extracted from the user's current Locale object, if there is one. |
| [<wml:link/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html#wpsstrtags__wml_link) | Renders a WML <a> element as a hyperlink to the specified URL. |
| [<wml:option/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html#wpsstrtags__wml_option) | Renders a WML <option> element, representing one of the choices for an enclosing <select> element. |
| [<wml:options/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html#wpsstrtags__wml_options) | Renders a set of WML <option> elements, representing possible choices for a <select> element. |
| [<wml:password/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html#wpsstrtags__wml_password) | Renders a WML <input> element of type password, populated from the specified value or the specified property of the bean associated with our current form. |
| [<wml:postfield/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html#wpsstrtags__wml_postfield) | Renders a WML <postfield> element. |
| [<wml:rewrite/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html#wpsstrtags__wml_rewrite) | Renders a request URI based on exactly the same rules as the link tag does, but without creating the <a> hyperlink. |
| [<wml:select/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html#wpsstrtags__wml_select) | Renders a WML <select> element, associated with a bean property specified by our attributes. |
| [<wml:text/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html#wpsstrtags__wml_text) | Renders a WML <input> element of type text, populated from the specified value or the specified property of the bean associated with our current form. |
| [<wml:wml/>](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsstrtags.html#wpsstrtags__wml_wml) | Renders a <wml> element. |

**TOPIC 3:- Changes to configuration files**

Struts Portlet Framework specific init parameters have been added that allows you to customize the Struts application for the portal environment. The portlet and web deployment descriptors require configurations specific to the Struts Portlet Framework. In addition, changes to the Struts configuration file must be made to specify a portal specific request processor as the controller.

* [Configuring the Struts application for the IBM container](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html#wpstrconfig__web_xml_changes)
* [Configuring the Struts application for the standard portlet container](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html#wpstrconfig__struts_changes)
* [Setting an initial view](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html#wpstrconfig__welcome-file-list)
* [Using modes](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html#wpstrconfig__modes_struts)
* [Support for devices](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html#wpstrconfig__struts_devices)
* [Internationalization support](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html#wpstrconfig__i18n)
* [WML support](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html#wpstrconfig__wml_webxml)
* [Logging](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html#wpstrconfig__logging)
* [Changes to the Struts configuration file](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html#wpstrconfig__struts_cfg)
* [Tiles support](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html#wpstrconfig__tiles)

**Configuring the Struts application for the IBM container**

Most of the configuration for a Struts application is in the Web deployment descriptor file (web.xml). The servlet class that is specified in the Web deployment descriptor should be the portlet servlet. The class name is WpsStrutsPortlet. Here is an example definition.

<servlet id="Servlet\_1">

<servlet-name>MyStrutsApp</servlet-name>

<servlet-class>com.ibm.wps.portlets.struts.WpsStrutsPortlet

</servlet-class>

...

</servlet>

When sub-classing *WpsStrutsPortlet*, you should designate the subclass name for <servlet-name/>. The servlet *id* should be suffixed with a unique ID to prevent conflicts with other portlets. See [Deployment descriptors](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsptdesc.html) for more information.

**Initialization parameters**

The following is a list of names of the initialization parameters that can be configured in the Web deployment descriptor for Struts portlets (using the <init-param> element).

**CommandManagerPlugin**

This configuration parameter will allow setting a pluggable CommandManager

**struts-servlet-mapping**

Identifies a Struts action mapping so that paths that should be treated as Struts actions can be recognized. See [Servlet mapping](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html#wpstrconfig__serv_map) for further details.

**ModuleSearchPath**

Indicates the search path for Struts modules, default value is the markupName and mode. See [Using modes](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html#wpstrconfig__modes_struts) for more information.

**IncludesSearchPath**

Indicates a more fine-tuned search path for including a file. If this value is not specified then PortletContext.include() is used instead.

**NamescopeServletContext**

This allows the WpsStrutsPortlet to be defined in the web deployment descriptor more than once by namescoping the parameters in the servlet context.

**NamescopeFormName**

The default behavior is for the Struts form tag to namescope the name of the portlet. Setting this parameter to false will allow the behavior to be overridden.

**UseGroupsForAccess**

Configures the Struts portlet to use portal server group names as if they were role names. Struts uses roles to determine access to an Action. Set this parameter to **true** to configure the Struts application to use the portal's group names as if they were role names to emulate role-based access control.

**UsePortalsLocale**

This parameter will allow a Struts application to override default behavior of using the portal's locale. If this parameter is set to false the Struts application can use the Request Processor's processLocale method for controlling locale or set the locale in a Struts action; otherwise, it uses the portal's locale and also supports dynamic changes. The default value is true.

**WelcomeFileSearchPath**

This configuration parameter will allow customizing the initial view page search.

**Servlet mapping**

In a portlet, you need to use a servlet URL mapping for your portlet. So, one would typically use a path prefix servlet mapping similar to the following.

<servlet-mapping id="ServletMapping\_1">

<servlet-name>MyStrutsApp</servlet-name>

<url-pattern>/Struts/\*</url-pattern>

</servlet-mapping>

The typical Struts application uses an extension mapping. The Struts example uses extension mapping and it is expected that most Struts applications use the extension mapping "\*.do". In the normal Struts case, this means that URIs ending in "\*.do" are directed to the Struts servlet. Remember that Struts provides a single servlet that you can use (or subclass) for all of your Struts applications. When you wish to cause control to be given to a Struts Action when a link is selected or form submitted, use the extension from the extension mapping (e.g. "\*.do") on the URI so that the processing is routed to the Struts *ActionServlet*. Other URIs, such as those for JSPs, do not have this extension and, thus, would not be routed through the Action Servlet for processing.

A Struts application, which is itself a servlet application, also requires a servlet mapping. Although our Struts portlet already has a servlet mapping as shown above, you still need to identify what to use as a Struts action mapping so that paths that should be treated as Struts actions can be recognized. To specify the pseudo servlet mapping used by Struts, specify it as a servlet initialization parameter in the web.xml file, as follows:

<init-param>

<param-name>struts-servlet-mapping</param-name>

<param-value>\*.do</param-value>

</init-param>

The Struts Portlet Framework supports the same servlet mappings modes as the servlet-based Struts application. However, the current Struts implementation only supports extension mapping for the module implementation. Since the portal server implementation uses modules for the mode and markup support, it is strongly recommended that the extension mapping is used.

**Configuring the Struts application for the standard portlet container**

In the standard container the Struts application is configured in the portlet deployment descriptor. The portlet class and init parameters are specified in the portlet deployment description. In the standard container, the only configuration in the web deployment descriptor is the welcome file list and the tag library (taglib) elements. The portlet class that is specified in the portlet deployment descriptor must be the com.ibm.portal.struts.portlet.StrutsPortlet class or subclass.

<portlet>

<portlet-class>com.ibm.portal.struts.portlet.StrutsPortlet</portlet-class>

</portlet>

If sub-classing the StrutsPortlet, the subclass class name should be designated as the <portlet-class/>.

**Initialization parameters**

The following is a list of names of the initialization parameters that can be configured in the Web deployment descriptor for Struts portlets (using the <init-param> element):

**IncludesSearchPath**

Indicates a more fine-tuned search path for including a file. If this value is not specified then PortletContext.include() is used instead.

**ModuleSearchPath**

Indicates the search path for Struts modules, default value is the markupName and mode. See Support for multiple Struts applications for more information

**NamescopeFormName**

The default behavior is for the Struts form tag to namescope the name of the portlet. Setting this parameter to false will allow the behavior to be overridden.

**NamescopeServletContext**

This allows the WpsStrutsPortlet to be defined in the web deployment descriptor more than once by namescoping the parameters in the servlet context.

**struts-servlet-mapping**

Identifies a Struts action mapping so that paths that should be treated as Struts actions can be recognized. See Servlet mapping for further details.

**UseGroupsForAccess**

Configures the Struts portlet to use portal server group names as if they were role names. Struts uses roles to determine access to an Action. However, WebSphere Portal V 4.2 does not support roles. Set this parameter to true to configure the Struts application to use the portal's group names as if they were role names to emulate role-based access control.

**UsePortalsLocale**

This parameter will allow a Struts application to override default behavior of using the portal's locale. If this parameter is set to false, the Struts application can use the Request Processor's processLocale method for controlling locale or set the locale in a Struts action. Otherwise, it uses the portal's locale and also supports dynamic changes. The default value is true.

**WelcomeFileSearchPath**

This configuration parameter will allow customizing the initial view page search.

**Setting an initial view**

The Struts application needs to specify what the initial view is for the application. The typical way that the initial screen is specified is through the welcome file list. The Struts Portlet Framework supports the welcome file list configuration, and has some additional features. The servlet implementation of the welcome file list only supports JSP and HTML pages. The Struts Portlet Framework supports a JSP or HTML welcome file, and also allows a Struts action to be specified as the welcome file. See Example 1 for implementation details.

The alternate method allows Welcome files to be configured on a per portlet basis as a configuration parameter in the portlet deployment descriptor portlet.xml. The default search will look for the initial view file per portlet mode. The other feature in the Struts Portlet Framework is the support for modules. A unique welcome file should be configured for each module by specifying the application prefix. For example, to support the view mode for the markup WML, a welcome file wml/view can be specified. See Example 2 for implementation details.

*Figure 1. Example 1: Welcome file list that is specified in the file web.xml*

<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

<welcome-file>help/index.jsp</welcome-file>

<welcome-file>wml/view/index.jsp</welcome-file>

</welcome-file-list>

*Figure 2. Example 2: Welcome file list that is specified on a per portlet basis (standard portlets)*

<portlet-preferences>

<preference>

<name>viewMode.page</name>

<value>portlet1/welcome.jsp</value>

</preference>

<preference>

<name>editMode.page</name>

<value>html/edit/index.jsp</value>

</preference>

</portlet-preferences>

*Figure 3. Example 3: Welcome file list that is specified on a per portlet basis (IBM)*

<config-param>

<param-name>viewMode.page</param-name>

<param-value>portlet1/welcome.jsp</param-value>

</config-param>

<config-param>

<param-name>editMode.page</param-name>

<param-value>portlet1/edit.jsp</param-value>

</config-param>

**Note:**

* A welcome file entry should be configured for each module. The welcome file entry is the initial screen shown when first switching to that module.
* The default Struts configuration, which has a "" prefix, is used any time a module prefix is not found.

**Using modes**

The Struts Portlet Framework is set up to use the mode and markup to find the Struts configuration. However, if you do not want to use a Struts application in one of the modes, the Struts portlet (Wps)StrutsPortlet should be subclassed. The appropriate service() method should be overridden to implement the desired behavior. For example, if the developer wants a help mode that does not use a Struts application, then the doHelp() method should be implemented in a subclass.

Portlet modes can be supported with the Struts Portlet framework. The Struts Portlet Framework stores the IViewCommand objects on a per mode basis. That means that an application that supports view and edit mode will have an IViewCommand object for view and edit mode. The welcome file list is typically used for the initial page when entering a mode.

The SPFLegacyMultipleModules.war example for the IBM container and the SPFStandardMultipleModules.war example for the standard portlet container demonstrate support for edit mode. The Module 1 portlet implements an edit mode wizard that steps through three screens. The example also demonstrates how to use a StrutsActionForward so that the IViewCommand object for the last page of the wizard can be marked for removal on a mode change. This allows the IViewCommand object for edit mode to be deleted when the user switches from edit mode back to view. The anticipated use is that the user is entering data in edit mode and the configuration data is submitted to be stored. The last page displays that the data has been successfully stored. The user then leaves edit mode using the skin. If the user goes back into edit mode, they do not want to see the message that the data was successfully stored. Since the command was deleted, the welcome file list entry can be used to start the wizard again. This feature also reduces the number of attributes stored in session.

<action path="/nextEditPage2"

type="com.ibm.wps.struts.example.multipleapps.actions.ForwardAction">

<forward className="com.ibm.wps.struts.action.StrutsActionForward"

name="success"

path="/edit3.jsp">

<set-property property="removeOnModeChange"

value="true"/>

</forward>

</action>

The Struts example application demonstrates support for help mode. A Struts configuration has been added to the Web deployment descriptor, web.xml, to demonstrate help mode. This mode is entered when the user clicks the help button for the Struts example portlet. The following code fragment from the Web deployment descriptor demonstrates how to configure the struts-configuration to support help mode.

<init-param>

<param-name>config/html/help</param-name>

<param-value>/WEB-INF/html/help/struts-config.xml</param-value>

</init-param>

The search path set in the Web deployment descriptor includes mode, which enables a separate Struts configuration for a mode.

<init-param>

<param-name>ModuleSearchPath</param-name>

<param-value>markupName, porletMode</param-value>

</init-param>

The welcome file list defines the welcome file for the help mode, as follows:

<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

**<welcome-file>html/help/index.jsp</welcome-file>**

<welcome-file>wml/view/index.jsp</welcome-file>

</welcome-file-list>

**Support for devices**

The Struts Portlet Framework's support of devices was designed to be flexible. The way the configuration and JSP files are placed in the directory structure should be tolerant of the devices that need to be supported. Some applications may just support HTML browsers and only need a single Struts configuration file. Other applications may support multiple markups and locales but do not need to know specifics about the device itself, while other implementations may need unique implementations for each of the supported devices.

**Note:** This support requires extension mapping. The prefix mapping only supports only a single configuration file.

WebSphere Portal is designed to allow portlets to access information about the client device, such as locale and markup language supported, using the *Client* interface. The Struts support for modes and markups has been implemented by allowing a configuration parameter to specify the fields in the client object that are interesting. The Client object is treated like a bean, where the field value corresponds to a get method in the Client object. This allows developers to specify the fields in the Client object that are necessary to their device support.

**Note:** The standard portlet container does not have a Client object but the following fields are supported:

* markupName
* locale
* portletMode

There are two paths that can be specified for supporting modes and device types (clients). The first path is the search path, and that path is used to specify the module to use. The search path locates the appropriate Struts configuration to use and is the prefix that is used as the base directory for the necessary JSPs. The default search path is *markupName, portletMode*. This will search for a configuration based on the current markup and the portlet mode.

<init-param>

<param-name>ModuleSearchPath</param-name>

<param-value>markupName, portletMode</param-value>

</init-param>

For example, if a desktop browser is used and the user is in view mode, then the search path would be:

* html/view
* html
* "" ( default configuration )

And if the Client object specified "wml" as the markupName then the search path would be:

* wml/view
* wml
* "" ( default configuration )

The search path is checked against all of the prefixes for the ModuleConfig objects for a match. The modules are configured in the Web deployment descriptor, web.xml, by specifying a prefix for a Struts configuration.

For example, a module for the WML markup in view mode would be configured by adding a configuration to the web deployment descriptor like:

<init-param>

<param-name>config/wml/view</param-name>

<param-value>/WEB-INF/wml/view/struts-config.xml</param-value>

</init-param>

If the wml/view module is selected, the corresponding JSPs are searched for under the wml/view directory of the Struts application WAR file.

This is a good start for separating the files for different markups and modes, but this granularity is probably not enough to deal with all available devices, or even locales. You can modify this search path to have a finer granularity. For example, you could create a unique module for every phone that they support.

The search path of markupName, portletMode, locale, manufacturer, model, and version could be used. This offers a very fine-grained deployment of the JSPs. The problem with this approach is that the majority of the JSPs would probably be the same for each of the devices. There are probably a few that need to be modified for each device, but this approach does not allow sharing.

This is why two paths are offered, search and include. The search path is used to find the module. This would be for applications that can use a common configuration, share actions and some of the JSPs. The include path can then be used to look for specific implementations of the JSP and if not found, fall back to a common file.

Take the following include path for including JSPs that are dependent on the device.

<init-param>

<param-name>IncludesSearchPath</param-name>

<param-value>manufacturer, model, version</param-value>

</init-param>

This would search for the file relative to the module and then take the requested fields into account. So if the search path found the wml/view module and the file index.jsp is requested, then the above search path for an ACME T400 would be as follows:

* wml/view/Acme/T400/index.jsp
* wml/view/Acme/index.jsp
* wml/view/index.jsp

Note that version did not show up in the search. That is because the default client configuration for the T400 did not have one defined. If one was configured at a later time, then it would be used.

Portal aggregation is flexible in how the differences between devices are supported. The common features are supported through the search path; all WML devices in view mode in a locale. The subtitle differences between devices would be supported through the file search of the include path. The common versions of the JSPs would be at a root level, and specific files would be at lower levels.

The Struts example ships with several tags, two of which required a minor change to support modules. The LinkSubscriptionTag and LinkUserTag tags had to be modified so that the current ModuleConfig's prefix had to be added to the computed URL.

**Internationalization support**

The Struts Portlet Framework supports i18n through the use of resource bundles. Each Struts application can specify a message resource. The Java resource bundle implementation determines the resource bundle to use based on the locale and variant. The Struts application typically uses the message tag to include translated text from the application-supplied resource bundle.

<init-param>

<param-name>IncludesSearchPath</param-name>

<param-value>locale</param-value>

</init-param>

The Struts framework was designed to give the application control over locale by using a locale object that is set in session with a known key. The Struts RequestProcessor sets the locale object for each user during the processLocale() method. The processLocale() method sets the locale object using the key Globals.LOCALE\_KEY if the feature is enabled through the Struts configuration file and the attribute has not already been set in session.

A setting on the controller element that is available in the Struts configuration file controls the locale support in Struts. The support can be turned off, which will stop the locale processing in the Request Processor processLocale() method.

WebSphere Portal supports dynamic changes to the locale. Portlets should use the locale object that is obtained through the PortletRequest object for the locale setting. The Struts Portlet Framework sets the locale object in the session using the key *Globals.LOCALE\_KEY* during the actionPerformed() and service() methods. The support is implemented by calling the (Wps)StrutsPortlet processLocale() method, which sets the locale object if one is not found or the locale is different than the one already in session. This assures that a portlet written using the Struts Portlet Framework uses the correct locale.

Locale support can also be customized. The (Wps)StrutsPortlet processLocale() method can be overridden for a customized implementation. The invocation of (Wps)StrutsPortlet processLocale() can also be stopped by adding the following init-parameter to the Web deployment descriptor for the application.

<init-param>

<param-name>UsePortalsLocale</param-name>

<param-value>false</param-value>

</init-param>

However, in most cases the portlet should use the default implementation.

**Using the IBM container**

WebSphere Portal also provides an implementation to find localized JSPs using the include() method on PortletContext. PortletContext inserts the locale into the path to locate the locale-specific JSP. For more information, see Generating markup for multiple devices, markups, and languages.

