

Best practices: Developing portlets using JSR 168 and WebSphere Portal V5.02

[Stefan Hepper](#)

Architect, IBM Websphere Portal Development

[Marshall Lamb](#)

Chief Programmer for WebSphere Portal V5

March 3, 2004

© Copyright International Business Machines Corporation 2004. All rights reserved.



Table of contents

Overview	3
Definitions	3
Portlet entity	3
Portlet window	4
Portlet application definition	4
JSR 168 specific concepts	4
Persistent state	4
Transient state	5
Portlet and servlet relationship	5
Supported extensions in WebSphere Portal V5.0.2	6
Portlet design guidelines	8
Portlet application design principles	8
Portlet design principles	8
Portlet development guidelines	11
Portlet coding guidelines	11
JSP coding guidelines	14
Portlet packaging guidelines	17
Data management	17
Session management	18
Internationalization	20
Inter-portlet communication	21
Multiple markup support	21
Performance considerations	22
Security considerations	24
Remote portlet considerations	25
Portlet documentation	25
Summary	25
Resources	26
Standards and specifications	26
WebSphere Portal	26
Struts support for portal	26
Related topics	27
Accessibility standards	27
About the authors	28

Overview

The first release of Java™ Portlet Specification, JSR 168, by the Java Community Process (JCP), provides a standard for interoperability between portlets and portals. Portlets written to this portlet API can be deployed on any JSR 168 compliant portal and run out-of-the-box.

The portlet architecture is an extension to the Java Servlet architecture. Therefore, many aspects of portlet development are common to typical Web application development. However, the unique aspects of a portal add complexity to the application model, such as:

- Multiple portlets per Web page
- Portlet URL addressing
- Portlet page flow control
- User interface rendering restrictions
- Inter-portlet communication

You need to carefully consider the design of a portlet to take advantage of portal features instead of falling prey to its complexity. By following the best practices outlined in this document your portlets will be portable, maintainable, and will perform optimally.

This document is a collection of best practices for portlet developers who want their portlets to conform to, and to leverage the IBM® WebSphere® Portal infrastructure for, JSR 168. Divided into major categories, you can use these coding guidelines when designing and developing JSR 168 portlets for WebSphere Portal. It is not a primer for portlet development, because it does not address the fundamentals of portlet programming. Instead, use it as a checklist during design and code reviews to help promote consistent and quality portlet implementations.

Refer to the [WebSphere Portal V5.02 InfoCenter](#) and the [Portlet Development Guide](#) for more information on developing portlets. See also the [Resources](#) listed at the end of this article, including references for JSR 168.

Definitions

This section covers the basic definitions you need to understand in order to program JSR 168 portlets.

Portlet entity

A portlet entity is a Java instance of a portlet parameterized with additional settings, a pattern known as the *flyweight* pattern. See *Design Pattern* in [Resources](#). The

parameterization is done with the portlet preferences, which form persistent state information that can be customized.

The initial settings can be defined in the portlet deployment descriptor. A portal administrator can modify these settings for a base portlet entity, such as a server address from which a portlet retrieves stock quotes. From this base portlet entity, new entities can be created for each user who places this portlet on his or her page. The user can then further customize the portlet preferences, such as selecting favorite stock quotes. However, further details of how to build and deploy portlet entity hierarchies is not specified in the first version of the portlet specification.

Portlet window

The portlet window is the window in which the markup fragment, produced by a portlet produces, displays. The portlet window has the portlet mode, window state, and render parameters attached to it. Decoupling of the portlet window from the portlet entity allows different windows to point to the same portlet entity, and to be in different window states or portlet modes.

Portlet application definition

The portlet application definition describes some or all of the portlets defined in the portlet application. By default the portlet application definition contains all portlets available in the portlet application. Using portlet application cloning the administrator can create new portlet application definitions, and can add to or remove portlets from the portlet application definition. All portlet application definitions run in the same Web application, and therefore, also share the same session.

The portlet application definition concept is IBM WebSphere Portal specific; therefore, it does not show up in the JSR 168 standard portlet deployment descriptor. You can use XMLAccess to import or export portlet application definitions.

JSR 168 specific concepts

This section covers concepts that are significantly different in the JSR 168 compared to the IBM Portlet API that was introduced in WebSphere Portal V4.1.

Persistent state

A portlet can access two different types of persistent data: initialization parameters and portlet preferences.

Initialization parameters are read-only data which you specify in the portlet deployment descriptor to define settings that are the same for all portlet entities created from this portlet description. You can use them to specify basic portlet parameters, such as the names of the JSPs that render the output.