The Struts Application can be deployed to use the portal server's aggregation method to find localized JSPs, instead of a resource bundles for each locale. The Struts Portlet Framework uses the PortletContext.include() on the legacy container to include JSP pages, so both methods of localization are supported.

PortletContext.include() searches for locale-specific files by inserting the markup and locale name in the path. Depending on how the file structure has been set up, this may not be desirable. This behavior can be configured through an initialization parameter in the web deployment descriptor. An include search path can be specified that controls how the includes are located. For example, the default implementation would be markup, locale but can be changed to just locale if desired. The mode and markup support may have been already configured to include markup, so this configuration allows the markup to not be needed a second time.

**WML support**

Because WebSphere Portal supports pervasive devices, such as WAP phones, the example has been configured to support a generic WML device. Adding a Struts configuration for WML devices supports the view mode for a WML device. The following code fragment demonstrates how to set up the struts-configuration for the view mode for WML markup.

<init-param>

<param-name>config/wml/view</param-name>

<param-value>/WEB-INF/wml/view/struts-config.xml</param-value>

</init-param>

The search path that is set up in the Web deployment descriptor, takes markup, mode, and locale into account. When a WAP device connects to a portal server, the markup will be WML, and the mode will be view. This example does not use the locale that is requested.

The welcome file list also defines the welcome file for the wml/view mode.

<welcome-file-list>

<welcome-file>index.jsp</welcome-file>

<welcome-file>html/help/index.jsp</welcome-file>

**<welcome-file>wml/view/index.jsp</welcome-file>**

</welcome-file-list>

For more information about WML support in the Struts Portlet Framework, see [Changes to Struts JSPs](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrjsps.html).

**Logging**

The Struts Portlet Framework uses the Commons-Logging interface for a logging facility. The Struts Portlet Framework has supplied an implementation of the Commons-Logging Log interface that can be used to map the trace messages to the logging facility used by the portal server. This file is normally found in the [*PortalServer\_root*](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/reference/wpsdirstr.html#wpsdirstr__wp_root)/log directory. Setting properties in the [*PortalServer\_root*](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/reference/wpsdirstr.html#wpsdirstr__wp_root)/shared/app/config/log.properties file enables trace logging. The trace string for tracing a Struts portlet application can be configured in log.properties. The trace string can be modified to add or remove classes. The entire trace string must be on one line in log.properties.

The logging in WebSphere Portal typically creates a logging object per class and the trace messages display the log class that the message is from. This is very convenient to determine where a trace message originated from and also allows only enabling the tracing on certain classes.

The typical trace string configured in the log.properties file for tracing a Struts application would be as shown below (all on one line). This can be modified to add or remove classes.

**IBM container:**

traceString=

org.apache.struts.\*=all:

com.ibm.wps.portlets.struts.\*=all:

com.ibm.wps.struts.common.\*=all:

com.ibm.wps.struts.base.\*=all

Some of the Apache source uses a string to specify the class name and, in some cases, this string may not be a valid class. In those cases the class name in the trace message will be com.ibm.wps.portlets.struts.logging.WpsStrutsTraceLogger. This package or class should be specified in the traceString to account for these occasions.

The log messages from Struts use the Portal Server's logging facility when logging is enabled. When this is the case, the log messages from Struts may not display the correct method name as part of the log message. There is a configuration parameter to enable correcting this problem, but this is an expensive operation. This feature should only be enabled when needed. This feature is not required as much with the logging changes implemented in this release.

**JSR compliant container:**

traceString=

org.apache.struts.\*=all:

com.ibm.portal.struts.\*=all:

com.ibm.wps.struts.common.\*=all:

com.ibm.wps.struts.base.\*=all

**Customizing logging**

The Struts Portlet Framework specifies the commons logging Log Factory in the META-INF/services directory. In the samples, the org.apache.commons.logging.LogFactory file specifies the default logger that the Struts Portlet Framework uses. The log factory class name is dependent on the WebSphere Portal container. The org.apache.commons.logging.LogFactory file can be edited to specify other commons log factories.

**Using the default logging implementation class**

**Struts Portlet Framework 5.0.3 or later** Struts Portlet Framework 5.0.3 and later no longer use the System property org.apache.commons.logging.LogFactory to specify the log factory class. The commons log factory can be specified either in the META-INF/services directory or the commons-logging.properties file that must be found through the classpath. The suggested way to specify the commons log factory is to through the META-INF/services directory of the WAR file.

**IBM container with a Struts Portlet Framework version prior to 5.0.3** Versions of the Struts Portlet Framework prior to SPF 5.0.3 set the logging implementation class using the System property org.apache.commons.logging.LogFactory. The portlet initialization method, init, in com.ibm.wps.portlets.struts.WpsStrutsPortlet calls the setCommonsLogFactory method.

public void setCommonsLogFactory()

{

String methodName = "setCommonsLogFactory";

s\_traceLogger.text( WpsStrutsTraceLogger.TRACE, methodName,

"Set common logging to use StrutsLogFactory" );

System.setProperty( "org.apache.commons.logging.LogFactory",

"com.ibm.wps.portlets.struts.logging.StrutsLogFactory");

}

**Note:** A portlet should not use this mechanism, as the system property will affect other applications in the WebSphere Application Server. If an older version of the Struts Portlet Framework is used, first the WpsStrutsPortlet should be extended and the setCommonsLogFactory should be implemented to do nothing. Then modify the application to specify the commons log factory through the org.apache.commons.logging.LogFactory file in the META-INF/services directory of the application.

**Struts Portlet Framework version 5.0.3 or later Log Factory's class name**

**Standard portlet container:** org.apache.commons.logging.LogFactory

**IBM portlet container:** com.ibm.wps.portlets.struts.logging.StrutsLogFactory

**Overriding the default logging implementation class**

Struts portlet developers can override the default logging implementation class. Depending on the version of Struts, the implementation varies.

**Struts Portlet Framework 5.0.3 or later** There are two ways to customize the logger. The first and recommended option is to specify the log factory in the org.apache.commons.logging.LogFactory file in the META-INF/services directory. The Commons Logging implementation will check the META-INF/services directory before looking for the commons-logging.properties file in the classpath to determine the class name of the LogFactory. The second option is to specify the commons-logging.properties file in the application's classpath. The org.apache.commons.logging.LogFactory file can be updated to specify the desired LogFactory.

**Note:** Version 5.0.3 of the Struts Portlet Framework used the commons-logging.properties file to specify the logger, and this file was packaged in the wp.struts-commons-logging.jar. Specifying the log factory in the META-INF/services directory will take precedent over the log factory specified in the wp.struts-commons-logging.jar file.

**IBM container with a Struts Portlet Framework version prior to 5.0.3** To override the default logging implementation class, developers can extend com.ibm.wps.portlets.struts.WpsStrutsPortlet and override the setCommonsLogFactory method. The developer could remove the call to set the org.apache.commons.logging.LogFactory System property in setCommonsLogFactory, and create the file org.apache.commons.logging.LogFactory in the META-INF/services directory (or the file commons-logging.properties in the WEB-INF/classes directory of the application to determine the logging implementation class. The META-INF/services directory is the preferred method.

**Note:** The portlet's web deployment descriptor will then need to be updated to specify the WpsStrutsPortlet subclass as the servlet class.

**Logging conflict between new and old versions of Struts portlets**

The JCL determines the logging implementation by first looking for the System property org.apache.commons.logging.LogFactory. If the System property has not been set, it then looks for the org.apache.commons.logging.LogFactory file in the META-INF/services directory and then for commons-logging.properties in the classpath.

The value of the System property becomes available to all portlets once it is set, and this would determine the logging implementation class for the portlets on the page. There would be no effect on the logging if the default logging implementation is used by all portlets. However, if the logging implementation class for a portlet is set in a custom common-logging.properties file, it would be ignored. To overcome this, the developer would need to override the method, which sets the System property. See the section Overriding the default logging implementation class for more details.

**Commons Logging jar**

The WebSphere Application Server ships a version of the commons-logging.jar file and specifies the default commons logger to be the com.ibm.ws.commons.logging.TrLogFactory class. If the Struts portlet does not specify a log factory then the default log factory of TrLogFactory is used. However, if the application ships a version of commons-logging, then a ClassCastException will result. This release of the Struts Portlet Framework no longer ships commons-logging.jar, but instead will use the version shipped in the Application Server. Older applications can continue to use the commons-logging.jar file packaged with the Struts Portlet Framework as long as the application specifies a logger, which is the typical case.

**Changes to the Struts configuration file**

The Struts configuration used by Struts portlets references the 1.1 version of the DTD.

<!DOCTYPE struts-config PUBLIC

"-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"

"http://jakarta.apache.org/struts/dtds/struts-config\_1\_1.dtd">

In order to have the Struts Portlet Framework operate properly, the RequestProcessor must be configured. The RequestProcessor is responsible for the main functionality of processing a request. The default RequestProcessor can be overridden in the struts-config.xml via the controller element. In the struts-config.xml, the following portal server RequestProcessor must be specified:

**Standard portlet container:**

<controller

processorClass="com.ibm.portal.struts.portlet.WpRequestProcessor">

</controller>

**IBM portlet container:**

<controller

processorClass="com.ibm.wps.portlets.struts.WpsRequestProcessor">

</controller>

Of course, you may also have had the need to extend the Struts *RequestProcessor* in order to extend the functionality of Struts. If you simply place the path to your *RequestProcessor* in the controller element in the struts-config file, your application will not work properly in the portlet environment. Instead, what you should do is have your *RequestProcessor* class extend the *WpsRequestProcessor* rather than the normal Struts RequestProcessor.

The following are the methods overridden in *WpsRequestProcessor*:

* processLocale
* processActionPerform()
* processActionForm()
* processActionForward()
* processPath()
* processRoles()
* processMapping()
* processValidate()
* doForward()
* doInclude()
* getMapping()

If you override any of the methods listed above, you should make sure to invoke the super class version of the method from within your method.

There are existing Struts Extensions which require you to use their *RequestProcessor*. Clearly, you cannot use their *RequestProcessor* and the *WpsRequestProcessor* at the same time. Instead, you would need to get the source to that *RequestProcessor* and following the approach just described.

**Tiles support**

Struts integrates the Tiles package for creating views. A Tiles sample is included with the Struts Portlet Framework that demonstrates how to configure Tiles in the Struts Portlet Framework. Tiles requires a specific Request Processor that is configured through a plugin. The following is from the Struts configuration to show how to configure Tiles for a Struts application that is deployed in WebSphere Portal.

**Standard portlet container:**

<!-- add the Tiles plugin. -->

<plug-in className="com.ibm.portal.struts.plugins.WpTilesPlugin">

<set-property property="definitions-config"

value="/WEB-INF/conf/tiles-def.xml"/>

<set-property property="definitions-debug" value="2"/>

<set-property property="definitions-parser-details" value="2"/>

<set-property property="definitions-parser-validate" value="true"/>

</plug-in>

The Tiles plugin requires that compatible request processor is configured in the controller element. The WpTilesPlugin has been written to accept the default Struts RequestProcessor, WpStrutsRequestProcessor, or the WpTilesRequestProcessor.

**IBM portlet container:**

<!-- add the Tiles plugin. -->

<plug-in

className="com.ibm.wps.portlets.struts.plugins.WpsStrutsTilesPlugin">

<set-property property="definitions-config"

value="/WEB-INF/conf/tiles-def.xml"/>

<set-property property="definitions-parser-details" value="2"/>

<set-property property="definitions-parser-validate" value="true"/>

</plug-in>

The Tiles plugin requires that compatible request processor is configured in the controller element. The WpsStrutsTilesPlugin has been written to accept the default Struts RequestProcessor, WpsStrutsRequestProcessor, or the WpsStrutsTilesRequestProcessor.

# TOPIC 4:- Supporting multiple Struts applications

Learn how Web applications that allow packaging more than one Struts-based portlet in a single application can be created either using Struts modules or a namescoped servlet context.

The Jakarta Struts Framework does not support configuring more than one Struts ActionServlet in the Web deployment descriptor. This limitation occurs because there is only one servlet context per Web application and the commons digester imports the Struts configuration into the servlet context. Struts stores the required information as attributes in the servlet context, and the information is retrieved by using globally defined key names. Struts will suffix these attribute names with the prefix of the Struts modules, so several Struts modules are supported in a single Web application, but multiple Struts applications are not.

The Struts Portlet Framework is used to create Struts applications that are deployed in WebSphere Portal. This section discusses two techniques for creating Web applications that allow packaging more than one Struts-based portlet in a single application. As stated, Jakarta Struts does not support declaring the Struts ActionServlet in the Web deployment descriptor more than once. The Struts Portlet Framework has the same limitation because the servlet context is used when the Struts configuration is digested in the portal as well. However, the Struts Portlet Framework supports two ways that an application can be developed in order to implement multiple Struts-based portlets. The first technique uses a single Struts controller while defining a Struts module for each portlet in the application. The second technique uses a servlet context wrapper that namescopes the attributes stored in the servlet context, so each portlet can have its own namespace in the servlet context.

**Using Struts modules (IBM portlet container only)**

One method of supporting multiple Struts portlets in a single war file is to define a Struts module for each portlet. The welcome file can be specified as a configuration parameter for each concrete portlet. This technique takes advantage of the fact that the Struts Portlet Framework uses the welcome file to set the Struts module. The prefix corresponding to the Struts module for a portlet is stored in the IViewCommand object. The IViewCommand object is retrieved for each portlet and the module prefix is used to select the module, so each portlet in the application will use the Struts module selected during the welcome file processing. Obviously, the portlet can then forward to other modules as well, but initially each portlet can have a unique configuration using the Struts module.

The Struts Portlet Framework controller, WpsStrutsPortlet, is defined as the servlet-class in the Web deployment descriptor. The corresponding abstract portlet is then defined in the portlet deployment descriptor. The application can then define each concrete portlet, and specify a welcome file to use. The configuration for one of the concrete portlets is demonstrated below.

<concrete-portlet-app uid="wp.struts.legacy.examples.MultipleStrutsApp.1">

<portlet-app-name>Struts Legacy Multiple Modules Application 1</portlet-app-name>

<concrete-portlet href="#Portlet\_1">

<portlet-name>Struts Legacy Multiple Modules 1</portlet-name>

<default-locale>en</default-locale>

<language locale="en">

<title>Struts Legacy Multiple Modules 1</title>

<title-short>Struts Legacy Multiple Modules 1</title-short>

<description>Struts Legacy Multiple Modules 1</description>

<keywords>WPS, Struts, Legacy</keywords>

</language>

<config-param>

<param-name>viewMode.page</param-name>

<param-value>portlet1/welcome.jsp</param-value>

</config-param>

<config-param>

<param-name>editMode.page</param-name>

<param-value>portlet1/edit.jsp</param-value>

</config-param>

</concrete-portlet>

</concrete-portlet-app>

The application can configure a separate Struts module that maps to each portlet. This scheme is customizable and will support as many Struts modules as are desirable. However, only one abstract portlet can be defined because there can only be one Struts-based servlet-class. All of the concrete portlets created from the one abstract portlet must share the abstract's portlet configuration. See the SPFLegacyMultipleModules example for more information.

**Using a namescoped servlet context**

Multiple Struts portlets in a single Web application can also be supported by configuring the portlet to use a servlet context wrapper. The (Wps)StrutsPortlet class instantiates the WpsActionServlet object to digest the Struts configuration. The Wp(s)ActionServlet extends the Struts ActionServlet and uses the ActionServlet's init implementation to digest the configuration. The (Wps)StrutsPortlet object will invoke the init method of the Wp(s)ActionServlet to digest the configuration. Since the (Wps)StrutsPortlet instantiates the Wp(s)ActionServlet and calls the init method, the Struts Portlet Framework has the opportunity to provide a servlet context wrapper. If the application is configured to use the servlet context wrapper, then the Wp(s)ActionServlet will return the servlet context wrapper when the getServletContext method is invoked. The servlet context wrapper defines a namespace unique to the portlet and then stores the attributes in the real servlet context. The concept is similar to how Struts prefixes the attributes in the servlet context for the configuration of each Struts module.

**Note:** Note: The Wp(s)ActionServlet should be use to obtain the servlet context wrapper when this technique is used. The servlet context obtained from the pageContext, for example, will not be the wrappered servlet context. The Wp(s)ActionServlet can be obtained from the request object using the Globals.ACTION\_SERVLET\_KEY.

**For an IBM portlet container:**

The WpsStrutsPortlet class, which is specified as the servlet class in the Web deployment descriptor, can be defined multiple times when the NamescopeServletContext initialization parameter, init-param, is set to true.

<servlet id="wp.struts.legacy.examples.ServletContext.1">

<servlet-name>StrutsLegacyMultipleServletContexts1</servlet-name>

<servlet-class>

com.ibm.wps.portlets.struts.WpsStrutsPortlet

</servlet-class>

<init-param>

<param-name>config</param-name>

<param-value>

/WEB-INF/portlet1/struts-config.xml

</param-value>

</init-param>

<init-param>

<param-name>struts-servlet-mapping</param-name>

<param-value>\*.do</param-value>

</init-param>

<init-param>

<param-name>NamescopeServletContext</param-name>

<param-value>true</param-value>

</init-param>

</servlet>

An abstract portlet for each servlet-class can then be defined in the portlet deployment descriptor.

<portlet id="Portlet\_1"

href="WEB-INF/web.xml#wp.struts.legacy.examples.ServletContext.1"

major-version="1" minor-version="0">

<portlet-name>

Struts Legacy Multiple Servlet Contexts 1

</portlet-name>

<cache>

<expires>0</expires>

<shared>NO</shared>

</cache>

<allows>

<maximized />

<minimized />

</allows>

<supports>

<markup name="html">

<view />

<edit />

</markup>

</supports>

</portlet>

**For a standard portlet container:**

The StrutsPortlet class, which is specified as the portlet class in the portlet deployment descriptor, can be defined multiple times when the NamescopeServletContext initialization parameter, init-param, is set to true.

<portlet>

<description xml:lang="en">

Struts Standard Multiple Servlet Contexts 2

</description>

<portlet-name>

Struts Standard Multiple Servlet Contexts 2

</portlet-name>

<display-name xml:lang="en">

Struts Standard Multiple Servlet Contexts 2

</display-name>

<portlet-class>

com.ibm.portal.struts.portlet.StrutsPortlet

</portlet-class>

<init-param>

<name>config</name>

<value>/WEB-INF/portlet2/struts-config.xml</value>

</init-param>

<init-param>

<name>struts-servlet-mapping</name>

<value>\*.do</value>

</init-param>

<init-param>

<name>NamescopeServletContext</name>

<value>true</value>

</init-param>

**...**

</portlet>

Each portlet is independent of the other Struts-based portlets. See the SPFLegacyMultipleServletContexts.war and SPFStandardMultipleServletContexts.war examples for more information.