A portlet can also access persistent data using *portlet preferences*, of which there are two different categories:

1. **User independent preferences** are declared in the portlet deployment descriptor as read-only and cannot be changed by the user. Only an administrator can change these settings using the portlet config mode or an external tool. Examples of user independent preferences include server settings and billing information.
2. **User dependent preferences** can be changed by the user to customize the portlet, normally in edit portlet mode. Examples of user dependent preferences include stock quotes or news topics of interest to the user.

Preferences can be read and written in the action phase, and read in the render phase. The preferences can be either strings or string array values associated with a key of type *string*. You can set default values for preferences in the deployment descriptor.

Transient state

A portlet has access to two different kinds of transient state: session state and navigational state.

The *session* state is available to the portlet for each authenticated user. The session concept is based on the *HttpSession* defined for Web applications. Because portlet applications are Web applications, they use the same session as servlets.

To allow portlets to store temporary data private to a portlet entity, the default session scope is the portlet scope. In portlet scope, the portlet can store information, needed across user requests, that are specific to a portlet entity. Attributes stored with portlet scope are prefixed in the session by the portlet container to avoid two portlets (or two entities of the same portlet definition) overwriting each other's settings.

The second scope is the Web application session scope, in which every component of the Web application can access the information. The information can be used to share transient state among different components of the same Web application (such as between portlets, or between a portlet and a servlet).

Navigational state defines how the current view of the portlet is rendered and is specific to the portlet window in which the portlet is rendered. Navigational state can consist of data such as the portlet mode, window state, a sub-screen id, and other data. The navigational state is represented in the JSR 168 portlet API through the portlet mode, window state, and the render parameters. The portlet receives the render parameter for each render call. The data can only be changed in an action, or by clicking on a render link with new render parameters.

Portlet and servlet relationship

In the IBM Portlet API, portlets extend servlets and all the major interfaces (such as Request, Response, and Session) extend the corresponding servlet interfaces. In JSR 168,

portlets are separate components that may be wrapped as servlets, but they do not need to be servlets.

This separation was made to enable the different behavior and capabilities of portlets. Because a portlet is not a servlet, in JSR 168 it is possible to define a clear programming interface and behavior for portlets.

However, in order to reuse as much as possible of the existing servlet infrastructure, JSR 168 leverages functionality provided by the Servlet Specification wherever possible, including:

- Deployment
- Classloading
- Web applications
- Web application lifecycle management
- Session management
- Request dispatching

Many concepts and parts of the portlet API have been modeled similar to the servlet API.

Supported extensions in WebSphere Portal V5.0.2

These optional parts of JSR 168 are supported in WebSphere Portal V5.0.2:

- **Custom mode config**, which enables administrators to configure portlets for all users of a portlet.
- **P3P user profile attributes**, which allows a portlet access to user profile attributes. The list of supported profile attributes is provided in the Portlet Specification, *Appendix D. The Platform for Privacy Preferences*.
- **Caching**. JSR 168 expiration- based caching is supported.

The following non-standard extensions are supported in WebSphere Portal V5.0.2, in order to close gaps of the first version of the JSR 168 portlet API. Use these extensions with discretion, and program the portlet so that it has a fallback mechanism if these extensions are not present.

- **Markup property** – WebSphere Portal sends a render request property name, `wps.markup`, to enable the portlet access to the markup type, which is a more fine-grained definition of the markup to return than the MIME type. For example, HTML and cHTML both have MIME type `text/html`, but have markup `html` and `chtml` to enable the portlet to produce different output for the HTML and cHTML cases.
- **User agent property** – WebSphere Portal sends the client user agent as a render request parameter with the name `wps.user-agent` to enable the portlet to tailor its output to specific devices.

Best practices for developing portlets using JSR 168 and WebSphere Portal V5.02

- **Parallel portlet rendering** – WebSphere Portal allows rendering portlets on a page in parallel to improve the response time for portlets with backend connections. This is completely transparent for the portlet.

WebSphere Portal V5.0.2 sets the following Strings in the `PortalContext` to allow a portlet to identify IBM WebSphere Portal 5.0.2 as the calling portal:

- Portal info
- Supported portlet modes: EDIT, HELP, VIEW, CONFIG
- Supported window states: MINIMIZED, NORMAL, MAXIMIZED

The following optional parts of the Java Portlet Specification are not implemented by the WebSphere Portal V5.0.2 JSR 168 implementation:

- No support of J2EE roles
- No mapping of additional user attributes beyond the P3P attributes

Portlet design guidelines

This section describes principles to follow as you design portlets and their applications. It provides guidelines to structure and design your portlets and applications in a modular and extensible manner.

Portlet application design principles

Portlets and pages are the basic building blocks of portal applications. They provide the sub-units of user experience that are aggregated by the portal application to provide the final user experience.

Portlets are packaged into portlet applications, which are similar to J2EE Web applications, except they make use of additional interfaces that make it easier to aggregate portlets onto a single page. The packaging format of portlet applications is the Web ARchive (WAR) format which, in addition to the `web.xml`, includes the `portlet.xml` deployment descriptor, which defines the portlet relevant parts of the Web application.

A single portlet produces the content of one portlet window. In order to leverage the flexibility that portlets can be used as building blocks, portlet application developers should strive for small portlets that contain a specific function instead of one large portlet that includes all functions. Using small, specific portlets has several advantages:

- The portal end-user can decide to only put the needed parts on the page and save space that otherwise would be occupied by parts that are not needed
- The different functions can be split across portal pages to suit the working behavior of the user, and can adapt to limited device display capabilities
- Additional functions can be added later as new portlets, without touching the existing running portlet.

Portlets that belong to the same logical application should be bundled together in a portlet application, providing several advantages over the approach of one portlet per portlet application. The advantages include the ability to share configuration data and session data, and the ease of deployment and administration .

Portlet design principles

The correct portlet design is broken down into three distinct parts: the model, the view, and the controller (MVC). This design follows classical object oriented design patterns where each part is self-contained and modular, easing maintenance, extensions, and advancements.

The *model* is the data to which the portlet provides an interface. Common data models are XML documents, database tables, and even other Web applications. The Java classes accessing the data model should have no knowledge of the form that the data is in, the

idea being that the model can be changed without affecting the rest of the portlet application.

The *view* is the interface to the data model, presenting it in some usable format. The view accesses the data to be rendered through the model interfaces, and therefore, does not care what format the model takes. It should also not understand the relationships between data models or represent any of the business logic for manipulating the data. Like the data model, the view should be independent and interchangeable, allowing other views to be substituted without affecting the business logic or the model. The typical embodiment of the view is through series of Java Server Pages (JSPs), but it can also render using other techniques such as using XSL stylesheets to format XML documents.

The *controller* is the glue that ties the model to the view and defines the interaction pattern in the portlet. The controller handles user requests from the view and passes control to the appropriate model objects to complete the requested action. The results of the action are then rendered back to the user by the controller using appropriate view objects and, perhaps, model objects which represent the data results of the completed action.

Best practices for developing portlets using JSR 168 and WebSphere Portal V5.02

The controller resides in the portlet Java classes. It knows the data model only through the model interfaces and it knows the view only in that it dispatches the view to render the data. Therefore, the controller logic can be just as easily replaced as the view and the model. Typical controller implementations utilize intrinsic functions in the JSR 168 portlet API for coordinating action sequences around user input, model data calculations, and result rendering.

As you design your portlets, it is extremely important to hold true to the MVC design principles. Portlets typically evolve over time and are largely reused as basis for new portlets. The ability to adapt a portlet to a new backend data provider, or add markup support for mobile devices, or enhance the portlet to include user personalization, requires that each part of the portlet be self-contained and extensible.

See [Resources](#) for links to other resources about MVC design principle.

Portlet development guidelines

The following portlet guidelines are organized by functional categories. As with any coding practice, there are exceptions to every rule. These guidelines are intended to help you produce best-of-breed JSR 168 portlets for the WebSphere Portal environment. The guidelines must ultimately be adapted to your development environment and product architecture

Portlet coding guidelines

1. **Do not use instance variables.** Portlets, like servlets, exist as a singleton instance within the server's JVM. Therefore, a single memory image of the portlet services all requests, and it must be thread-safe. Data stored in instance variables will be accessible across requests and across users and can collide with other requests. Data must be passed to internal methods as parameters. There are other means of storing data globally. Refer to the Data management section for more details.
2. **Pass data to the view (JSP) as a bean in the request object.** Use the `RenderRequest` object to pass data to the view for rendering, so that when the request is complete, the data falls out of scope and is cleaned up. Passing it as a bean lets the JSP simply refer to the data as properties on the bean using intrinsic functions in the JSP syntax.
3. **Adopt good code documentation habits.** While commenting of code is not required for the functionality of the portlet, it is essential for its maintenance. A portlet's maintenance can change hands over time, and well-written portlets serve as models for other portlets. Therefore, someone else must understand what you wrote. Well documented code implies more than simply inserting comments; it also implies good naming practices for Java resources. The following are examples of guidelines to follow:
 - a. Insert JavaDoc-compliant prologues for all public classes, variables, and methods. Document inputs and outputs.
 - b. Include inline comments, but do not include them on the same line as Java source. It is difficult to align and follow comments which are on the same line as Java source.
 - c. Use meaningful names for variables and methods. Variables x, y, z, while easy to type, are not easy to follow through code. Capitalization conventions and the use of underscores (`_`) within resource names is a matter of personal choice; be consistent once you make a choice.
4. **Follow Struts design guidelines for Struts portlets.** Struts is an emerging standard for Model-View-Controller Web application design and implementation. The Struts framework has been adapted to the portlet development environment and is available as a plugin (stand-alone WAR file) for WebSphere Portal. This support lets you incorporate Struts into your portlet application without having to work through the details of including Struts support in WebSphere Portal. See to the [Resources](#) section

for links to detailed information on Struts and how it works with WebSphere Portal. If you develop portlets based on Struts, use these additional guidelines:

- a. Before testing Struts support in your portlet, use the sample portlet applications that accompany the Struts Portal Framework package to ensure Struts is working properly in your environment.
 - b. Review the documentation on the Struts Portal Framework for application requirements and any restrictions. For existing Struts applications refer to the section in the *Migrating an Existing Struts Application* listed in [Resources: Struts support for portal](#).
5. **Categorize the portlet state early in the design phase.** As mentioned in the introduction, the JSR 168 supports different kind of states. The portlet programmer should very carefully and early on decide the category of information for each state. The categories are: navigational state, session state, persistent state.
- a. Use *navigational* state for all view-related data that will let the user navigate forwards and backwards using the browser buttons. The scope of navigational state information is the current request. Examples of navigational state information include the current selected article in a news portlet, and the current selected stock quote for which the portlet should render more details.
 - b. Use *session* state for all information which is relevant for the duration of the user session. Do not use session state as caching store. See also the [Session management](#) section. An example of session state information is the content of a shopping cart.
 - c. *Persistent* state has a life-time beyond the current user session. Use it to store customization data and user-specific personalization data. Examples include the server to retrieve the stock quotes from, the default list of stock quotes to display, or the news topics of interest for a specific user.
6. **Minimize navigational state information.** The navigational state of all portlets on the current page must be aggregated, and it is normally stored in the URL. Therefore, keep the navigational state that the portlet stores in the render parameters to a minimum in order to keep the URL small. Also, most small devices only support a very limited URL length.
7. **Internationalize the portlets using resource bundles.** Use a resource bundle per portlet to internationalize the portlet output, and declare this resource bundle in the portlet deployment descriptor.
8. **Only declare J2EE roles in your portlet application if absolutely necessary.** J2EE roles are separate and need to be managed separately from the portal. Therefore J2EE roles should only be used to perform access control in portlet applications if the user profile information is not sufficient.
9. **Provide version information.** In order to enable portal administrators to differentiate between different portlet application versions and to provide update support, declare the version of your portlet application in the `META-INF/MANIFEST.MF` using the

`Implementation-Version` attribute. Use the recommendation of the Java Product Versioning Specification for the version string with *major.minor.micro*, where:

Major version numbers identify significant functional changes.

Minor version numbers identify smaller extensions to the functionality.

Micro versions are ~~even~~ finer grained minor versions.

These version numbers are ordered with larger numbers specifying newer versions.

Example:

```
Implementation-Title:myPortletApplication
Implementation-Version:1.1.2
Implementation-Vendor:myCo
```

10. **Use the P3P user profile attributes whenever possible.** When a portlet needs to access user profile attributes, such as the user name or address, it should always use the keys that P3P defines for these attributes. These are listed in the Java Portlet Specification Appendix D.
11. **Use URL encoding for resources inside the portlet application WAR file.** In order to allow the portal to proxy resources of remote portlets inside the portlet application WAR file, encode these links using the `encodeURL` method.
12. **Do not use URL encoding for resources outside the portlet application WAR file.** In order to reduce the workload on the portal server, URLs outside the portlet application should not be encoded to let the client directly access the resource outside the portlet application WAR file.
13. **Do not spawn unmanaged threads from portlets.** Spawning new threads from portlets will result in unpredictable behavior. Currently J2EE does not provide an interface for portlets or servlets to spawn new threads. Therefore, managed threads can only be created using the proprietary IBM WebSphere Application server `async` bean interface by the portlet.
14. **Prepare portlet for parallel rendering.** In order to enable the portal to render the portlet in parallel with other portlets on the page, the portlet should:
 - a. Expect `IOExceptions` when writing to the `OutputStream` or `PrintWriter` and act accordingly. If a portlet takes too much time for rendering its content, the portal may cancel this portlet's rendering. An `IOException` will result when the portlet tries to write to the `OutputStream` or `PrintWriter` after the portal has canceled the rendering of this portlet.
 - b. In methods that are expected to take many computation cycles, the portlet should periodically check if the `flush` method of the `OutputStream` or `PrintWriter` throws an `IOException`. If the `flush` method throws such an exception, the portal has canceled the rendering of the portlet, and the portlet should terminate its current computation.
15. **Do not use RenderURLs for form POSTs.** The render phase should not change any state, but should provide a re-playable generation of the markup. Therefore, `HTTP`

POST requests that submit forms should always be handled in an action by creating an `ActionURL`, not in render. The portlet can then be used as a WSRP service, because WSRP does not support new parameters in a render request. The only exception from this rule is when the form does not consist of any parameters (such as a cancel button).