# TOPIC 5:- Creating link tags in Struts

Get an overview of how to write tags to create links in a JSP in both a servlet and a portlet in Struts framework.

Tags that create links need to create links that are serviced by the portlet. The URL that is created because of the Struts processing needs to be passed as a parameter back to the portlet. There should be a common tag for both a servlet and a portlet. The following demonstrates how to write a tag that can be used by a JSP in both a servlet and a portlet.

PortletApiUtils portletUtils = PortletApiUtils.getInstance();

if (portletUtils != null)

{

Object pResponse = portletUtils.getPortletResponse((HttpServletRequest)

pageContext.getRequest());

Object portletURI =

portletUtils.createPortletURIWithStrutsURL(pResponse,

calculateURL());

results.append(portletURI.toString() );

}

else

{

// servlet environment

results.append(calculateURL());

}

The else clause was the original statement for the servlet-only environment. An instance of PortletAPIUtils is initially obtained. In a non-portlet environment this call would return null. The initialization of the Struts Portlet Framework support would set the instance of the PortletAPIUtils implementation. A PortletResponse object is obtained as an Object. The PortletURI is then created with a DefaultPortletAction and a parameter that contains the Struts path through the call to createPortletURIWithStrutsURL.

## Creating links in a JSP

The preferred method of creating a link in a JSP is to use a tag. This is the most convenient method to assure that the link will be portal-aware and also take into account Struts Modules. The Struts module implementation requires that the link is prefixed with the ModuleConfig's prefix. A typical Struts tag implementation would include logic similar to the following.

ModuleConfig config =

(ModuleConfig) pageContext.getRequest().getAttribute(Globals.MODULE\_KEY);

if (config != null) {

value.append(config.getPrefix());

}

There are times when a link needs to be created in a JSP using Java code. The links created in a JSP must also prefix the ModuleConfig prefix so the links support Struts Modules. PortletApiUtils has a convenience method that can be use to prefix the URL using the ModuleConfig. The method is called addModulePrefix. The following snippet demonstrates creating a link in a JSP.

<%@ page language="java" import="com.ibm.wps.struts.common.PortletApiUtils"

%>

<%

PortletApiUtils portletUtils = PortletApiUtils.getInstance();

%>

<%

if (portletUtils != null)

{

String url = "/bean-write.jsp";

// add the module prefix to the url

url = portletUtils.addModulePrefix(url, request);

Object portletURI = portletUtils.createPortletURIWithStrutsURL(request, url);

%>

<a href="<%=portletURI.toString()%>"><bean:write></a>

<%

}

%>

# TOPIC 6:- Formatting XML documents with XSLT

Learn how a Struts application supports applying a style sheet to XML data.

The typical way that a Struts application supports applying a style sheet to XML data is to implement a Struts Action that writes directly to the response object. The IStrutsPrepareRender interface allows implementing a Struts action that will be executed in the render phase of portal, and therefore will have a response object that can be written to. The SPFStandardTransformation.war and SPFLegacyTransformation.war are examples of how to implement a Struts action in the Struts Portlet Framework that can apply a style sheet to an XML document

# TOPIC 7:- Static content in Struts

Portlet JSPs can link to the static content within the portlet’s WAR directory structure by using the services of the portlet container to create portlet URIs from which the content can be referenced.

Portlet JSPs cannot link directly to static content within the portlet's WAR directory structure. Instead, they have to use the services of the portlet container to create portlet URIs from which the content can be referenced. Links within static pages (for example, HTML pages) also need to be converted into portlet URIs. The Struts Portlet Framework processes links to static content and converts them to the form of portlet URIs so that linked content is brought back into the portlet and displayed.

## Creating a Home button for static content

When the Struts Portlet Framework is used to display static content, it can be difficult to navigate back through a number of screens to a location earlier in the browser history. To overcome this, you can add a homeFromStatic forward to your action mapping which references the static content. This causes a home button to be added to your static content. Pressing the home button causes the link specified by the homeFromStatic forward to be used.

As with any action mapping, it is possible also to use a global forward. Thus, if for all sequences of static pages it was desired to use the same link for every home button, one could use a global forward, as follows:

<global-forwards>

<forward name="homeFromStatic" path="/index.jsp"/>

...

</global-forwards>

A homeForStatic forward can also be created for a specific ActionMapping, as follows:

<action path="/staticpage"

type="org.apache.struts.actions.ForwardAction"

parameter="/html/view/static.html">

<forward name="homeFromStatic" path="/index.jsp"/>

</action>

The home button is not only placed on the first static content page, but on all pages generated by links resulting from that page. If no homeFromStatic forward is found with the action mapping which initiated the sequence of static content pages, and there is no global homeFromStatic forward either, the home button is not generated.

# TOPIC 8:- Migrating existing Struts applications

Get an overview of how the existing Struts applications can be migrated using the Struts Portlet Framework so that these applications can be deployed in IBM® WebSphere® Portal.

* [Migrating servlet-based Struts applications](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrmig.html#wpstrmig__migrating_blank)
* [Migrating servlet-based Struts applications to the Standard version](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrmig.html#wpstrmig__migrating_standard)

The Jakarta Struts distribution ships an example Struts application that demonstrates the features of Struts, including implementing Struts Actions, creating custom tags, using the Struts tag libraries, authoring JSPs, and form validation. The Struts Portlet Framework demo builds on this example by incorporating WebSphere Portal features, including support for help mode and WML markup. The Struts example required very few changes to enable it for the Struts Portlet Framework.

The file structure of the WAR file for a Struts Portlet Framework is the same as that used for a Struts servlet. The Struts application in the portal server is a WAR file just like it is in the servlet environment. The portlet WAR file has some additional JARS and other requirements, but the basis of the implementation is similar to the servlet application. See [WAR file directory structure](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsbscfg.html) for more information.

**Migrating servlet-based Struts applications**

Existing Struts applications can be migrated using the Struts Portlet Framework so the application can be deployed in WebSphere Portal. Since Struts is a framework there are many variations to how the application can be built with Struts. There are also numerous other frameworks that may also be incorporated into the Struts application. The steps in this section can be used as a starting point for the migration effort, but may not cover all of the issues that can be encountered.

1. Make sure the existing Struts application has been built as a Struts 1.1 application. (See [The Struts Application Framework](http://jakarta.apache.org/struts/).)
2. Check Struts Actions to see if the action writes directly to the response object. If it does then, the action must be modified to either return an ActionForward instead, or check to see if the IStrutsPrepareRender interface should be used.
3. The Web deployment descriptor must be updated to use the WpsStrutsPortlet as the servlet class instead of the ActionServlet. (See [Changes to the Web deployment descriptor](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html).)
4. The servlet mapping for Struts actions must be specified as the struts-servlet-mapping init parameter. (See [Changes to the Web deployment descriptor](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html).)
5. Create a portlet.xml. The SPFLegacyMailReader.war can be used as an example. (See [Portlet deployment descriptor](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsptdesc.html).)
6. Modify the struts-config.xml to specify the WpsRequestProcessor as the controller. (See [Struts configuration file](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html).)
7. Modify tags that use pageContext.forward() to use the PortletApiUtils forward. (See [Forwards and redirects](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html).)
8. Modify JSPs that use a forward to use the logic forward tag. (See [Changes that affect Struts JSPs](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrjsps.html).)
9. Modify JSPs as necessary to use the Struts tags for creating URLs, like the Struts Link and Form tags. (See [Creating portlet URIs](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrjsps.html)).
10. The JAR files from the WEB\_INF/lib directory of the SPFLegacyBlank.war must be used. These files are the Struts JARs and the required Struts Portlet Framework JARs.
11. The TLD files must be updated from the WEB\_INF/lib directory of SPFLegacyBlank.war. Verify the taglib attributes and that the JSP correctly reference the TLD files. This has been a common source of problems when migrating existing applications.
12. The JSPs should be modified so they do not use html, head and body elements. All HTML output to the portal is written in the context of an HTML table cell.

Refer to the topic, [Update portlets built with the Struts Portlet Framework](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/migrate/mig_prep.html), to determine what level of Struts your application is using. You might have to follow some additional migration steps that are described in that topic.

**Migrating servlet-based Struts applications to the Standard version**

Existing Struts applications can be migrated using the Struts Portlet Framework so the application can be deployed in WebSphere Portal. Since Struts is a framework there are many variations to how the application can be built with Struts. There are also numerous other frameworks that may also be incorporated into the Struts application. The steps in this section can be used as a starting point for the migration effort, but may not cover all of the issues that can be encountered.

1. Make sure the existing Struts application has been built as a Struts 1.1 application. (See [The Struts Application Framework](http://jakarta.apache.org/struts/).)
2. Check Struts Actions to see if the action writes directly to the response object. If it does then, the action must be modified to either return an ActionForward instead, or check to see if the IStrutsPrepareRender interface should be used.
3. The Web deployment descriptor must be updated to use the StrutsPortlet as the portlet class. Remove the servlet-class from the Web deployment descriptor, and specify the StrutsPortlet as the portlet class in the portlet deployment descriptor. (See [Changes to the Web deployment descriptor](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html#wpstrconfig__web_xml_changes).)
4. The servlet mapping for Struts actions must be specified as the struts-servlet-mapping init parameter. The portlets init parameters are specified in the portlet deployment descriptor.
5. Create a portlet.xml. The SPFStandardMailReader.war can be used as an example. (See [Portlet deployment descriptor](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpsptdesc.html#wpsptdesc__portlet.xml).)
6. Modify the struts-config.xml to specify the WpRequestProcessor as the controller. (See [Struts configuration file](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrconfig.html#wpstrconfig__struts_cfg).)
7. Modify tags that use pageContext.forward() to use the PortletApiUtils forward. (See [Forwards and redirects](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrsource.html#wpstrsource__forwards).) The following example illustrates the change in which the PortletApiUtils object is obtained:
   * **Old:** PortletApiUtils portletUtils = PortletApiUtils.getInstance();
   * **New:** PortletApiUtils portletUtils = PortletApiUtils.getUtilsInstance();
8. Modify JSPs that use a forward to use the logic forward tag. (See [Changes that affect Struts JSPs](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrjsps.html#wpstrjsps__struts_jsps).)
9. Modify JSPs as necessary to use the Struts tags for creating URLs, like the Struts Link and Form tags. (See [Creating portlet URIs](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/wpstrjsps.html#wpstrjsps__return)).
10. The JAR files from the WEB\_INF/lib directory of the SPFStandardBlank.war must be used. These files are the Struts JARs and the required Struts Portlet Framework JARs.
11. The TLD files must be updated from the WEB\_INF/lib directory of SPFStandardBlank.war. Verify the taglib attributes and that the JSP correctly reference the TLD files. This has been a common source of problems when migrating existing applications.
12. The JSPs should be modified so they do not use html, head and body elements. All HTML output to the portal is written in the context of an HTML table cell.

**MODEL SPI OVERVIEW**

# TOPIC 1:- Sub packages of the Model SPI

The Model SPI includes the following subpackages.

* [The sub package com.ibm.portal.admin](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/dgn_modelpkg.html#modelapic__admin)
* [The sub package com.ibm.portal.content](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/dgn_modelpkg.html#modelapic__content)
* [The sub package com.ibm.portal.identification](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/dgn_modelpkg.html#modelapic__identification)
* [The sub package com.ibm.portal.navigation](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/dgn_modelpkg.html#modelapic__navigation)
* [The sub package com.ibm.portal.portletmodel](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/dgn_modelpkg.html#modelapic__portletmodel)
* [The sub package com.ibm.portal.wire](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/dgn_modelpkg.html#modelapic__portal_wire)

See [Model SPI overview](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/dgn_modelovw.html) for a description of the main package in the Model SPI: com.ibm.portal.

**The sub package com.ibm.portal.admin**

WebSphere Portal provides several list models that contain information on installed resources, such as supported languages, supported markups, skins and themes. For each of these, a specific ListModel exists that provides information on these resources. This package defines several list models:

* Language list model
* Markup list model
* Skin list model
* Theme list model

Each list model contains elements implementing interfaces that coincide with their names: The language list model has Language elements, the markup list model has Markup elements, the skin list model has Skin elements, and the theme list model has Theme elements.

The LanguageList contains all languages that are supported by the portal. The MarkupList contains all markups that the portal generally supports. From these, a resource may only support a selection. The SkinList and the ThemeList hold the themes and skins that are installed for the portal.

**The sub package com.ibm.portal.content**

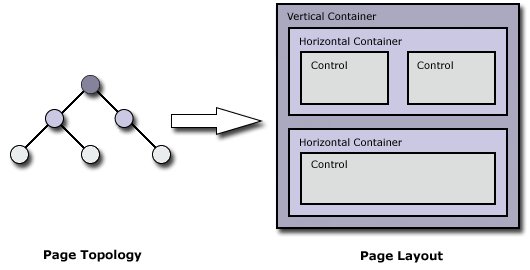
Elements of this package define how content is represented in the portal. The two main models of this package are the ContentModel and the LayoutModel. The content model defines a tree structure for content elements (such as pages and labels); it is used to group these logically. Please note that the topology of the content model currently dictates the topology of the navigation model (see section "The sub package com.ibm.portal.navigation"). Theoretically, the topologies of content model and navigation model are allowed to diverge.

Currently, the elements of the content model can be pages, labels or URLs. For each type, a specific ContentNode interface exists: ContentPage, ContentLabel and ContentURL. Each content node has different information associated with it.

Each content node can provide the type of content node it represents; if this type is ContentNodeType.PAGE, a model exists that represents the layout of that page. It can be obtained with the getLayoutModel method of the content model. The layout model holds LayoutNode elements that describe the layout and contents of the page. Each layout node can return metrics in form of the LayoutMetrics interface. It can describe attributes of the node such as width or orientation (horizontal/vertical).

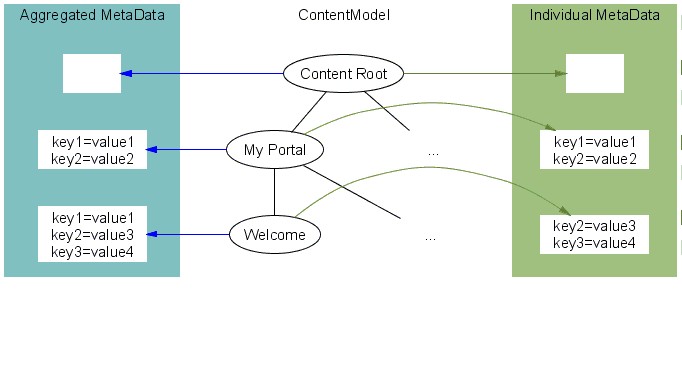
The elements of the layout model are either LayoutContainers that define rows and columns or LayoutControls, which represent portlets. This information is used when a page is rendered.

The following figure shows how page layout information and page content is represented. The figure depicts a page with three portlets. The surrounding vertical and horizontal containers define the layout while the controls hold the portlets that provide the actual content presented to the user.



Content nodes can provide meta data through the com.ibm.portal.MetaDataProvider interface. Meta data can be used to associate arbitrary information with a content node. The content meta data model provides a view on meta data of the nodes in the content model. It aggregates the meta data of the individual nodes into one meta data object by using the hierarchy of the nodes as described in the content model.

The following figure shows the relationship between the individual meta data that are exposed by content nodes and the aggregated view that the content meta data model provides:



The meta data of individual content nodes are exposed in XML Access scripts using the <parameter> tag.

**The sub package com.ibm.portal.identification**

This package holds the Identification interface which allows the conversion between ObjectID objects and their string representation. This is required where an object ID is to be passed as a parameter that can only be a String (for inclusion into URLs, for example). An implementation of this interface can be retrieved by using a JNDI lookup, as in the following example.

try

{

Context ctx = new InitialContext();

Name ctxName = new CompositeName("portal:services/Identification");

Identification identification = (Identification) ctx.lookup(ctxName);

String serializedOID = identification.serialize(aOID);

...

ObjectID anotherOID = identification.deserialize(serializedOID);

}

catch (SerializationException sx)

{

// some error handling code here

}

catch (NamingException nx)

{

// some error handling code here

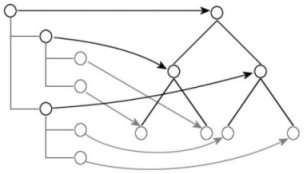
}

**The sub package com.ibm.portal.navigation**

This package defines the navigation model (NavigationModel) which represents a view on content model used for navigation. As described in section "The sub package com.ibm.portal.content", the topology of the navigation model currently corresponds with that of the content model - each node in the content model has an equivalent navigation node at the same hierarchical level.

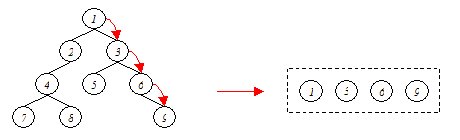
The elements of the navigation model are NavigationNodes. Each such node can reference a ContentNode. The navigation nodes have a title that is provided through the Localized interface. The following figure shows the connection between the navigation model and the content model.

The navigation model is illustrated on the left side of the figure, while the content model is illustrated on the right side of the figure. The figure shows how nodes in the navigation model can reference a node in the content model.



When a user is navigating through the portal, the currently selected navigation node is important to render the current page. The NavigationSelectionModel reflects the current selection and represents a list that defines a path through the navigation model.

The navigation model is illustrated on the left side of the figure, while the navigation selection model is illustrated on the right side of the figure. The figure shows how the navigation selection model defines a path through the navigation model.



The last node of this path is always the currently selected navigation node, and its referenced content node represents what is aggregated for the user to see (normally a page represented through a ContentPage object).

**The sub package com.ibm.portal.portletmodel**

The PortletModel represents portlets and their configuration data in the portal. It contains the following elements:

**WebApplication**

represents a deployed WAR file in the portal.

**Portlet**

the programmer in the deployment descriptor of a portlet web application. Each portlet web application contains one or more portlets.

**PortletDefinition**

represents administrator configuration for a portlet. Multiple portlet definitions can be associated with the same portlet, so that the same portlet code can be invoked with different administrative settings. The administrative UI of the portal allows to create new portlet definitions as copies of existing ones.

**PortletEntity**

represents end-user configuration for a portlet. It is normally created by placing a portlet definition on a page; the same portlet definition can be added to multiple pages, resulting in multiple portlet entities. There can be two levels of end-user configuration (shared and personalized configuration) that are stored in the form of separate portlet entities.

**PortletWindow**

represents a particular view on a portlet. It normally corresponds to a LayoutControl in the layout model. The same PortletWindow can be associated with different PorltetEntities for different users, if they have personalized the portlet independently.

**CommunicationEndpoint**

represents an endpoint that is available for communication defined for the given portlet. Such endpoints can be of type CommunicationSource or CommunicationTarget and are available via the CommunicationEndpointProvider. This provider can be retrieved by the getEndpointProvider method on the PortletDefinition. For JSR 286 portlets that were developed for eventing, PublishingEventDefinition[s] and ProcessingEventDefinition[s] represent the events that a single portlet can process or publish as defined in the portlet.xml. The PublishingEventDefinition is a CommunicationSource and the ProcessingEventDefinition is a CommunicationTarget for JSR286 portlets.

**The sub package com.ibm.portal.wire**

The WireModel represents the interconnections between JSR 168 cooperative portlets or JSR 286 eventing portlets. A WireModel can be obtained from a LayoutModel by using the getWireModel method of the layout model. It contains only interconnections that originate from the page that the parent layout model represents. It contains the following element:

**Wire**

represents the interconnection between a CommunicationSource of the PortletDefinition of a PortletWindow on a given source page and the CommunicationTarget of the PortletDefinition of a PortletWindow on a target page. Examples of interconnections are:

* Between a JSR 168 cooperative portlet with an output property and a JSR 168 cooperative portlet with a target action defined.
* Between a JSR 286 portlet that defines a publishing event and a JSR 286 portlet that defines a processing event.

# TOPIC 2:- Obtaining a model from the portal

Portal models can be obtained using three different ways, depending on where the code using them resides.

**JNDI lookup**

A JNDI lookup can be performed for code that resides in a request/response cycle of the portal, for example in a JSP. The lookup results in a home interface from which a model provider can be obtained. Interfaces for the models mentioned above can be found in com.ibm.portal.model. To avoid calling the JNDI lookup too often, it is recommended to store the provider object. The following example, which is only applicable to non-portlet code running inside of a portal, shows how the content model is obtained using a JNDI lookup.

*Figure 1. Obtaining a content model using JNDI lookup*

try

{

Context ctx = new InitialContext();

ContentModelHome home = (ContentModelHome)

ctx.lookup("portal:service/model/ContentModel");

if (home != null) {

ContentModelProvider provider = home.getContentModelProvider();

ContentModel model =

provider.getContentModel(aRequest, aResponse);

...

}

}

catch (NamingException nx)

{

// some error handling code here

}

**Portlet service (standard portlet API)**

Standard portlets can access models through specific portlet services. These are located in com.ibm.portal.portlet.service.model. The following example shows how the navigation model is obtained by a standard portlet.

*Figure 2. Obtaining the navigation model*

PortletServiceHome psh;

javax.naming.Context ctx = new javax.naming.InitialContext();

boolean serviceAvailable = false;

try {

psh = (PortletServiceHome)

ctx.lookup("portletservice/com.ibm.portal.portlet.service.model.NavigationModelProvider");

serviceAvailable = true;

}

catch(javax.naming.NameNotFoundException ex)

{

... error handling...

}

...

if (serviceAvailable) {

NavigationModelProvider provider = (NavigationModelProvider)

psh.getPortletService(NavigationModelProvider.class);

NavigationModel model = provider.getNavigationModel(aRequest, aResponse);

...

}

**Portlet service (IBM Portlet API)**

IBM portlets can access models through specific portlet services. These are located in com.ibm.portal.ibmportlet.service.model. The following example shows how the navigation model is obtained by an IBM portlet.

*Figure 3. Obtaining the navigation model (IBM portlet)*

NavigationModelProvider provider = (NavigationModelProvider)

PortletContext.getService("com.ibm.portal.ibmportlet.service.model.NavigationModelProvider");

if (provider != null) {

NavigationModel model = provider.getNavigationModel(aRequest, aResponse);

...

}

**Note the following limitations for obtaining a portal model:**

1. Portlets that use the following packages must not be enabled for parallel portlet rendering (PPR):
   * com.ibm.portal.admin.MarkupList
   * com.ibm.portal.admin.SkinList
   * com.ibm.portal.admin.ThemeList
   * com.ibm.portal.content.ContentMetaDataModel
   * com.ibm.portal.content.ContentModel
   * com.ibm.portal.navigation.NavigationModel
   * com.ibm.portal.navigation.NavigationSelectionModel
   * com.ibm.portal.portletmodel.PortletModel

**Note:** The restriction covers all the models that can be obtained from the above packages.

1. WSRP portlets must not use the Model SPI.
2. Model access is only possible after the portal has initialized the request appropriately. Access is possible inside of code invoked through the portal servlet. Models cannot be accessed in servlet filters.

# TOPIC 3:- Obtaining the object ID for a page or portlet

There are several use cases when a portlet needs to obtain the object ID used to uniquely identify a portlet or a page. For example, the object ID of a page definition is required for a portlet to launch a dynamic instance of that page. You can use the lookup() method of the JNDI Context class to obtain the object ID for a portlet or a page, passing the unique name of the page or portlet. As an alternative for portlets, you can obtain the object ID by using JNDI lookup() and passing a combination of the portlet application ID and the portlet name.

* For standard portlets, the portlet application ID is the value of the ID attribute of the <portlet-app/> element in the portlet.xml. However, this attribute is not required by the Java Portlet Specification. If the ID attribute is not set, the portlet WAR file name is used as the portlet application ID.
* For IBM portlets, the portlet application ID is the UID attribute of the <portlet-app/> element. This attribute is required.

The following example shows both ways to find an object ID.

// initialization

Context ctx = new InitialContext();

// portal:uniquename prefix is required for unique name lookup

Name uniqueName = new CompositeName("portal:uniquename");

// portal:config/ prefix required for portlet definition lookup

Name portletName = new CompositeName("portal:config/portletdefinition");

// the unique name assigned to the page is example.org.page

uniqueName.add(org.example.page);

ObjectID oidForName = (ObjectID) ctx.lookup(uniqueName);

// appID and portletName have already been set programmatically

portletName.add(appID);

portletName.add(portletName);

ObjectID portletDefOID = (ObjectID) ctx.lookup(portletName);

The Name used for the lookup() method is created from a CompositeName, which is prepopulated with the required portal prefixes enclosed in quotes. This technique is used to avoid having to escape special characters in the prefix.

For this example, the following packages are required at a minimum for the import statements:

* javax.naming.CompositeName
* javax.naming.Context
* javax.naming.Name
* com.ibm.portal.ObjectID

# TOPIC 4:- Model SPI samples

The Model SPI can be used in portlets, themes, and skins. The models can be used with authenticated users and also with the anonymous user. These samples focus on JSPs, but the code can also be used in JAVA source files.

**Displaying the portal selection path (breadcrumb trail)**

The following example shows how to render a breadcrumb trail that shows the current selection path.

*Figure 1. Breadcrumb trail example using the Model SPI*

<%@ taglib uri="http://www.ibm.com/xmlns/prod/websphere/portal/v6.0/portal-fmt" prefix="portal-fmt" %>

<%@ taglib uri="http://www.ibm.com/xmlns/prod/websphere/portal/v6.0/portal-navigation" prefix="portal-navigation" %>

<%@ page import="com.ibm.portal.model.NavigationSelectionModelHome" %>

<%@ page import="com.ibm.portal.model.NavigationSelectionModelProvider" %>

<%@ page import="com.ibm.portal.navigation.NavigationSelectionModel" %>

<%@ page import="com.ibm.portal.navigation.NavigationNode" %>

<%@ page import="com.ibm.portal.ModelException" %>

<%@ page import="java.util.Iterator" %>

<%@ page import="javax.naming.InitialContext" %>

<%@ page import="javax.naming.Context" %>

<%@ page import="javax.naming.NamingException" %>

<%

try{

Context ctx = new InitialContext();

NavigationSelectionModelHome home = (NavigationSelectionModelHome)

ctx.lookup("portal:service/model/NavigationSelectionModel");

if (home != null) {

NavigationSelectionModelProvider provider =

home.getNavigationSelectionModelProvider();

NavigationSelectionModel model =

provider.getNavigationSelectionModel(request, response);

for (java.util.Iterator i = model.iterator(); i.hasNext(); )

{

NavigationNode node = (NavigationNode) i.next();

if (i.hasNext()) {

%>

<a href="<portal-navigation:navigationUrl type='link' varname='<%=node%>'/>">

<portal-fmt:title varname='<%=node%>'/>

</a>

&gt;

<%

}

else

{

%>

<portal-fmt:title varname='<%=node%>'/>

<%

}

}

}

}

catch (ModelException mx) {

%>

<p><span style="color:#ff0000">A model exception occured</span></p>

<%

}

catch (NamingException nx) {

%>

<p><span style="color:#ff0000">A naming exception occured</span></p>

<%

}

%>

This example uses a JNDI lookup to obtain the navigation selection model. The model is then iterated and for each node a title is produced using the <portal:title/> tag. Until the last node is reached the <portal:navigationUrl/> tag is used to generate links to the referenced pages.

**Getting page layout information**

This example shows portlet code that displays layout information of a page as retrieved from the layout model of that page. The portlet outputs the layout structure of the page it resides on. Two methods of the portlet are shown in the table below. The first, showTableModel, is the entry point for rendering a table that shows the layout of a page. The page is identified through the selected navigation node of the navigation selection model. The other method, printLayoutElement, recursively traverses the layout model and outputs markup according to the element it meets.

*Figure 2. Example: Getting page layout for a portlet*

/\*\*

\* Displays a table model of the layout of a page.

\*

\* @param aWriter the writer where the output goes to

\* @param aLayoutModel the layout model to show

\* @param aPage the page for which to show the layout model\

\* @param aRequest the portlet request

\* @throws ModelException if an exception occurs

\*/

private void showLayoutModel(final PrintWriter aWriter,

final LayoutModel aLayoutModel, final ContentPage aPage,

final RenderRequest aRequest)

throws ModelException {

aWriter.print("<h3>Layout for page ");

aWriter.print(getTitle(aPage));

aWriter.println("</h3>");

// invoke the recursive traversal of the layout model;

// start with the root node

printLayoutNode(aWriter, aLayoutModel,

(LayoutNode) aLayoutModel.getRoot(), aRequest);

}

/\*\*

\* Outputs a single element of the layout model to the output writer.

\*

\* @param writer the writer where the output goes to

\* @param model the layout model to use

\* @param node the current node of the layout model

\* @param request the portlet request

\* @throws ModelException if an exception occurs

\*/

private void printLayoutNode(final PrintWriter writer, final LayoutModel model,

final LayoutNode node, final RenderRequest request) throws ModelException {

if (node != null) {

if (node instanceof LayoutControl) {

// output control information

writer.print("<b>Control</b> (");

writer.print(getTitle((LayoutControl) node));

writer.println(")");

}

else {

writer.println("<table border=\"1\">");

// get the layout metrics

// (needed to find out the orientation of containers)

final LayoutMetrics metrics = node.getLayoutMetrics();

final Object info = metrics.getValue(LayoutMetrics.ORIENTATION);

writer.print("<tr><td><b>Container</b> (");

writer.print(info);

writer.println(")");

if (info == Orientation.HORIZONTAL) {

writer.println("<table>");

writer.println("<tr>");

// recurse for the horizontal container

if (model.hasChildren(node)) {

for (Iterator i = model.getChildren(node); i.hasNext();) {

writer.println("<td valign=\"top\">");

printLayoutNode(writer, model, (LayoutNode) i.next(), request);

writer.println("</td>");

}

}

writer.println("</tr>");

writer.println("</table>");

}

else {

// recurse for the vertical container

if (model.hasChildren(node)) {

writer.println("<table>");

for (Iterator i = model.getChildren(node); i.hasNext();) {

writer.println("<tr><td>");

printLayoutNode(writer, model, (LayoutNode) i.next(), request);

writer.println("</td></tr>");

}

writer.println("</table>");

}

}

writer.println("</td></tr></table>");

}

}

}

**Remote Model SPI REST service**

The Remote Model SPI gives you access to portal models through REST services. It allows you to obtain and modify portal resources that are exposed by some of the models of the model SPI remotely, that is from clients that are outside the JVM of the server. This is achieved by means of REST services.

The Remote Model SPI supports the following models:

**ContentModel**

This allows you to obtain and modify the content topology and the properties of content nodes such as pages, labels, and content URLs.

**NavigationModel**

This allows you to obtain the navigation topology only, as the navigation model is implied by the structure of the content model.

**LayoutModel**

This allows you to obtain and modify the layout of a page, that is the topology of layout elements of a page, and the properties of layout elements, such as layout containers and layout controls.

**PortletModel**

This allows you to obtain, create, update, and delete portlets.

Note that the Remote Model SPI currently does not support the following models:

* LanguageList
* MarkupList
* SkinList
* ThemeList

The Remote Model SPI uses feeds in the [Atom Syndication Format](http://www.ietf.org/rfc/rfc4287) in conjunction with the [Atom Threading Extension](http://www.ietf.org/rfc/rfc4685.txt) to expose model resources, and the HTTP-based [Atom Publishing Protocol (APP)](http://tools.ietf.org/html/rfc5023) to modify portal resources. With the Remote Model SPI as a REST service, you must use the following HTTP verbs:

* To obtain model information: HTTP GET
* To modify existing model information: HTTP PUT
* To create model resources: HTTP POST
* To delete model resources: HTTP DELETE
* [**Feeds for REST services**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/rest_feed.html)  
  When you access a REST service to get information or to modify a portal resource, the response and in some cases also the request works by means of a feed. A feed contains information about one or more portal resources in a specific format as exposed by portal models. Learn how you obtain feeds for portal resources and what the format of such feeds is.

**Navigational State SPI**

Navigational state represents the view of the portal that is associated with a particular client. The client can request (query) different views by interacting with the Web page, for example by navigating to a new page. This interaction does not change the state on the server but requests a new view from the server. It is therefore a “safe” operation in terms of HTTP 1.1 (See RFC 2616, Section 9.1.1). This allows the client to navigate back and forward through its recent views, bookmark views, and go back to them by invoking a browser bookmark. This behavior is achieved by encoding the navigational state of IBM® WebSphere® Portal into the URL. Different navigational states result in different URLs.

For details about the back button behavior and the support of bookmarks refer to the topic about Back button behavior.

This section introduces the Navigational State SPI, in particular how the SPI is used in order to create URLs carrying navigational state. The SPI offers two services (obtainable via JNDI) to create such URLs:

* The ***Portal* State Manager Service** is a portal service that is used to realize use cases that go beyond what the portal tags provide. In particular, it can be used to create URLs within portal artifacts such as themes and skins, for example in custom JSP tags, as well as in artifacts that are removed from the request processing, for example Enterprise JavaBeans. However, the Portal State Manager Service is not meant to be used for portlets.
* The ***Portlet* State Manager Service** is the counterpart service supporting JSR168 and JSR286 compliant portlets. Accordingly, use the Portlet State Manager Service to create URLs within portlets that cannot be created using the Standard Portlet APIs.

The com.ibm.portal.state package is the main package of the Navigational State SPI holding the service interfaces as well as the interfaces making up the navigational state object model. For more information about all SPI interfaces, refer to the Javadoc.

# TOPIC 1:- Object Model

Learn about the main object models used in the navigational state SPI.

**Navigational state**

The Navigational state object model is a hierarchy of state information, which is modeled as a DOM-like document containing untyped state information represented as strings because a typical page contains many URLs that require the navigational state to be serialized multiple times per request. The string-based memory allows for efficiently serializing navigational states into URLs, because it avoids processing time and CPU consuming object to string conversions during the serialization process.

The Navigational state is modeled via the following interfaces:

* com.ibm.portal.state.StateHolder
* com.ibm.portal.state.StateHolderController
* com.ibm.portal.state.dom.DocumentModel
* com.ibm.portal.state.dom.DocumentController

The StateHolder interface provides read access to the navigational state information by exposing a getModel() method that returns an interface to the untyped document model, namely the DocumentModel interface. The DocumentModel interface provides a set of methods to inspect both the hierarchical structure of the state document as well as single nodes being part of that document.

In addition, the StateHolder interface offers a newState() method which can be used to create a clone of that particular StateHolder instance. This is an important method because, typically, the current navigational state has to be cloned before modifying it according to the specific semantics of the generated URL. For example if the programmer wants to create a URL that points to a particular portal page, the page selection information in the state document model has to be changed; however, this modification must not take effect on all URLs of a page.

Beyond that, the StateHolderController interface also allows you to modify state information by offering a getController() method that delivers a controller interface to the untyped state document called DocumentController. The DocumentController interface provides a means to modify the hierarchical structure of the state document, which includes creating document nodes, inserting them into the node hierarchy, and removing nodes.

**Engine URLs**

URLs are modeled via the com.ibm.portal.state.EngineURL interface. An EngineURL represents a URL that contains navigational state. The initial StateHolder an EngineURL refers to is specified when requesting a new EngineURL instance from the appropriate URL factory; see [URL generation services](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/url_gen_serv.html) for additional information. Typically it is a copy of the request-specific base state.

The EngineURL interface provides the following methods:

**StateHolderController getState()**

Returns a read-write interface to the navigational state carried by the URL.

**void setProtected(Boolean flag)**

Specifies whether the URL should point to the public or protected area.

**void setSecure(Boolean flag)**

Specifies whether a secure https connection is required.

**Writer writeCopy(Writer out)**

Streams the URL to the given writer. Maintains the state of the URL. For example this method can be used to write the URL multiple times.

**Writer writeDispose(Writer out)**

Streams the URL to the given writer and finally releases the state of the URL. The EngineURL object must not be accessed again after invoking this method.

**void dispose()**

The dispose() method is inherited from the Disposable interface. It must be invoked to indicate that the EngineURL object is no longer needed. The EngineURL object must not be accessed again after invoking this method. Alternatively, you can invoke the writeDispose() method, which calls dispose().

The crucial method is the getState() method, which returns the state holder object this particular EngineURL instance refers to.

**Note:** The method returns a controller interface (StateHolderController) that allows the programmer to modify the state of this EngineURL. See [Accessor SPI](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/accessor_spi.html) for additional information about modifying the state.

**Resource URLs**

URLs that address generic resources such as, but not limited to, files, icons, and voice grammars cannot contain navigational states and are therefore modeled using a separate interface called com.ibm.portal.state.DisposableURL. The DisposableURL interface almost offers the same methods as EngineURL but does not provide a getState() method.

Resource URLs can also be created using the URL generation Services that are offered along with the Navigational State SPI. See [URL generation services](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/url_gen_serv.html) for additional information.

# TOPIC 2:- Accessor SPI

The Accessor SPI provides typed access to the state document model. It allows the programmer to query and modify navigational state information. The Accessor SPI is part of the package com.ibm.portal.state.accessors.\*.