16. **Do not depend on extensions.** If your portlet uses extensions it should also be coded to run in a plain JSR 168 environment with no extensions. Degradation of functionality is acceptable when running in a plain JSR 168 environment. The portlet should check at runtime the support extensions of the calling portal using the `PortalContext` and act accordingly.
17. **Use the IBM extension markup property with care.** The IBM JSR 168 implementation provides the portlet with a markup property for each request that gives the portlet additional information about the expected markup to return. The property was introduced to allow the portlet further markup distinctions that go beyond the pure MIME type (for example, `cHTML` and `HTML`). Program the portlet to also work on portals that do not provide this property.

Code sample:

```
if((markup=RenderRequest.getProperty("wps.markup")) != null) {
    // property supported
    // insert code that depends on markup
} else {
    // property not supported
    // insert code that depends on
    // RenderRequest.getResponseContentType()
}
```

18. **Do not name portlets and servlets the same within a Web application.** Tools and portals can use the portlet name to identify the portlet in a Web application and may get confused if the Web application includes a servlet with the same name.

JSP coding guidelines

This section contains general JSP coding guidelines with a special emphasis on HTML. For guidelines for other markups, see [Multiple markup support](#).

1. **Include HTML fragments only in JSPs.** Because portlets contribute to the content of a larger page, they should only provide HTML fragments and should not have `<HTML>`, `<HEAD>`, or `<BODY>` tags. Make sure the fragments are well-formed to prevent unbalanced pages. Also, remember that the HTML fragment is being added to a table cell (`<TD>...</TD>`) in the portal page.
2. **Design the view to fit on a page with other portlets.** Unlike servlet-based applications, portlets contribute a portion of a larger page. The size of the JSPs (in terms of horizontal and vertical span) can determine how easily the portlet can fit on a page with multiple columns and other portlets. A large portlet will “squeeze” other

portlets off the screen and create large scrolling regions, resulting in usability issues. Therefore, when designing a portlet's JSPs, avoid unnecessary layout elements, focus the portlet's view on the pertinent information, and consider whether the portlet is intended to be placed on pages with other portlets.

3. **Use Java style comments instead of HTML style.** HTML comments remain in the rendered content, adding to the document size and the amount of data that passes to the client. If you use Java style comments within your JSPs, instead, they are removed from the rendered source, along with the rest of the Java code. You must embed these comments within scriptlets:

```
<% // this is a comment %>
```

4. **Use portlet style classes instead of specific style-oriented attributes.** Portal administrators and users can affect the look and feel of the portal by changing the portal theme and the portlet skins. Portlets can pick up on style changes by using styles defined in the portal theme's cascading style sheet (`Styles.css`). For example, instead of decorating an `<INPUT>` element yourself, refer to the theme's input class definition:

```
<input class="wpsButtonText" type="submit">
```

`Styles.css` is loaded by the theme and should not be reloaded by a portlet. Refer to the `Styles.css` file for class definitions and associated values.

5. **Make pages fully accessible.** To allow portal users with disabilities to be able to use your portlet, the JSPs should be fully enabled for keyboard-only control and other assistive technologies. For a complete set of accessibility options for HTML, see [Resources](#). Examples of accessibility enablement features include:
 - a. Use `ALT` attribute with images to define descriptive text of the image content.
 - b. Use `<LABEL>` tags to associate labels with form input controls, so that page readers will be able to associate prompts with inputs.
 - c. Do not use color alone to denote state or information. For example, using red to emphasize text does not help those who are color blind. Use **bold** or *italics* instead, or use color in conjunction with graphic changes.
 - d. Do not use voice or other sounds to convey information.
6. **URIs, HTML element name attributes, and JavaScript resources must be namespace encoded.** Since many portlets can exist on a page, and it is possible that more than one portlet will produce content with like-named elements, there is a risk of namespace collision between elements, causing functional problems with the page. Use the `<portletAPI:encodeNamespace/>` tag to encode such resources with the portlet instance name. For example, to add the tag to a `FORM` name:

```
<FORM method=POST name="<portletAPI:encodeNamespace value='form1' />"  
action="<%=returnUri%>">
```

When the portlet is rendered on the page, the above `<FORM>` tag would look something like this:

```
<FORM method=POST name="PC_202_form1"
action="/wps/myportal/.cmd/ad/.ar/19807697/.c/201/.ce/502/.p/502">
```