The Accessor SPI is an abstract layer surrounding the access to particular nodes in the hierarchical document model; see [Object Model](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/obj_model.html) for additional information about the hierarchical document model. For each state aspect such as, but not limited to, page selection, expansion states, and portlet states, the SPI offers an accessor factory that provides read-only and read-write accessor controls that are designed for the particular state aspect they refer to. The accessors read from or write to the respective positions in the state document model and perform required type conversions.

The navigational state information is located in the state document model and is available when the accessor factory is used. After the node is located, the accessor factory passes a node reference to the accessor or accessor controller during. The accessor and accessor controller are independent from the state document model structure; you can reuse accessors even if the information is moved to another node in the state document.

For example, the SelectionAccessorFactory offers the following interfaces:

**SelectionAccessor getSelectionAccessor(StateHolder)**

This method returns a SelectionAccessor interface that allows for reading page selection information from the given StateHolder.

**SelectionAccessorController getSelectionAccessorController(StateHolderController)**

This method returns a SelectionAccessorController interface that allows programmers to modify page selection information. The controller uses the StateHolderController interface to modify the navigational state accordingly.

The flyweight pattern – the StateHolder or StateHolderController is used as an argument – is commonly used in the accessor factory interfaces. The navigational state the accessor operates on may not be the base state retrieved from the request URL; typically, it is the state clone created for a particular EngineURL. Call getState() on the EngineURL object to obtain the URL-specific state holder. The following example shows how to let a created EngineURL point to a certain portal page (for example, the “Stock Market” page) using the SelectionAccessorController:

final EngineURL url = ...;

final SelectionAccessorFactory selectionFct = ...;

final SelectionAccessorController selectionCtrl =

selectionFct.getSelectionAccessorController(url.getState());

try {

selectionCtrl.setSelection(“wps.StockMarket”);

} catch (StateException e) {

// include error handling here

} finally {

selectionCtrl.dispose();

}

**Using the SelectionAccessorController to create a page link**

The base Accessor interface is derived from the com.ibm.portal.Disposable interface. Invoke the dispose() method to indicate when the accessor is no longer required. Using the dispose() method allows the accessor factory to store the accessors and accessor controllers in object pools to achieve better performance (due to less initialization and garbage collection overhead).

The Navigational State SPI offers the following accessor factories, each covering a certain state aspect:

**SelectionAccessorFactory**

The SelectionAccessorFactory provides accessors to read and write portal page selection information. To create a URL that points to another page, the SelectionAccessorController needs to be requested from the factory in order to include the new selection into the state the created EngineURL is associated with

**PortletAccessorFactory**

The PortletAccessorFactory provides accessors to read and write portlet-related navigational state information, which includes portlet mode, window state, and render parameters. In particular the PortletAccessorController can be used to change the navigational state of a given portlet (for example, the portlet mode).

**PortletTargetAccessorFactory**

The PortletTargetAccessorFactory provides accessors to read and write portlet action-related information. In particular the PortletAccessorController can be used to declare a portlet as the target of an action. This allows the programmer to create URLs that trigger portlet actions.

**SoloAccessorFactory**

The SoloAccessorFactory provides accessors to read and write the so-called Solo state. If the portal is in Solo state, it renders only one particular portlet of the current portal page; all navigation controls and tool bars are hidden. The SoloAccessorController can be used to create URLs that activate/deactivate the Solo state for a particular portlet.

**ThemeTemplateAccessorFactory**

The ThemeTemplateAccessorFactory supports reading and writing theme template information. In particular the ThemeTemplateAccessorController can be used to create URLs that switch to a certain theme template.

**LocaleAccessorFactory**

The LocaleAccessorFactory provides accessors to read and write locale information. The LocaleAccessorController can be used to set a special locale into the navigational state and thus into a URL.

**Note:** A locale retrieved from such a URL takes precedence over user preferred locales or locales defined on the user’s browser.

**ExpansionStatesAccessorFactory**

The ExpansionStatesAccessorFactory provides accessors to read and write expansion states information; for example, to determine whether a given navigation node in a navigation tree control is expanded or collapsed. The ExpansionStatesAccessorController is typically used to generate URLs that toggle the expansion state of a navigation node.

**ShowToolsAccessorFactory**

The ShowToolsAccessorFactory provides accessors to read and write tool-related information. The ShowToolsAccessorController is typically used to create a URL that blends in the tool icons for portlet windows offering function such as moving/deleting the respective portlet.

**StatePartitionAccessorFactory**

The StatePartitionAccessorFactory provides accessors to read and write state partition identifiers. The StatePartitionAccessorController can be used to include a state partition identifier into the navigational state. A new state partition identifier should be included into URLs that open new browser windows or iFrames.

**EngineActionAccessorFactory**

The EngineActionAccessorFactory provides controllers that should be used to create engine action URLs. The EngineActionAccessorController, in particular, allows you to set action parameters.

# TOPIC 3:- URL generation services

Learn about the services used to create URLs in the navigational state SPI.

The Navigational State SPI offers the following two services that create URLs that are accessible via the Java Naming and Directory Interface (JNDI):

* The com.ibm.portal.state.service.PortalStateManagerService is a portal service that provides services beyond what the portal tags provide. Use the PortalStateManagerService to create URLs within portal artifacts such as themes and skins (for example custom JSP tags) and within server-side artifacts that are removed from the request processing (for example Enterprise JavaBeans). This service is for portals only.
* The com.ibm.portal.portlet.service.PortletStateManagerService supports JSR 168 and JSR 286 compliant portlets. Use the PortletStateManagerService to create URLs within portlets that cannot be created using the Standard Portlet API. This service is for portlets only.

Both services use a common interface called com.ibm.portal.state.service.StateManagerService, which provides the functions that are common to both services. You can reuse URL generation code in themes, skins, and portlets if they use the common interface. For example, you can use a custom JSP tag, which was originally designed for use in themes, in a portlet-specific JSP because only the service lookup is different.

**Getting access to the PortalStateManagerService**

Access the PortalStateManagerService through a JNDI lookup using the lookup name “portal:service/state/PortalStateManager”. The lookup returns a com.ibm.portal.state.service.PortalStateManagerServiceHome interface that provides the following two get commands that retrieve com.ibm.portal.state.service.PortalStateManagerService:

**PortalStateManagerService getPortalStateManagerService( HttpServletRequest request, HttpServletResponse response)**

The returned service is suitable for creating URLs in themes, skins, and custom JSP tags. All server-related information that is needed to generate the URLs (such as host name, protocol, server port, context path etc.) is retrieved from the given servlet request.

**PortalStateManagerService getPortalStateManagerService( ServerContext ctx, Locale locale, Profile profile, boolean isProtected, boolean isSecure)**

This method returns a PortalStateManagerService used for “offline” use cases such as creating URLs in environments where the servlet request is not available (for example, in an Enterprise JavaBean). Using this method requires that the server-related information (such as host name, protocol, server port, context path etc.) is provided in the ServerContext argument. The com.ibm.portal.state.accessors.url.ServerContext interface is the required abstraction for this kind of information. If you want to create URLs that point to a certain virtual portal, the provided ServerContext object needs to address that virtual portal either via the context path or by host name. The two boolean arguments specify whether the URLs should point to the protected area by default (isProtected) and whether the created URLs should be served using a secure connection by default (isSecure). To support creating resource URLs, the arguments locale and profile have to be provided. The locale argument is needed to create URLs that address locale-specific resources such as language-dependent icons. The profile argument represents the client profile and in particular allows for checking the capabilities of the client.

The lifetime of a PortalStateManagerService object depends on whether you requested a “request-specific” service or “offline” service. The request-specific service has request scope; it is only used for the duration of one servlet request. For all subsequent servlet requests, request a new service instance from the home interface. The lifetime of the offline PortalStateManagerService corresponds with the lifetime of the ServerContext object.

The obtained PortalStateManagerServiceHome object is valid for the lifetime of the portal. You should perform the JNDI lookup only once and store the retrieved home object accordingly, for example in an instance or static variable.

The following example shows how to get access to the service:

/\*\* the JNDI name to retrieve the PortalStateManagerServiceHome object \*/

private static final String JNDI\_NAME =

"portal:service/state/PortalStateManager";

/\*\* PortalStateManagerServiceHome object to retrieve the service from \*/

private static PortalStateManagerServiceHome serviceHome;

public void **useRequestService**(final HttpServletRequest request,

final HttpServletResponse response) {

try {

// get the request-specific service from our home interface

final StateManagerService service

= getServiceHome().getPortalStateManagerService(request, response);

// use the service

// ...

// indicate that we do not need it any longer

service.dispose();

} catch (Exception e) {

// error handling

}

}

public void **useOfflineService**(final ServerContext ctx,

final Locale locale,

final Profile profile,

final boolean prot,

final boolean secure) {

try {

// get the offline service from our home interface

final StateManagerService service

= getServiceHome().getPortalStateManagerService(ctx, locale, profile, prot, secure);

// use the service

// ...

// indicate that we do not need it any longer

service.dispose();

} catch (Exception e) {

// error handling

}

}

/\*\*

\* Looks up the PortalStateManagerServiceHome being valid

\* for the lifetime of the portal.

\*/

private static PortalStateManagerServiceHome **getServiceHome**() {

if (serviceHome == null) {

try {

final Context ctx = new InitialContext();

serviceHome =

(PortalStateManagerServiceHome) ctx.lookup(JNDI\_NAME);

} catch (Exception e) {

// error handling

}

}

return serviceHome;

}

**Note:** The PortalStateManagerService interface is derived from the generic StateManagerService interface. As the StateManagerService extends the com.ibm.portal.Disposable interface, you should indicate when you do not need to access the service any longer by invoking the offered dispose() method on it. Dispose the request-specific PortalStateManagerService by the end of the processed servlet request.

**Getting access to the PortletStateManagerService**

Access the PortletStateManagerService via JNDI using the lookup name "portletservice/com.ibm.portal.state.service.PortletStateManagerService". The lookup returns a generic com.ibm.portal.portlet.service.PortletServiceHome interface which offers a getPortletService(Class) method to get the PortletStateManagerService. The retrieved PortletStateManagerService instance is valid for the lifetime of the portal. You should perform the service retrieval in the init method of the portlet and store it in a portlet instance variable. The following example shows the required lookup code:

public class MyPortlet extends GenericPortlet {

/\*\* The JNDI name which is needed to lookup the service \*/

private static final String JNDI\_NAME =

"portletservice/com.ibm.portal.state.service.PortletStateManagerService";

/\*\* portlet state manager service \*/

protected PortletStateManagerService service;

/\*\*

\* @see javax.portlet.GenericPortlet#init()

\*/

public void init() throws PortletException {

super.init();

try {

// lookup the portlet state manager service

final Context ctx = new InitialContext();

final PortletServiceHome serviceHome = (PortletServiceHome)

ctx.lookup(JNDI\_NAME);

service = (PortletStateManagerService)

serviceHome.getPortletService(PortletStateManagerService.class);

} catch (NameNotFoundException e) {

throw new PortletException(e);

} catch (NamingException e) {

throw new PortletException(e);

}

}

You can use the actual service within the render method of the portlet (or in the helper methods serving the mandatory portlet modes such as doView, doEdit, and doHelp) to include URLs into the markup or in the processAction method in order to send a redirect to a certain URL. The PortletStateManagerService interface exposes the following two methods:

**PortletStateManager getPortletStateManager( PortletRequest request, PortletResponse response)**

This method returns a PortletStateManager object which you can use during action processing and rendering (for example in the processAction method, doView method, doEdit method, etc.). The PortletStateManager interface adds additional methods to extend the generic StateManagerService interface, which, for example, allows you to directly read the current request-specific navigational state of the portlet (portlet mode, window state, and render parameters).

**PortletStateManagerController getPortletStateManagerController( ActionRequest request, Action response)**

This method returns a PortletStateManagerController that extends the PortletStateManager interface. The PortletStateManagerController provides additional methods that allow for modifying the navigational state of the portlet for the current request and is therefore only accessible during action processing (for example, in the processAction method of the portlet).

Both the PortletStateManager and the PortletStateManagerController have request scope, which means that you must not store references to them across requests. Instead you must retrieve the service from the PortletServiceHome object. To indicate that the retrieved PortletStateManager or PortletStateManagerController instance is no longer accessed in the scope of a request, you must invoke the dispose method inherited from the Disposable interface on it. The following example shows you how to retrieve the service from the PortletServiceHome object:

/\*\*

\* @see javax.portlet.GenericPortlet#doView(RenderRequest, RenderResponse)

\*/

protected void doView(

final RenderRequest request, final RenderResponse response)

throws PortletException, IOException {

response.setContentType(request.getResponseContentType());

final PrintWriter writer = response.getWriter();

try {

// get the request-specific portlet state manager

final PortletStateManager mgr = service.

getPortletStateManager(request, response);

// do something (create URLs etc.)

// ...

// indicate that we do not need the portlet state manager any longer

mgr.dispose();

} catch (StateException e) {

throw new PortletException(e);

}

}

**The base interface StateManagerService**

The PortletStateManager and the PortalStateManagerService are derived from the com.ibm.portal.state.service.StateManagerService interface, which offers functionality that is common to both URL generation services. The use cases that are common to both services refer to the creation of EngineURLs that carry navigational state and resource URLs. Therefore, the StateManagerService interface should be sufficient to implement most of the use cases. The interface provides the following two methods:

**URLFactory getURLFactory()**

This method returns a com.ibm.portal.state.URLFactory object that offers several methods to create a variety of URLs. For further details on the URLFactory.

**AccessorFactory getAccessorFactory(Class accessorFactoryClass)**

This method provides access to the various accessor factories; see [Accessor SPI](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/accessor_spi.html) for additional information. You must pass the respective accessor factory interface in as a method argument to retrieve a certain accessor factory the Class object.

**Note:** When using the PortletStateManagerService, the set of accessor factories is restricted to the SelectionAccessorFactory, PortletAccessorFactory, PortletTargetAccessorFactory, SoloAccessorFactory, ThemeTemplateAccessorFactory, LocaleAccessorFactory, StatePartitionAccessorFactory, and ExpansionStatesAccessorFactory.

Typically, you request an EngineURL object from the URLFactory and then modify the navigational state according to the required URL semantics. To perform this step, call the getState() method of the EngineURL to get the StateHolderController object that is required to modify the navigational state via the Accessor SPI. The following code snippet exemplarily illustrates this typical usage pattern. The following example shows how to call the getState() method of the EngineURL:

protected EngineURL createPageLink(final ObjectID pageID)

throws StateException {

// get the URL factory from the state manager service

final URLFactory urlFactory = service.getURLFactory();

try {

// get a EngineURL from the factory; maintain navigational state

final EngineURL url = urlFactory.newURL(null);

final SelectionAccessorFactory selectionFct = (SelectionAccessorFactory)

service.getAccessorFactory(SelectionAccessorFactory.class);

// get a selection controller that operates on the URL-specific state

final SelectionAccessorController selectionCtrl =

selectionFct.getSelectionAccessorController(url.getState());

try {

// modify page selection and return URL

selectionCtrl.setSelection(pageID);

return url;

} finally {

selectionCtrl.dispose();

}

} finally {

urlFactory.dispose();

}

}

In the above code sample, the newURL(Constants.Clone) method of the URLFactory is used. However, the URLFactory offers several additional convenience methods to create EngineURLs as well as resource URLs. The following list provides a complete description of the URLFactory interface:

**EngineURL newURL(Constants.Clone type)**

Choose this method to create a EngineURL because it is suitable for all prevalent use cases. The method provides an EngineURL, which refers to the StateHolder representing the navigational state of the current request. The type argument specifies how the request-specific StateHolder should be cloned for the URL to be created. There are the following four pre-defined clone constants:

* SMART\_COPY indicates that a shallow StateHolder copy should be created which records the state modifications applied for this particular EngineURL only, instead of copying all nodes in the document model to construct the clone. The SMART\_COPY represents the default; therefore passing in null is equivalent to SMART\_COPY. This clone method also allows the generation of relative “delta” URLs.
* DEEP\_COPY results in a complete copy of the request-specific StateHolder, for example each node and the node hierarchy is cloned. The deep copy prevents the generation of delta URLs.
* EMPTY\_COPY indicates that the contents of the request-specific StateHolder should be cleared, for example the created EngineURL is based on an empty state. Any interaction with such a URL results in the lost the navigational state of previous interactions.

If the URLFactory does not have access to the current request a new empty StateHolder is created internally. In that case the given type argument does not take any effect.

**EngineURL newURL(StateHolder state, Constants.Clone type)**

This second getURL method requires that the StateHolder the EngineURL is based on is passed in explicitly. Accordingly, the type argument refers to this particular StateHolder. Use this method when the URL to be created should encode a StateHolder that was defined programmatically. Clicking on this URL results in the lost the navigational state of previous interactions with the portal.

**EngineURL newURL(URLContext ctx, Constants.Clone type)**

This variant allows you to specify whether the created URL should be absolute, server-relative, or relative. This can be achieved by passing in an URLContext interface which has to be implemented accordingly; for example if the URL must be absolute, the isAbsolute() method must return true whereas isRelative() and isServerRelative() must return false. Because this method does not require an explicit StateHolder argument, the EngineURL to be created will encode the StateHolder retrieved from the given request. To reduce markup size it is strongly recommended to pass in an URLContext that allows for relative URLs . If the URLFactory does not have access to the current request, a new empty StateHolder is created internally. In that case the given type argument does not take any effect.

**EngineURL newURL(URLContext ctx, Constants.Clone type)**

This method is the counterpart of the previous method and takes a StateHolder defined as an explicit argument.

**DisposableURL newResourceURL( String name, PortalResources.Type type)**

This method creates a URL that points to a generic resource. The filename and resource type identify the resource. The resource lookup takes request-specific information such as the current locale, the client device, and the markup name into account (if the request is available). The com.ibm.portal.state.accessors.url.PortalResources interface provides several useful constants for the resource type representing resource types such as files, sounds, icons, voice grammars, etc.

**DisposableURL newResourceURL( String name, PortalResources.Type type, PortalResources.State state)**

This method creates a URL that points to a generic resource. The filename and resource type identify the resource. It can also contain a resource state that is used to further distinguish the lookup of the resource. The resource lookup takes additional information from the request into account (if available). This is the current locale, the client device, markup chosen for the client, and the theme name.

**Note:** An absolute URL is a complete URL containing protocol, host name and port. In the case of a server-relative URL, the browser implies the current protocol, host name and port. In the case of a relative URL, the browser appends the URL to either the current request URL or to the value of the HTML base tag (if any). Server-relative and relative URLs cannot be enforced. In case of a protocol switch from “http” to “https”, for example, the generated URL is absolute in any case.