7. **Minimize dependencies on JavaScript.** Because JavaScript implementations and behavior differ widely among browser types and versions, the more your portlet depends on JavaScript, the more browser-dependent your portlet becomes. Additionally, the more of the page that is rendered using JavaScript, the more difficult it is to maintain and to extend to markups other than HTML. Also, it is more difficult to namespace encode JavaScript resources and nearly impossible to properly encode (`response.encodeUrl()`) URLs built using JavaScript.
8. **Do not use pop-ups.** Interactions within the portal are state-based, which means that the portal tracks your trail through the pages and portlets. Using the browser's back button, or creating pop-up browser instances containing portlets, can cause the portal to lose track of your current state and cause problems. Other than using JavaScript prompts, there is no safe way to spawn pop-ups within the portal, unless the new link takes you to an external page, outside the portal. The alternative is to link to an external page within a new browser window (using the `TARGET` attribute on the anchor), so that the user is left within the portal in the original browser window.
9. **Use taglibs whenever possible.** Encapsulating Java code within taglibs not only lets you easily reuse common view functions, it keeps the JSPs clean and makes them more like normal HTML pages. The page designer can concentrate on layout and decoration, and you reduce the possibility of breaking the embedded Java code.
10. **Use IFRAMEs with caution.** `IFRAMEs` are an easy way to include external content within a portlet, but undermine the whole portlet principle because the portlet API is just tunneled or side-stepped. Therefore, `IFRAMEs` should only be used for very special cases, such as surfacing legacy applications. Other potential issues to consider when using an `IFRAME` are:
 - a. The `IFRAME` fills its content based on a URL. The URL must be addressable by the browser, therefore the server that is the target of the URL must be accessible by the browser.
 - b. Not all browser levels support `IFRAMEs`.
 - c. If the content is larger than the `IFRAME` region, then enable horizontal and vertical scrolling to let the user scroll through the embedded content. Content which contains scrolling regions itself can make it difficult for the end user to manipulate all scrolling regions to view all embedded content, causing usability problems.
11. **Use JSTL instead of re-inventing common tags.** The Java Standard Tag Library (JSTL) defines many commonly needed tags for conditions, iterations, URLs, internationalization and formatting. See also the *Portlet development basics* topic in the WebSphere Portal V5.0.2 InfoCenter.

Portlet packaging guidelines

1. **Make common functions available externally to portlets.** If the portlet contains common functions that are replicated across several portlets, consider isolating them so they are externally accessible by the portlets. The easiest way to do this is to build another JAR file of the common classes and place the JAR file in each portlet application or in a location that is in each portlet application classpath, such as the `PortalServer/shared/app` directory.
2. **Group portlets that operate on the same backend data into one portlet application.** Leverage the portlet application concept by grouping portlets into one portlet application which operates on the same backend data. The portlets of this application can share configuration settings, such as the backend server name or `userid/password`, and data, such as a date of interest to the user.

Data management

There are different kinds of portlet configuration data.

1. **Use portlet initialization parameters for storing static information not meant to be changed by administrators or users.** These data are specified in the portlet deployment descriptor using the `init-param` tag and are read-only for the portlet. The portlet can access these data using `PortletConfig.getInitParameter`. For example, declare the JSP names and directories that are used for rendering in the `init` parameters to be able to switch to different JSPs without needing to re-compile the portlet.
2. **Use read-only portlet preferences for storing configuration data.** Configuration data is user independent and should only be changed by administrators in the config portlet mode; for example, the name of a news server. Declare configuration data in the portlet deployment descriptor using the `preference` tag, together with the `read-only` tag. The portlet should also provide one or more config screens to allow the administrator to change this setting.
3. **Use writeable portlet preferences for storing customization data.** Customization data is user specific data that the user can set, in edit mode, to customize the portlet output; for example, news of interest to the user. Declare customization data in the deployment descriptor and provide some meaningful default values as writeable portlet preferences. The portlet should provide one or more edit screens to allow the user to change these values.
4. **Write persistent data only in the action phase.** The portlet must only change persistent state information in the action phase. The render phase must be completely re-playable to avoid issues with the browser back button and bookmarkability.

5. **Use String arrays for preference lists.** The JSR 168 lets you define preferences as Strings or String arrays. Use String arrays if you have a list of preference values for one key.

Session management

1. **Limit the use of the portlet session for storing portlet state information.** The PortletSession is a convenient place to store global data that is user and portlet specific and that spans portlet requests. However, there is considerable overhead in managing the session, both in CPU cycles as well as heap consumption.

Because sessions may not expire for quite some time, the data contained in the sessions will remain resident in active memory even after the user is finished with the portlet. Be very judicious about what is stored in the portlet session.

Data which is appropriate for storing in the portlet session is that which is user specific and cannot be recreated by any other means, such as portlet state information. However, parsed configuration data (from portlet preferences) should not be stored in the Portlet Session since it can be recreated at any time.