**Changing the host name of absolute URLs**

You can change the host name of absolute URLs for security reasons. For example, this can be the case if the DOM of an application, such as a portlet, runs within an iframe and you do not want the JavaScript code within that iframe to be able to access the DOM of the HTML document itself. If all URLs within that iframe are absolute URLs and the host name of these URLs is different from the one from which the document originates, then the iframe has access to itself only. In other words, it can neither manipulate nor access the rest of the document DOM. To ensure this, create all URLs within the portlet by using the Navigational State SPI with absolute URLs. The portlet should then define a **virtual host name**, which must be rerouted by a proxy to the portal server again. In addition the portlet must ensure that all requests in which the desired absolute URLs are to be generated contain a special request header that tells the portal the name of the virtual host name. The headers are as follows:

**com.ibm.lotus.openajax.virtualhost**

Use this for setting the host name of every generated absolute URL to the value of this request header.

**com.ibm.lotus.openajax.virtualport**

Use this for setting the port of every generated absolute URL to the value of this request header.

**Controller SPI**

You can use the Controller SPI for portal administration. It allows you to modify portal resources. It enhances the read-only portal Model SPI by adding writable aspects.

The interfaces of the programming model for portal resources that were published previously under the [Model SPI overview](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/dgn_modelovw.html) offered only read-only methods. The Controller SPI extends them by adding a set of new interfaces. These interfaces are derived from the read-only portal models and interfaces and map to them, but they also extend them with methods for modifying the resources that they represent. This way the Controller SPI allows you to modify portal resources to a certain extent.

**Controller SPI overview**

The Controller SPI provides controllers for portal resources. You can use these controllers to modify portal resources that are exposed by particular models of the Model SPI. Controllers offer methods to modify the topology and properties of the model and of its nodes. They expose the same interfaces as the corresponding read-only model, and they instantly reflect modifications that you apply to the controller.

**Note:** While the modifications come into effect immediately for the controller, they not reflected in the persistence layer until you commit the controller and the changes that you made by it.

The resources that are exposed by the controller can be modified through specific interfaces which match their read-only counterparts. For an example, refer to the following class list:

|  |  |
| --- | --- |
| **Classes from the read-only model . . .** | **. . . are reflected in the Controller SPI by the following classes** |
| ContentNode   * MarkupCapable * Localized * Identifiable * ActiveFlag | **Modifiable**ContentNode   * **Modifiable**MarkupCapable * **Modifiable**Localized * **Modifiable**Identifiable * **Modifiable**ActiveFlag |

Further benefits of the Controller SPI are as follows:

* Controller instances work as workspaces where you make your modifications. You can try your modifications and assess them in a preview mode. When you are happy with your changes, you apply them to the portal by a commit step.
* You can make and apply your changes to a running portal environment. You do not need to restart the portal for the changes to take effect.

The Controller SPI provides the following controllers:

**Content Model Controller**

This allows you to modify the content topology and the properties of content nodes such as pages, labels, and content URLs.

**Note:** If you modify the content topology, this might also change the navigation of your portal for your users.

**Layout Model Controller**

This allows you to modify the layout of a page, such as the topology of layout elements of a page, and the properties of layout elements such as layout containers and layout controls.

**Portlet Model Controller**

This allows you to create, update and delete portlets.

**Note:** At this time there is no controller for the following models:

* NavigationModel. This is by implication of the structure of the content model.
* NavigationSelectionModel. This is computed from the navigational state per request.
* LanguageList, MarkupList, SkinList, and ThemeList.

A controller is based on the corresponding read-only model. This means when you first create the controller on the basis of a read-only model, both the controller and the model expose the same information. You can then use the controller to create, update, or delete information exposed through it. These changes will be reflected in the controller immediately. To persist changes that you made to the underlying read-only model, you need to commit the controller.

In particular, a controller offers methods to do all of the following:

* Provide modifiable instances of existing resources. Such modifiable instances exist for each modifiable property of the resources, and they allow the modification of these properties.
* Create and delete model resources.
* Obtain dependent controllers. This is optional. For example, a ContentModelController offers a method to obtain a LayoutModelController.
* Persist the modifications.

**Note:** Before you use the Controller SPI, be sure to familiarize yourself with the read-only models. Refer to [Model SPI overview](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/dgn_modelovw.html).

**Scope of the Controller SPI**

A controller instance is based on a read-only model instance. Therefore it has the same scope and lifetime as the corresponding read-only model. Consequently, the following equivalences apply:

* If the underlying model is scoped to a particular user, then so is the controller.
* If the underlying model is scoped to a request, then so is the controller.
* If the underlying model is scoped to a virtual portal, then so is the controller.
* [**Packages of the Controller SPI**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapic_pckg.html)  
  The portal provides the SPI Controller in several separate packages.
* [**Working with controllers**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_wrk.html)  
  When you modify a portal resource with the Controller SPI, you go through a set of steps.
* [**Making modifications by using the Controller SPI**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_mdfy.html)  
  The Controller SPI allows you to modify portal resources, the topology of your portal, and properties.
* [**Confirming modifications**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_cnf_mod.html)  
  Modifiable interfaces and Controller interfaces provide methods for confirming modifications. You can use these confirm methods to check whether modifications might be prevented by portal internal constraints.
* [**Hints and tips for using the Controller SPI**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapic_hint.html)  
  These are some hints and tips for using the Controller SPI.

# TOPIC 1:- Packages of the Controller SPI

The portal provides the SPI Controller in several separate packages.

These packages are as follows:

* **com.ibm.portal**. This package holds the following interfaces:
  + Base interfaces for the Controller SPI, for example Modifiable and Controller.
  + Modifiable interfaces, such as ModifiableActiveFlag, ModifiableLocalized, ModifiableMetaData.
* **com.ibm.portal.admin**. This package holds modifiable interfaces, for example the ModifiableMarkupCapable and the LanguageListController.
* **com.ibm.portal.content**. This package holds the following interfaces:
  + Interfaces for content and layout model controller modifiable interfaces for content and layout nodes. For example, these can be ModifiableContentPage or ModifiableLayoutContainer.
  + Modifiable interfaces for modifiable aspects of content and layout nodes, for example ModifiableBookmarkableFlag
  + Creation contexts for creating content pages and layout nodes, for example LayoutContainerCreationContext.
* **com.ibm.portal.model.controller**. This package holds home and provider interfaces to obtain controllers, including a builder factory for CreationContext instances.
* **com.ibm.portal.model.controller.exceptions**. This package holds controller specific exceptions.
* **com.ibm.portal.portlet**. This package holds the following interfaces:
  + Interfaces for the portlet model controller.
  + Modifiable interfaces for portlet definitions, entities and preferences, for example ModifiablePortletPreferences and ModifiablePortletEntity.
  + Portlet creation and cloning context interfaces.

# TOPIC 2 :- Working with controllers

When you modify a portal resource with the Controller SPI, you go through a set of steps.

For example, these steps can be as follows:

1. Obtain the appropriate controller.
2. Obtain a modifiable instance of the resource that you want to modify from the controller.
3. Apply your modifications as required to the modifiable instance Commit the controller to persist the modifications.
4. Optionally repeat steps [2](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_wrk.html#ctrlrapit_wrk__ctrlrapit_wrk_2) and [3](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_wrk.html#ctrlrapit_wrk__ctrlrapit_wrk_3) above as required.
5. Commit the controller to persist the modifications. This step saves and applies the modifications to your portal.
6. **Obtaining a controller for working with resources**
7. To modify, create, or delete portal resources by using the Controller SPI, you first need to create a controller.
8. You do this by using a JNDI based lookup for the correct "home" interface, that is, the corresponding read-only interface.
9. **Note:** The provider lookup for a controller home is possible from servlet level code and portlets.
10. The following controllers are available via JNDI:
11. **ContentModelController**
12. To obtain a ContentModelController, perform a lookup for the string ContentModelControllerHome.CONTENT\_MODEL\_CONTROLLER\_JNDI\_NAME.
13. **PortletModelController**
14. To obtain a ContentModelController, perform a lookup for the string PortletModelControllerHome.PORTLET\_MODEL\_CONTROLLER\_JNDI\_NAME.
15. The **LayoutModelController** cannot be obtained via a JNDI lookup. You obtain it through its associated **ContentModelController**.
16. **Example - Obtaining a content model controller:**
17. ContentModelController result = null;
18. final Context ctx = new InitialContext();
19. final ContentModelControllerHome home = (ContentModelControllerHome)
20. ctx.lookup(ContentModelControllerHome.CONTENT\_MODEL\_CONTROLLER\_JNDI\_NAME);
21. if (home != null) {
22. result = home.getContentModelControllerProvider().createContentModelController(aContentModel);
23. }
24. **Note:** To obtain a ContentModelController, you must pass an existing content model to the createContentModelController method of the ContentModelControllerProvider.
25. **Example 2 - Obtaining a layout model controller for a specific page:**
26. // locate the page for which you want to create a LayoutModelController
27. final Locator locator = cmController.getLocator();
28. final ContentPage page = (ContentPage) locator.findByUniqueName("MyPage");
29. // create a LayoutModelController
30. final LayoutModelController lmController = cmController.getLayoutModelController(page);

**Committing and persisting your modifications**

To persist the modifications that you applied to the controller, you commit the controller.

You can commit only the ContentModelController and the PortletModelController as only these two implement the Committable interface. You cannot commit a LayoutModelController. Committing the LayoutModelController is included when you commit the ContentModelController from which you obtained the LayoutModelController. Committing the ContentModelController can include committing more than one LayoutModelController.

1. Committing content model controller

| | 2. commits | layout model |

| content model | ---------> | controller |

| controller | |

| | 3. commits | layout model |

| | ---------> | controller |

| | |

| | 4. commits | layout model |

| | ---------> | controller |

**Notes:**

1. After you have successfully committed a controller, you must not use it any more. In particular, do not invoke the commit() method again at a later stage.
2. After you have used a controller, you always need to dispose it.
3. If you do not want to persist the changes made by using the controller, dispose it without invoking commit() on it.

The following example shows how you commit a controller and then dispose it:

// commit the controller

try {

controller.commit();

} finally {

controller.dispose();

}

**Modifying portal resources and topologies**

The Controller SPI allows you to modify portal resources and the topology of your portal in different ways.

You can make the following modifications:

* Create new resources
* Insert new resources into the portal topology
* Move existing resources
* Delete existing resources.

Depending on the resource that you create or modify, you can perform different types of modifications. For more detailed information about each of these refer to the following topics.

* [**Creating resources**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_crt_rsrc.html)  
  You create resources by using methods of controllers. Each controller type allows you to create specific resources for its type of model.

**Obtaining creation contexts**

You need a creation context to define immutable properties of a resource that you create. You can use the creation context builder factory to generate multiple such creation contexts without having to implement those interfaces directly.

The creation context builder factory can do both of the following:

* Generate single creation contexts, that is contexts that contain only one or more immutable properties. Examples:
  + A creation context for an object ID contains only the object ID property.
  + A layout control creation context contains two properties, portlet definition and portlet entity.
* Combine several creation contexts into one in order to define multiple immutable properties.

**Note:** You can only combine creation contexts that have not already been combined by using the creation context builder.

**Example 1 - Obtaining a simple creation context:**

// obtain creation context builder

final CreationContextBuilderFactory builder = CreationContextBuilderFactory.getInstance();

// obtain creation context

final CreationContext creationContext = builder.newIdentifiableCreationContext(objectID);

**Example 2 - Obtaining a combined creation context:**

// obtain creation context builder

final CreationContextBuilderFactory builder = CreationContextBuilderFactory.getInstance();

// obtain combined creation context

final CreationContext creationContext = builder.combine(new CreationContext[]

{builder.newContentPageCreationContext(true), builder.newIdentifiableCreationContext(objectID)});

The following list describes creation contexts that you can create by using the creation context builder factory:

**ContentPageCreationContext**

Use this creation context to define whether a page that you create is private. This applies to the resource type ContentPage on the ContentModelController.

**DerivedContentPageCreationContext**

Use this creation context to define the derivation parent of a page that you want to create. This applies to the resource type ContentPage on the ContentModelController.

**Note:** This creation context derives from the ContentPageCreationContext.

**IdentifiableCreationContext**

Use this creation context to define an object ID for a resource that you want to create. This applies all resource types: ContentModelController, LayoutModelController, PortletModelController.

**PortletDefinitionCloningContext**

Use this creation context to define the portlet definition ID, and optionally, the domain for the portlet definition that you want to clone. This applies to the resource type PortletDefinition on the PortletModelController.

**PortletEntityCreationContext**

Use this creation context to define the parent ID, and optionally, the domain for the portlet entity that you want to create. This applies to the resource type PortletEntity on the PortletModelController.

* [**Moving or Inserting Nodes**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_mvnsrt_nodes.html)  
  To move existing nodes or insert new nodes, use the insert() method of the controller.

**Placing a portlet on a page**

To put a portlet on a page, use the insert() method of the controller.

To place a portlet on a page, proceed by the following steps:

1. Obtain a layout model controller for the page the portlet is to be put on.
2. Obtain a LayoutControlCreationContext; this includes obtaining the portlet definition ID, and optionally the portlet entity ID.
3. Create a layout control by using the context from above and by using the create() method of the layout model controller.
4. Insert the created layout control into the layout model controller.
5. Persist your modifications by using the commit() method of the controller.

**Example 1 - Placing a portlet on a page:**

// obtain layout model controller

final LayoutModelController lmController = cmController.getLayoutModelController(page);

// obtain layout control creation context

final LayoutControlCreationContext context = ... (portletDefinition, null);

//final LayoutControlCreationContext context = ... (portletDefinition, portletEntity);

// create layout control

final Modifiable control = lmController.create(LayoutControl.class, context);

// insert control into the topology of the layout model controller (given a container and a sibling)

lmController.insert(control, container, sibling);

// commit the content model controller

cmController.commit();

* [**Deleting Nodes**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_del_nodes.html)  
  To delete nodes, use the delete() method of the controller.

**Deleting Nodes**

To delete nodes, use the delete() method of the controller.

To delete a node, proceed by the following steps:

1. Obtain an appropriate controller.
2. Locate the resource that you want to delete. To do this, use an appropriate locator of the controller, or search the model by iterating until the required node is found.
3. Delete the resource by using the delete() method of the controller.
4. Persist your modifications by using the commit() method of the controller.

**Example 1 - Deleting a node** (error handling omitted):

// obtain locator of ContentModelController

final Locator locator = cmController.getLocator();

// locate page to delete

final ContentPage page = (ContentPage) locator.findByUniqueName("MyPage");

// delete the page

cmController.delete(page);

// commit the controller

cmController.commit();

# TOPIC 3 :- Modifying properties

The Controller SPI allows you to modify properties resources.

To modify properties of a resource, proceed by the following steps:

1. Obtain the appropriate controller for the model of which you want to modify a resource.
2. Obtain a modifiable instance of the resource from that controller.
3. Invoke the appropriate methods of the modifiable instance to perform the modifications.
4. Persist the modifications by using the commit() method of the controller.

* [**Setting titles and descriptions**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_set_title.html)  
  You can set the titles and descriptions of a resource.

To do this, proceed by the following steps:

1. Obtain a modifiable instance of the resource for which you want to set titles or descriptions.
2. Check whether the resource implements the ModifiableLocalized interface. To do this, use the operator instanceof. If the resource does not implement the ModifiableLocalized interface, you cannot modify it.
3. Use the appropriate methods to set titles and descriptions. For example, if you want to set a title, use the setTitle method.

**Note:** You cannot set a description in a particular locale without having a title set in that same locale.

**Example - Modifying titles and descriptions:**

// obtain modifiable instance of a model node

final Modifiable modifiable = controller.getModifiableNode(node);

// check if the resource implements ModifiableLocalized interface

if (modifiable instanceof ModifiableLocalized) {

// set title and description

((ModifiableLocalized) modifiable).setTitle(Locale.GERMAN, "Titel");

((ModifiableLocalized) modifiable).setDescription(Locale.GERMAN, "Beschreibung");

}

* [**Setting unique names**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_set_unqnm.html)  
  You can set unique names for resources.

You can set unique names for resources.

To set the unique name for a resource, proceed by the following steps:

1. Obtain a modifiable instance of the resource for which you want to set the unique name.
2. Check whether the resource implements the ModifiableIdentifiable interface. To do this, use the operator instanceof. If the resource does not implement the ModifiableIdentifiable interface, you cannot modify it.
3. Obtain a modifiable instance of the resource object ID, that is ModifiableObjectID. To do this, use the getModifiableObjectID.
4. Set the unique name by using the setUniqueName() method of the ModifiableObjectID.

**Example - Setting unique names:**

// obtain modifiable instance of a model node

final Modifiable modifiable = controller.getModifiableNode(node);

// obtain modifiable instance of the resource's object id; note that modifiable instances

// of all model nodes implement the ModifiableIdentifiable interface

final ModifiableObjectID modifiableObjectID = ((ModifiableIdentifiable) modifiable).getModifiableObjectID();

// set unique name

modifiableObjectID.setUniqueName("MyUniqueName");

* [**Setting metadata**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_set_metadat.html)  
  You can set metadata on all modifiable instances that implement the ModifiableMetaDataProvider interface.

To set metadata for a resource, proceed by the following steps:

1. Obtain a modifiable instance of the resource for which you want to set metadata.
2. Check whether the resource implements the ModifiableMetaDataProvider interface. To do this, use the operator instanceof. If the resource does not implement the ModifiableMetaDataProvider interface, you cannot modify it.
3. Obtain a modifiable instance of the metadata of the resource.
4. Set the metadata by using the appropriate methods. For example, if you want to set metadata, use the setValue method.

**Example - Setting metadata for a resource:**

// obtain modifiable instance of a model node

final Modifiable modifiable = controller.getModifiableNode(node);

// check if the resource implements ModifiableMetaDataProvider interface

if (modifiable instanceof ModifiableMetaDataProvider) {

// obtain modifiable intance of the resource's meta data

final ModifiableMetaData modifiableMetaData = ((ModifiableMetaDataProvider) modifiable).getModifiableMetaData();

// set meta data

modifiableMetaData.setValue("MyKey", "MyValue");

}

* [**Setting supported markups**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_set_mrkp.html)  
  You can set supported markups on modifiable instances that implement the ModifiableMarkupCapable interface.

To set the supported markups for a resource, proceed as follows:

1. Obtain a modifiable instance of the resource for which you want to set supported markups.
2. Use the appropriate methods of the ModifiableMarkupCapable interface to set supported markups. For example, if you want to set a markup, use the addMarkup method.

**Example - Setting a supported markup on a layout control** (error handling omitted):

// obtain markup list

final MarkupList markupList = ...;

// obtain modifiable instance of a layout control

final Modifiable modifiable = lmController.getModifiableNode(control);

// obtain markup object

Markup markup = markupList.getByName("html");

// set markup

((ModifiableMarkupCapable) modifiable).addMarkup(markup);

* [**Setting the orientation for layout containers**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_set_orient.html)  
  You can set the orientation for modifiable instances of LayoutContainer nodes.

To set the orientation for a layout container, proceed by the following steps

1. Obtain a modifiable instance of a layout container.
2. Obtain the modifiable layout metrics instance.
3. Set the orientation by using the setValue() method.

**Example - Setting orientation of an existing layout container:**

// get modifiable instance of layout container

final ModifiableLayoutNode modifiable = (ModifiableLayoutNode) lmController.getModifiableNode(container);

// get modifiable layout metrics and set the orientation

ModifiableLayoutMetrics modifiableLayoutMetrics = modifiable.getModifiableLayoutMetrics();

modifiableLayoutMetrics.setValue(LayoutMetrics.ORIENTATION, Orientation.HORIZONTAL);

* [**Setting portlet preferences**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_set_ptlt_prf.html)  
  You can set the portlet preferences for portlet definitions and portlet entities.
* [**Setting flags**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_set_flag.html)  
  The Controller SPI allows you to set flags for resources. For example, you can set a flag for a page so that portal users can bookmark the page.

The Controller SPI allows you to set flags for resources. For example, you can set a flag for a page so that portal users can bookmark the page.

You set flags for resources either on modifiable instances of those resources or on controllers managing the resources. The following flags are available:

| *Table 1. Flags that you can modify* | | |
| --- | --- | --- |
| **Flag** | **Flag is associated to resource** | **Flag is set on** |
| ActiveFlag | ContentNode (resources that implement the ActiveFlag interface) | ModifiableContentNode |
| BookmarkableFlag | ContentNode (resources that implement the BookmarkableFlag interface) | ModifiableContentNode |
| ShareableFlag | ContentNode (resources that implement the ShareableFlag interface) | ModifiableContentNode |
| AllPortletsAllowedFlag | ContentPage | LayoutModelController |
| DeletableFlag | LayoutNode | LayoutModelController |
| ModifiableFlag | LayoutNode | LayoutModelController |

**Notes:**

1. ContentNode is the super class of ContentPage, ContentLabel, ContentURL, and InternalContentURL.
2. LayoutNode is the super class of LayoutContainer and LayoutControl.

In case of layered resources, for example derived content pages, you set flags only on the layer on which you work. To set flags, use the appropriate method. For example, to set the ActiveFlag, use the setActiveFlag method on the ModifiableContentNode instance. For more details about how the flags are aggregated in derivation scenarios refer to the Model SPI documentation.

**Example 1 - Setting the DeletableFlag for a layout container:**

// set modifiable flag on the layout model controller

lmController.setDeletableFlag(container, true);

**Example 2 - Setting the BookmarkableFlag for a content page:**

// obtain modifiable instance of an existing content page

final Modifiable modifiable = cmController.getModifiableNode(page);

// set modifiable flag on the modifiable instance

((ModifiableBookmarkableFlag) modifiable).setBookmarkableFlag(true);

* [**Setting themes**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_set_theme.html)  
  The Controller SPI allows you to set themes on modifiable instances that implement the ThemeSetter interface, for example ContentPage and ContentLabel.
* Obtain a modifiable instance of the resource for which you want to set a theme.
* Obtain a Theme object from the ThemeList model.
* Check whether the resource implements the ThemeSetter interface. To do this, use the operator instanceof. If the resource does not implement the ThemeSetter interface, you cannot set the theme.
* Use the setTheme() method to set the theme.

**Example - Setting a theme for a content page** (error handling omitted):

final Modifiable modifiable = cmController.getModifiableNode(page);

// obtain theme to set from com.ibm.portal.admin.ThemeList

final Theme theme = ...

// set theme

if (modifiable instanceof ThemeSetter) {

((ThemeSetter) modifiable).setTheme(theme);

}

* [**Setting URLs**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/ctrlrapit_set_url.html)  
  You can set URLs to point to external content (ExternalContentURL) or to internal content, that is to nodes in the portal (InternalContentURL).

**Setting external content URLs**

Learn how to set URLs that point to external content (ExternalContentURL).

To set a URL that point to external content (ContentURL), proceed by the following steps:

1. Obtain a modifiable instance of an external content URL.
2. Obtain the markup for which you want to set the URL.
3. Set the required URL for the modifiable instance of the external content URL.

**Example - Setting an external content URL:**

// obtain modifiable instance of an existing ContentURL

final Modifiable modifiable = cmController.getModifiableNode(contentUrl);

// identify target of url

final String target = ...

// identify markup

final Markup markup = ...

// set URL

((ModifiableContentURL) modifiable).setURL(target, markup);

**Setting internal content URLs**

Learn how to set URLs that point to portal internal content (InternalContentURL).

To set a URL that point to portal internal content (InternalContentURL), proceed by the following steps:

1. Obtain a modifiable instance of an internal content URL.
2. Identify the content node to which you want to assign the URL.
3. Obtain the markup for which you want to set the URL.
4. Set the required URL for the modifiable instance of the internal content URL.

**Example - Setting an internal content URL:**

// obtain modifiable instance of an existing InternalContentURL

final Modifiable modifiable = cmController.getModifiableNode(internalContentUrl);

// identify target of url

final Identifiable identifiable = ...

// identify markup

final Markup markup = ...

// set URL

((ModifiableInternalContentURL) modifiable).setTarget(identifiable, markup);

**Confirming modifications**

Modifiable interfaces and Controller interfaces provide methods for confirming modifications. You can use these confirm methods to check whether modifications might be prevented by portal internal constraints.

The confirm methods do **not** perform a modification, but they only indicate whether a modification can be performed or not. For example, if an administrator has locked the layout of a page, you cannot move portlets on that page by using the Controller SPI or by any other method.

For every method of the Controller SPI that modifies portal resources, the Controller SPI also provides a corresponding method to confirm the respective modification. All confirm methods start with the prefix confirm followed by the name of the method which is to be confirmed.

**Example - Confirming placement of a portlet by confirming the movement of the control of that portlet:**

// check if the control may be moved to the specified layout container

final boolean result = lmController.confirmInsert(control, container, null);

// check result

if (result == true) {

// control may be moved to specified layout container

} else {

// control may not be moved to specified layout container

}

**User and group management**

The Portal User Management Architecture (PUMA) System programming interface (SPI) provides interfaces for accessing the profiles of a portal *User* or *Group*.

PUMA SPI is used to find, create, modify and delete users and groups. Also, profile information about the currently logged in user can be retrieved. The *User* and *Group* interfaces returned by the SPI inherit the getObjectID() method from the *com.ibm.portal.Identifiable* interface of the Model SPI. This method is used to obtain the *ObjectID* that uniquely identifies a resource, in this case a user or group in the portal user registry. The following provider objects are used to access the *User* and *Group* objects.

**PumaProfile**

contains methods that provide read-only access the *User* and *Group* attributes and identifiers. You can use this interface to get the *User* object for the current user.

**PumaLocator**

contains methods for looking up *User* and *Group* objects. You can use this interface to obtain a *List* of *Group* objects for all of the groups in which the current user is a member.

**PumaController**

contains methods for creating and deleting Users and Groups and for modifying the *User* and *Group* profiles and membership.

Before the portlet can use these provider objects, it must first retrieve the appropriate home interface, depending on the type of application.

**Standard portlet**

*com.ibm.portal.um.portletservice.PumaHome*

**IBM portlet**

*com.ibm.portal.um.portletservice.legacy.PumaHome*

**Portal application (for example, theme or skin)**

*com.ibm.portal.um.PumaHome*

Examples of how these interfaces are retrieved are provided in the accompanying Javadoc. The following example shows how a standard portlet would obtain the identifier of a *User* as a *String*.

PortletServiceHome psh;

try{

javax.naming.Context ctx = new javax.naming.InitialContext();

psh = (PortletServiceHome)

ctx.lookup("portletservice/com.ibm.portal.um.portletservice.PumaHome");

if (psh != null){

PumaHome service = (PumaHome) psh.getPortletService(PumaHome.class);

PumaProfile pp = service.getProfile(request);

User user = pp.getCurrentUser();

String user\_objid = pp.getIdentifier(user);

}

}

catch (PumaException pe){

// ... error handling ...

} catch(javax.naming.NameNotFoundException ex) {

// ... error handling ...

} catch(javax.naming.NamingException ex) {

// ... error handling ...

}

**Note:** The PUMA SPI cannot be used with WSRP and remote portlets.

* [**Remote PUMA SPI REST Service**](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/uprof_rest.html)  
  The Remote PUMA SPI gives you access to user profiles through REST services. It provides a remote interface for user and group management for the configured Portal user repository. It is based on the REST (REpresentational State Transfer) architecture model.

**Remote PUMA SPI REST Service**

The Remote PUMA SPI gives you access to user profiles through REST services. It provides a remote interface for user and group management for the configured Portal user repository. It is based on the REST (REpresentational State Transfer) architecture model.

The Remote PUMA SPI REST service allows you to remotely perform the following tasks related to user and group data based on the HTTP protocol:

* Access, retrieve, and modify information about available user or group attributes and their meta data
* Search for users or groups based on attributes or group membership
* Create, update, or delete user or group profiles and group membership relations.

**Error codes**

* If the operations described above fail for some reason, the Remote PUMA SPI REST Service returns a subset of the HTTP error codes. They include a message with a detailed description of the error.
* The following list gives the error codes that the Remote PUMA SPI REST Service can return, and typical situations in which each specific error code is returned.
* **400 - Bad Request**
* The input provided in the request body does not comply to the expected format. For example, this can be invalid XML in a posted profile.
* **401 - Unauthorized**
* Access control check during request processing failed.
* **403 - Forbidden**
* The operation that the client tries to perform is not possible. For example, this can be creating a user ID that already exists, setting attribute values for attributes that are not defined, or using mutually exclusive request parameters.
* **404 - Not Found**
* The URI does not match any of the defined URI paths or a variable part of a defined URI path does not denote a resource that exists.
* **405 - Method Not Allowed**
* The request addresses a defined URI, but uses an HTTP method that is not defined for this URI.
* **415 - Unsupported Media Type**
* The format specified in the mime-type URI parameter or accept headers is not supported.
* **Data types for attributes**
* Data types for attributes belong to a subset of the XML Schema data type specification.
* You can use the following of data types with the PUMA REST API:
* **xs:string**
* String value or string representation of an arbitrary object.
* **xs:long**
* Number of type Long.
* **xs:integer**
* Number of type Integer.
* **xs:double**
* Number of type Double.
* **xs:dateTime**
* A timestamp with this format: yyyy-MM-dd'T'HH:mm:ss
* **xs:anyURI**
* URI path to a resource defined by the Remote PUMA SPI REST Service, for example to reference a user profile. This can be an absolute URL that includes the server name or a URI that denotes the absolute path.
* **xs:hexBinary**
* A hex encoded binary string.

**Web 2.0 user interface features**

Learn about portal features that pertain to the Web 2.0 generation type of Web user interface.

Web technology has evolved towards a different direction. In the public this evolution has been named Web 2.0. This term does not describe a new type of technology, but has been used in a broad manner to describe a change to a more user centered focus. Among the benefits are improved customer and service orientation, increased user activities, easier communication and collaboration, better usability, faster performance, etc.

WebSphere Portal Version 6.1 offers several **technical** features that can be related to this next generation type of Web user interface as they greatly enhance the portal user experience:

* The portal provides improved performance, as portal pages and portlets are rendered faster, and load is moved from the server to the clients.
* The user experience of the portal behavior is more similar to that of a desktop, as more logic is executed on the client rather than on the server.
* Due to new APIs, developing AJAX portlets becomes easier for portlet programmers.

Within portal, the Web 2.0 features are implemented as follows:

* **Web 2.0 portal theme with support for client side aggregation:** The portal can use client side aggregation (CSA) instead of server side aggregation (SSA). This has the following advantages:
  + Faster rendering and performance
  + Desktop like user experience
  + Existing portlets that were written to the server side programming model can be tied in by using AJAX (Asynchronous JavaScript and XML). For example, such portlets can be refreshed individually rather than requiring the refreshing of the whole page
  + The use of the portal Web 2.0 theme for portlets in your portal is optional. Server side aggregation is provided as a fallback option. For example, if the browser does not support JavaScript, the "old" portal rendering procedure is still available.
* **Client side portlet programming model:** You can use the client side programming model for your portlets. You can do everything with the client side programming model that you can do with the server side portlet programming model. Additionally, the client side programming model has the following advantages:
  + Improved user experience by faster responses and performance, as many interactions are processed on the client side rather than on the server.
  + User customization of user profile, preferences, and changes to the portlet state are done locally, and therefore with a faster response time. A fragment that contains the customization is later sent to the server and saved.
  + The user experience is consistent between both client side aggregation and server side aggregation. The user cannot tell the difference between CSA and SSA, except that CSA performs better.
* **Live text:** You can use live text. Live text provides elements embedded in portal pages that become active in the browser and are enhanced with additional functionality by JavaScript libraries. For example, if you include portal user IDs in your portlet output and mark them as live text, users can click on these IDs in the browser and see a person info card or a context menu that allows them to send a mail to the person. Live text has the following advantages:
  + Live text allows easier click to action.
  + You can adopt new portal content within your company more easily, as it is now easier to handle portal tags. For example, you can write tags and make them available centrally, and UI developers can reuse these tags for in their portlets for various purposes.
  + Content editors can add meaningful live text elements to portlets without requiring portlet development knowledge.
  + You can embed content from other sources, for example, from a HTTP or .NET server.
* **REST services:** 
  + Portal now offers the following REST services: Layout model, Portlet model, Content model, Navigation model, Wire model, User profile, composite applications.
  + By the use of REST services you can write your own advanced Web application and build it on top of REST (Representational State Transfer) services that provide the XML request information.
  + REST services allow you to access portal models remotely for both read and write access. The Navigation model allows read access only; it is updated by changes made to the content model.
* **Controller SPI:** The Controller SPI is a public portal interface. It is not directly related to the Web 2.0 type of user experience, but it allows you to perform certain administrative tasks more easily.

**Terminology**

These are terms that are used in the documentation for the Web 2.0 features:

**CSA**

**Client Side Aggregation.** This means aggregation based on JavaScript and XSLT transformations that are executed on the client. This is the aggregation model that provides an improved user experience by faster response and performance.

**SSA**

**Server Side Aggregation.** This means aggregation based on JSPs that are executed on the Server. This is the "old" portal aggregation model; this still works as before.

**Pure Server Side Portlet**

This is a normal server side portlet that uses Java and JSPs; it usually uses no JavaScript. Portlets that are written to the client side model use no or few JSPs.

**AJAX Portlet**

This is a normal server side portlet that uses lots of JavaScript and AJAX technologies and less Java & JSPs.

**DPR**

**Differential Page Rendering.** This denotes the server side rendering model that is used by the Web 2.0 theme. The concept of DPR is to render only those parts of a portal page that were affected by the a user interaction. For example, if a user interacts with a single self-contained portlet that runs in the Web 2.0 theme, the portal refreshes only this portlet rather than the entire portal page.

**REST:-**Representational State Transfer

**TOPIC 1:-**

**The client side portlet programming model**

You can use the client side programming model to make use of AJAX techniques in your standard and IBM portlet API portlets.

You can use the client side programming model for writing your portlets. You can do everything with the client side programming model that you can do with the server side portlet programming model. Additionally, the client side programming model has the following advantages:

* The client side portlet programming model works in both client side aggregation (CSA) and server side aggregation (SSA).
* Improved end user experience by faster response and performance. Portlets that use the clients side programming model render faster, as the portal does not re-render the whole page, but only the aspects of the portlet that change.
* The client side programming model allows you to handle changes of the portlet mode and window state, preferences, and user customization of user profile locally. This provides a faster response time for the user. A fragment that contains the customization is later sent to the server and saved.
* Writing portlets to the client side programming model does not require deep Java knowledge. You can write such portlets as HTML code with css style sheets and Javascript. They have few or no JSPs. Example scenarios for writing client side portlets are:
  + A user can add some markup to the portlet view by selecting options in a form.
  + Preference changes that a user makes to the portlet are applied immediately in the browser view without reloading the whole portlet. The preference change is later sent to the server and saved.
  + Access to the user profile on the client side.
  + Portlet mode changes are performed entirely on the client. Performance improves as no server-client round trips are required.
  + The portlet can retrieve markup fragments from the server. The user does not notice this, as the ATOM feed format is hidden by the XMLPortletRequest implementation, similar to a XMLHttpRequest (XHR).
  + XSLT and XPath helper functions make it easier to handle XML feeds.

**Comparing the standard portlet API to the IBM portlet API**

Find out about the similarities and differences between the standard portlet API and the IBM portlet API.

Between the Java Portlet Specification and the IBM portlet API, the fundamental concepts of portal servers are the same. The *portlet container* is the run time environment for portlets using the portlet API. The portal server receives the client request for a page, calls the portlets for the page and calls the container to provide the output for each portlet. The portal server aggregates the output from all portlets on the page to provide a complete portal page in the response to the client. The following topics describe the differences between the APIs used by portlets running in the portlet container.

* [Relationship to servlet API](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/jsrcpr.html#jsrcpr__servletapi)
* [Comparing common portlet API concepts](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/jsrcpr.html#jsrcpr__commconc)
* [Comparing elements of the API](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/jsrcpr.html#jsrcpr__elements)
* [Concepts unique to the Java Portlet Specification](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/jsrcpr.html#jsrcpr__uniqjsr)
* [Concepts unique to the IBM portlet API](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/jsrcpr.html#jsrcpr__uniqibm)

**Relationship to servlet API**

The IBM portlet API extends the servlet API and many of the main interfaces (request, response, session). The standard portlet API does not extend the servlet API, but shares many of the same characteristics (see *Relationship with the Servlet Specification* in the *Java Portlet Specification*). The standard portlet API takes advantage of much of the functionality provided by the servlet specification, such as deployment, class loading, Web applications, Web application lifecycle management, session management and request dispatching.