2. **Do not rely on portlet sessions if the portlet is to allow anonymous access.** If your portlet is to be used by unauthenticated users, such as on the Welcome page, then the portlet should not rely on the portlet session at all. Portlet sessions are bound to a user context. By default, for unauthenticated users, a portlet session can be requested by a portlet, but it is a temporary session which only lasts for the duration of the current request. Public session support exists, where sessions are issued to unauthenticated users, but because there is no log out action for an unauthenticated user, the session will not be invalidated until it times out, which can lead to severe memory consumption issues.

You can preserve global data for anonymous access by:

- a. Storing data in a collection whose key is communicated to the client using a cookie.
- b. Storing data in the WebSphere Application Server's dynamic fragment cache facility, again, using a unique key which is communicated to the browser using a cookie.
- c. Storing the state information as a cookie itself.
- d. Storing non-sensitive data as hidden field elements within FORMs so that it gets passed back on subsequent form submission. This practice has the added advantage of binding state-aware data to the page requiring it.

Alternatively, consider limiting the function of the portlet for anonymous access. You can check for the presence of a user object, `PortletRequest.getUser()`, to see if a user has logged into the portal, in which case, a portlet session should be available.

3. **Always request an existing portlet session.** Always request a portlet session using either `request.getPortletSession()` or `request.getPortletSession(false)`, which will only return a session if one does not already exist. This practice helps prevent the case where a temporary session is generated for a portlet during anonymous (unauthenticated) access.
4. **Prevent temporary sessions from being generated in the JSP.** Add the JSP page directive `<%@ page session="false" %>` to the JSP to prevent temporary sessions from being created by the JSP compiler if none already exist. This practice will help guard against attempting to use the session to store global data if the session will not exist past the current request. You will need to be sure the Portlet Session exists before trying to use it.
5. **Be aware of session timeouts.** Because each portlet application is a separate Web application, each portlet application will have its own session. This results in different timeouts for different portlets on a page, because the user may interact with some portlets more frequently than with other.
6. **Use attribute prefixing for global session scope.** JSR 168 lets use write the portlet into the Web application session, without any prefixing of the portlet container, using the application scope portlet session setting. You can use this support to share data between a portlet and other portlets, or servlets, of the same Web application and among several entities created out of the same portlet. The portlet must take into account that there may be several entities of it on the same page and that the portlet may need to prefix the global setting to avoid having other entities overwrite this setting. One convenient way to do this is provide a read-only portlet preference entry called session-prefix that the administrator can set in the config mode.

One example where this may be needed is a group of office portlets (such as calendar, todo, mail) that want to share the date that the user has selected in the calendar to display the emails of that day and the todo's of that day. Because there could be two instances of these portlets on the same page (for example, one configured for the office mail server and the other configured for the private mail account), these portlets need to put some prefix in front of the global session attribute.

WebSphere Portal allows cloning portlet applications within one Web application; therefore, the prefix should also consist of the portlet application name that the portlet can access using the `PortletContext.getPortletContextName` method.

Summary:

- **Use no prefixing only if the data needs to be shared among all portlet application entities of one Web application.** This is the exception case, because most portlets intend to share data only within one portlet application.

Code sample:

```
portletSession.setAttribute(attributeName, attributeValue,
```

```
PortletSession.APPLICATION_SCOPE)
```

- **Use application prefixing for sharing data among all portlet entities in one portlet application.** Use only when all entities of a portlet need access to the same data, otherwise see bullet below.

Code sample:

```
portletSession.setAttribute(PortletContext.getPortletContextName()  
+ attributeName, attributeValue,  
PortletSession.APPLICATION_SCOPE)
```

- **Use application and config param prefixing for sharing data in a portlet application, but not between different entities of the same portlet.** This is considered to be the default case of sharing data in the session with global scope.

Code sample:

```
portletSession.setAttribute(PortletContext.getPortletContextName()  
+ PortletPreferences.getValue("session-prefix", "default")  
+ attributeName, attributeValue,  
PortletSession.APPLICATION_SCOPE)
```

Internationalization

1. **All portlet strings should be fetched from resource bundles.** All displayable strings should be stored in resource bundles and fetched using the `ResourceBundle` Class or in JSPs using the JSTL internationalization tags. The resource bundles should be organized by language under the portlet's `WEB-INF/classes` directory and the portlet should always provide a default resource bundle with only the base name to enable the fallback mechanism for locales not supported by the portlet.

For example, if a portlet supports English and Japanese, it would have three resource bundles (default, English, and Japanese) and might be organized under `WEB-INF/classes` like this:

```
WEB-INF/classes/nls/mystrings.properties  
WEB-INF/classes/nls/mystrings_en.properties  
WEB-INF/classes/nls/mystrings_ja.properties
```

Using the `ResourceBundle` class, you would refer to the properties file as `"nls.mystrings"`. Java will automatically look for the appropriate properties file based on the current locale.