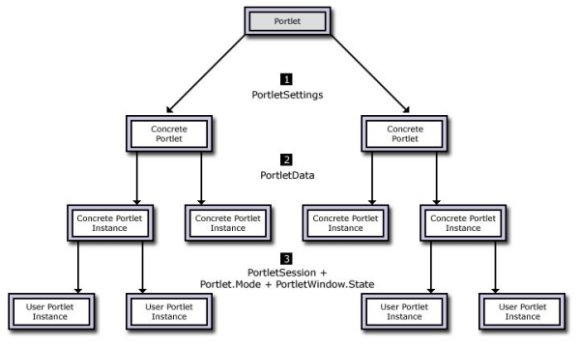
Standard portlets can call servlets and package them in the same portlet application. Portlets, servlets and JSPs within the same portlet application share the same classloader, application context and session.

**Comparing common portlet API concepts**

For both portlet APIs, there is one portlet object instance per portlet configuration in the Web deployment descriptor. The APIs differ in how each portlet object is represented in the portal.

**Logical representation of portlets in the IBM portlet API**

The PortletSettings object ( 1 ) provides the portlet with its unique set of configuration parameters and values. Configuration parameters are initially defined in the portlet deployment descriptor, but can also be added or modified by the portal administrator. There can be many PortletSettings objects, parameterizing the same portlet object according to the Flyweight pattern, provided on a per-request basis. Each PortletSettings object along with the portlet object comprises a concrete portlet.



When users edit the settings in a portlet according to their preferences, the settings are saved to persistence using the PortletData object ( 2 ). There can be many PortletData objects parameterizing the same concrete portlet. Each PortletData object along with the concrete portlet comprises a concrete portlet instance.

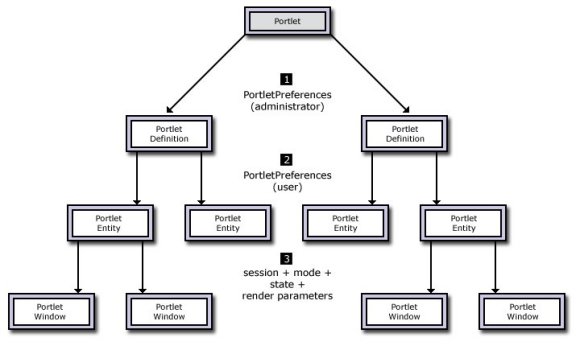
When multiple users interact in different browser sessions with the same concrete portlet instance on a portal page, each user sees a particular user portlet instance, which is a concrete portlet instance parameterized with the session, a mode and window state ( 3 ).

**Logical representation of portlets in the Java Portlet Specification**

In WebSphere Portal, standard portlets, like IBM portlets, can also be configured on two levels, by an administrator and by individual users. However, the standard portlet API has no concept of a concrete portlet application or concrete portlets. The number of configuration layers is opaque to the programming model; the portlet configuration is contained within a single PortletPreferences object which aggregates these configuration layers.

Preferences can be marked in the portlet.xml as read-only so that they may be customized only by administrators ( 1 ) using configure mode or through the administration portlets. Preferences that are not marked read-only can be modified by users in edit mode ( 2 ).

The parameterized view of a portlet for an individual user interaction is called portlet window in the Java Portlet Specification. The standard portlet API uses render parameters as an alternative way to hold interaction state for a portlet window. A portal page can contain more than one portlet window for a single portlet, each portlet window associated with unique portlet mode, state, and render parameters ( 3 ).



**Comparing elements of the API**

**Utility classes**

The Portlet interface is the main abstraction for both the Java Portlet Specification and the IBM Portlet API. Portlets for both APIs should extend the corresponding utility classes, which provide default functionality for the portlet.

* Standard portlets should extend the GenericPortlet class.
* IBM portlets should extend the PortletAdapter class.

**Life cycle**

The following table compares the portlet life cycle between IBM portlets and standard portlets.

| **IBM Portlet API** | **Java Portlet Specification** | **Description** |
| --- | --- | --- |
| init() | init() | Called after the portlet has been loaded and instantiated. Portlets use this method to perform one-time operations, especially those that might incur performance costs. The portlet can access its configuration, including initialization parameters, within this method. |
| initConcrete() | none | The Java Portlet Specification does not have a concept of a concrete portlet. |
| service() | render() | Used to render output to the client. These methods are typically applied using the mode-specific implementations of doView(), doEdit(), and doHelp(). Using the standard portlet API, this method is invoked when the portal receives a render request. |
| none | processAction() | Used by the portlet to received parameters, update state, and perform any other necessary action processing. This is not part of the life cycle of the IBM portlet because the IBM portlet must implement an ActionListener for its counterpart, actionPerformed(), to be called. This method is invoked when the portal receives an action request. |
| destroyConcrete() | none | The Java Portlet Specification does not have a concept of a concrete portlet. |
| destroy() | destroy() | Called when the portlet is to be removed from service(). |

**Request handling**

Similar to servlets, portlets interact with clients using a request/response paradigm implemented by the portlet container. In a typical scenario, a user selects a page and the portlet container sends the request to each portlet on that page to render output in its response. These requests are called *render requests*. Other requests, however, are launched when the user interacts with one of the portlets on the page, such as clicking a submit button. Such actions generate URLs to the portlet, which might require the portlet to process an action before sending the response. These are called *action requests*.

In the servlet model, all request handling would be performed in the service() method. Portlets, however, use a two phase processing that is split between an event phase and a render phase. If the portlet receives an action request, then the event phase is started in which the portlet's action processing method is called. The result of this action could invoke actions for other portlets on the page. After actions have been complete for all portlets on the page, each portlet's rendering method is called. The event phase is guaranteed to complete before the start of the render phase.

The following table describes the differences between the IBM Portlet API and the Java Portlet Specification for handling requests.

| **IBM Portlet API** | **Java Portlet Specification** |
| --- | --- |
| The portlet must implement an ActionListener to participate in the event phase. The actionPerformed() method is invoked to handle action requests. Request and session attributes can be set in the action and retrieved from the subsequent render phase.  During the render phase, the service() method is called. Methods for both phases take the same PortletRequest and PortletResponse objects as arguments. There is no distinction in the API between action request and response objects and render request and response objects. | The processAction() method is invoked to handle action requests. The main difference is that the action request and response objects are different from render request and response objects. The portlet can set render parameters or session attributes during the action that are available in the render call, but request attributes are not transferred between action and render phase. |

For both APIs, if the portlet extends the appropriate utility class, the service() method dispatches the doView(), doEdit() or doHelp() methods, depending on the current portlet mode.

**Portlet mode**

Modes allow portlets to provide different interfaces depending on the task that is required of them. The following modes are supported by both APIs.

**View**

View mode is used for displaying portlet content. Utility class invokes doView() method.

**Edit**

Edit mode is used for personalizing the portlet (portlet data). Utility class invokes doEdit() method.

**Help**

Help mode is used for displaying help on the portlet Utility class invokes doHelp() method.

The following table describes support for other modes in each API.

| **IBM Portlet API** | **Java Portlet Specification** |
| --- | --- |
| Configure mode is used by an administrator for globally configuring the portlet (portlet settings). Changes affect all occurrences of the portlet on all pages. Utility class invokes doConfigure() method. | Config mode supported as a custom mode. In config mode, the administrator can globally update the configuration of a portlet, including portlet's read-only preferences. Changes affect all occurrences of the portlet on all pages. In config mode, the portlet's read-only preferences can be updated by the administrator. |
| No custom modes. | Allows portlets and portal server implementations to support custom modes. At run time, portlets use PortalContext.getSupportedPortletModes() to retrieve the modes supported by the portal and adapt accordingly. The Java Portlet Specification also suggests these custom portlet modes: about, config, edit\_defaults, preview, and print. Vendors can also define other portlet modes.  The current portlet container in WebSphere Portal supports config and edit\_defaults. |

**Portlet window states**

Window states indicate the amount of space that the portlet consumes on a page. The portlet can query its state to determine the visible size of the content it should render. The following states are supported by both APIs.

**Normal**

The portlet is provided equivalent space as other portlets on the page or in the same container.

**Maximized**

No other portlets are displayed, providing maximum space for the portlet's use.

**Minimized**

Only the portlet's title bar is displayed. The Java Portlet Specification allows the portlet to provided a limited amount of output, but for WebSphere Portal, the portlet's output is not displayed at all.

The following table describes support for other states in each API.

| **IBM Portlet API** | **Java Portlet Specification** |
| --- | --- |
| Solo | Supported only as a custom window state. WebSphere Portal does not support solo state for standard portlets. |
| No custom states. | Allows portlets and portal implementations to define custom window states. At deployment time, the portal can map custom portlet window states from the portlet descriptor to its own custom window states, or it can ignore them. At run time portlets use PortalContext.getSupportedWindowStates() to retrieve the modes supported by the portal and adapt accordingly. |

**Portlet URLs**

Portlet URLs allow the portlet to create a URL to itself. When the user clicks a link or performs an action that launches the URL, it creates a new request to the portal targeted to the portlet. The following table describes how each API implements this functionality.

| **IBM Portlet API** | **Java Portlet Specification** |
| --- | --- |
| PortletResponse.createURI() creates a URI to the portlet. An action can be set on the PortletURI object, so that the actionPerformed() method of the portlet is called before the service() method. | RenderResponse.createRenderURL() creates an URL that triggers the render() method of the portlet.  RenderResponse.createActionURL() creates an URL that triggers the processAction() method of the portlet before the render() method.  HTML form processing must be done using an action URL. Render and action parameters are separate for standard portlets; parameters set on an action URL are not available to the render() method unless they are passed explicitly using ActionResponse.setRenderParameter().  See [Request handling](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/jsrcpr.html#jsrcpr__request_handling) for details about the action and render method. |

**Namespace encoding**

Portlets must contribute identifiers and names in their portlet output that is unique within the portal page. For example, two portlets could appear on a page, both with an anchor named return. Namespace encoding allows the portlet to ensure these names do not clash by adding the portlet's namespace to the anchor name. The following table shows how namespace encoding is performed in the Java source for each API.

| **IBM Portlet API** | **Java Portlet Specification** |
| --- | --- |
| PortletResponse.encodeNamespace(*name*) | RenderResponse.getNamespace(*name*) returns a namespace that is valid for the current user session. |

See [JSPs](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/jsrcpr.html#jsrcpr__jsps) for a comparison of how namespace encoding is performed for JSPs.

**JSPs**

The following table compares how JSPs are invoked from the portlet class for each API.

| **IBM Portlet API** | **Java Portlet Specification** |
| --- | --- |
| PortletContext.include() | PortletRequestDispatcher.include()  The request dispatcher serves as a wrapper for a resource at the path indicated by getRequestDispatcher(*path*). |

The following table describes the differences in the tag libraries for each API.

| **IBM Portlet API** | **Java Portlet Specification** |
| --- | --- |
| <portletAPI:init/> - provides access to the PortletRequest, PortletResponse, and PortletConfig objects. | <portlet:defineObjects/> - provides access to the RenderRequest, RenderResponse, and PortletConfig objects |
| <portletAPI:createURI/>  Creates a URI that points to the portlet. To invoke an action in an IBM portlet, include the <portletAPI:URIAction/> tag and action parameters. | <portlet:renderURL/> creates an URL that will trigger the render() method of the portlet. <portlet:actionURL/> creates an URL that will trigger the processAction() method of the portlet before the render() method. See [Portlet URLs](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/jsrcpr.html#jsrcpr__portlet_urls) for more information on render and action URLs. |
| <portletAPI:encodeNamespace/> | <portlet:namespace/> returns a namespace that is valid for the current user session. |
| Other tags provided by the IBM portlet API tag library | Use JSTL tags |

**URLs to portlet resources**

In WebSphere Portal, resources within a portlet cannot be invoked using relative URLs. For URLs to these resources to work, they must be encoded using the encodeURL() method of the portlet response. Resources include any file in the portlet application that is referenced or invoked from the portlet or one of its JSPs, including:

* images
* applets
* multimedia
* JSPs
* servlets

The following table explains the differences between each API for using the encodeURL() method.

| **IBM Portlet API** | **Java Portlet Specification** |
| --- | --- |
| Use the encodeURL() method of the response and specify the path to the resource. Example:  <%=  portletResponse.encodeURL(renderRequest.getContextPath()  + "/images/photo01.jpg")  %> | Use the encodeURL() method, just as you would with the IBM portlet, except that you also have to add the context path of the portlet from the request. Example:  <%= renderResponse.encodeURL("/images/photo01.jpg") %> |

**Caching**

Both APIs provide the means for portlets to cache their output and improve performance and response times. The table below describes the differences in the implementations.

| **IBM Portlet API** | **Java Portlet Specification** |
| --- | --- |
| Expiration timer set in the portlet.xml. The IBM Portlet API supports [invalidation-based caching](http://publib.boulder.ibm.com/infocenter/wpdoc/v6r1/topic/com.ibm.wp.ent.doc_v6101/dev/jsrcpr.html#jsrcpr__invalidation). Can be dynamically changed by the portlet when its getLastModified() method is called. | Expiration timer set in the portlet.xml. Can be dynamically changed by the portlet using the EXPIRATION\_CACHE property of the RenderResponse. |

**Portlet configuration**

For both APIs, the PortletConfig object is made available to portlet during initialization.

| **IBM Portlet API** | **Java Portlet Specification** |
| --- | --- |
| Provides the initialization parameters defined in the web.xml. Initialization parameters can be read using the getInitParameter() method of the Portlet interface. | Provides the initialization parameters, which can be defined in the portlet.xml or the web.xml. Initialization parameters can be read using the getInitParameter() method of the PortletConfig interface. |

**Concepts unique to the Java Portlet Specification**

**Render parameters**

These are parameters attached to the render request that stay the same for every render request until a new action occurs. This allows storing navigational state in the render parameters instead of the session. The portlet should put all state information it needs to redisplay itself correctly into render parameters (for example, which screen to render). This facilitates saving bookmarks and provides better support for the use of the browser's back button.

When the portlet is the target of an action, render parameters are reset. In the action, the portlet can set new render parameters to represent the new navigational state after the action is completed. The developer can use render URLs (see RenderResponse.createRenderURL()) to set render parameters directly without having to go through an action. As render parameters may be bookmarked, portlets should be prepared to handle parameters that are not valid (for example, that could be the result of an outdated bookmark). Such parameter values should be handled gracefully and should not lead to exceptions.

From the JSP, the portlet can set render parameters using the <portlet:renderURL/> tag.

**Portal context**

To allow portlets to adapt to the portal that is calling them, the Java Portlet Specification provides the PortalContext to be retrieved from the request. This portal context provides information such as the portal vendor, version, and specific portal properties. This allows the portlet to use specific vendor extensions when called by the portal of that vendor and fall back to some simpler default behavior when called by other portals.

As the portal context is attached to the request, it may change from request to request. This can be the case in the remote scenario, where one portlet (WSRP provider) may be called from different portals (WSRP consumers).

**Access to the portal user profile**

In the Java Portlet Specification, a portlet can define in the deployment descriptor which user profile information it wants to access. The specification proposes to use a list of standard P3P attributes, however the portlet is free to request any additional user information that is not covered by this attribute list. At deployment time the portal can map the requested user profile attributes to the supported profile attributes or the portal can ignore the requested attributes. At run time the portlet can use the USER\_INFO constant of the PortletRequest to determine which of the requested user profile attributes are available. For a list of attributes refer to the JSR portlet specification.

**Request and response properties**

The portlet and portlet container can exchange vendor-specific information using request and response properties. These properties are available in the action request and response and the render request and response. Properties are propagated for includes, but not between the action and render phase. When including a servlet or JSP, the properties are mapped to headers in the servlet API. WebSphere Portal does not support any vendor-specific request or response properties.

**Web application session scope**

The standard portlet API supports the Web application session scope, in which every component of the Web application can share information through the APPLICATION\_SCOPE field of the portlet session. This can be used to share transient information between different components of the same Web application, for example, between portlets or between a portlet and a servlet.

**Reuse of the HttpSession listeners**

As the standard portlet API reuses the HttpSession, it also allows reusing all the session and attribute listeners that the servlet 2.3 specification defines. The Java Portlet Specification provides a PortletSessionUtil class for decoding the attributes of the HttpSession, as these are namespaced in the private portlet session case.

**Resource bundles**

Portlets can define a resource bundle in the deployment descriptor that can be accessed at run time using PortletConfig.getResourceBundle(). This provides globalizations of resources, such as the portlet title, search keywords, or preference names and descriptions. At deployment time, all settings in the portlet descriptor that are intended to be viewed or changed by the administrator (for example, portlet description, init parameters, or display name) can be globalized by setting the xml:lang attribute on the associated tag (like the servlet 2.4 deployment descriptor). The Java Portlet Specification also recommends a notation for globalizing the preference attribute display names, values, and descriptions.

**Multiple response content types**

For each response, portlets can retrieve a list of supported content types using the PortletRequest.getResponseContentTypes() method. This allows portals to indicate to portlets that they have transcoding capabilities.

Portlets receive only content types that they have defined in the <supports> section of the deployment descriptor. If the portlet declares support for content types using wildcards (for example, "text/\*" or "\*/\*") in the deployment descriptor, the portlet container may also set these content types as response content type. Therefore portlets specifying wildcard content types in the deployment should handle wildcard content types as response content types accordingly.

**Redirect in action**

During the action phase, the standard portlet API enables portlets to redirect requests to other Web resources. This allows portlets to process the request from different resources (for example, an accounting servlet) in response to an action.

**Preference validator**

Portlets can provide a class that validates the set of preference values in the PortletPreferences object. The class is defined in the portlet.xml. The preference validator can incorporate logic to check cross-dependencies between different preference properties. The portlet container always calls the validator before storing a preference set to ensure that only consistent preference sets are stored.

**Concepts unique to the IBM portlet API**

* Portlet events

The IBM Portlet API has the concepts of events. This event concept is based on the JetSpeed event model, which is similar to the Java event model. The IBM Portlet API provides the following events:

* + ActionEvent
  + MessageEvent
  + PortletApplicationSettingsAttributeEvent
  + PortletSettingsAttributeEvent
  + WindowEvent
* Additional listeners

The listener concept of the IBM Portlet API allows the portlet to get notifications not only for events as described in the section above, but also for events related to the session lifecycle, event phase lifecycle or render phase lifecycle. The IBM Portlet API provides the following listeners to implement this functionality:

* + PortletPageListener
  + PortletSessionListener
  + EventPhaseListener
* Portlet menus

The portlet menu service allows the portlet to contribute content to the portal navigation menu, providing users easier navigation through portal pages.

* Invalidation-based caching

Portlet can actively invalidate the cache using the invalidate method on the portlet request as result of an action. This allows for more fine-grained cache control as the portlet can determine, as a result of this action, whether to invalidate the cache content for only the current portlet mode and markup or for all of the portlet's modes and markups. In the Java Portlet Specification, an action invalidates all cached markup.