2. **All view JSPs should be language-independent.** Even though JSPs can contain translated strings, they can be complicated enough to make managing several language-dependent versions impractical. Displayable strings should, therefore, be stored in resource bundles, and then rendered in the JSP using the JSTL `<fmt:taglib>`. Similar to portlet strings, the bundles should be organized by language under `WEB-INF/classes`. For example:

```
<fmt:setBundle basename="nls.mystring"/>
```

```
<fmt:message key="msg10" />
```

3. **Be sensitive to cultural-specific formatting.** Instead of using the local system's locale for referencing a resource bundle, use the locale specified on the portlet request, `PortletRequest.getLocale()`. Then, show your content in the language preferred by the portal user.
4. **Define preferences names as reference names to the resource bundle.** The names of the preferences in the portlet deployment descriptor should be references under which the localized name for this preference is stored in the resource bundle. The portlet should use the naming convention defined in the Portlet Specification section 14.3.1 for the resource bundle entries.

Entries for preference attribute descriptions should be constructed as:

```
'javax.portlet.preference.description.<attribute-name>'
```

where `<attribute-name>` is the preference attribute name.

Entries for preference attribute names should be constructed as:

```
'javax.portlet.preference.name.<attribute-name>'
```

where `<attribute-name>` is the preference attribute name.

These values should be used as localized preference display names.

Entries for preference attribute values that require localization should be constructed as:

```
'javax.portlet.preference.value.<attribute-name>.<attribute-value>'
```

where `<attribute-name>` is the preference attribute name and `<attribute-value>` is the localized preference attribute value.

5. **Define the supported locales in the deployment descriptor.** In the portlet deployment descriptor the portlet should define all locales it supports using the `<supported-locale>` element, which enables the portal to show users only portlets for the locales they have selected.

Inter-portlet communication

The only inter-portlet communication available for JSR 168 portlets is the sharing of data through the session. This mechanism is explained in the [Session management](#) section .

Multiple markup support

There are potentially many different types of browsers, or user agents, which access WebSphere Portal, including the typical desktop browser, hand-held devices (such as

PDA's or personal data assistants) and wireless phones. The capabilities of these devices vary widely, as do their users' attention spans and interaction models. Therefore, the portlet's design needs to take these facts into consideration.

Information priority is different depending on the device on which it is viewed. Capabilities of desktop browsers are limitless. Users can download almost anything quickly and randomly navigate through it. For handheld devices, however, screen real estate and device memory is at a premium, not to mention the relatively slow speed of a wireless connection versus a LAN connection.

Users of mobile and handheld devices need quick access to concise information and cannot afford the time or effort it takes to navigate through several screens to get to the desired information. When designing a portlet for mobile use, you need to understand what is most important to show to a mobile user, and then determine the best way to show it.

1. **Keep the view between 4 and 5 decks.** The mobile device typically has limited storage capacity and slower connection speeds. You may need to break portlet views into decks, or fragments, which are served to the device on demand. Provide links between each fragment to allow for forward and backward navigation.
2. **Fit the content to the device's viewing area.** It can be tedious to scroll horizontally through content which may only show a few lines at a time within a device's viewing area. Keep the view vertically oriented to eliminate horizontal scrolling as much as possible. Also avoid constructs which promote the full use of the page's width, such as tables.
3. **Eliminate unnecessary clutter.** Focus on the priority data. Extraneous information makes finding the important information difficult, and it consumes precious memory space. Some examples of extraneous items include inline images, titles, captions, ads, and related links.
4. **Put links at the end of the document.** Never place links inline with text which can make it difficult to navigate all of the text to get to the right link. Consistently placing relevant links at the end of the document teaches the user how to quickly navigate your portlet.

Performance considerations

1. **Do not spawn threads.** Because portlets are multi-threaded, spawning child threads can create problems with synchronization or multi-threaded access to the spawned threads. Use threads with extreme caution, and when necessary, use them briefly. Do not use long running threads, especially any that outlast a request.
2. **Do not use threads to access J2EE resources.** In certain Java Virtual Machine (JVM) implementations, such as on zOS, there are a limited number of threads in the process thread pool which are allocated for accessing J2EE resources, such as JCA

connectors. All such resources should be manipulated on the calling thread to minimize the possibility of starving the thread pool.

3. **Limit temporary storage.** Temporary storage includes temporary variables created within a block of code, which are used and then discarded. Temporary variables are typically utilitarian in nature, such as Strings, integers, booleans, Vectors, and such. However simple in nature, temporary variables take CPU cycles to create and destroy, and they occupy space on the heap which must be maintained by the garbage collector.

Follow these guidelines to reduce or optimize temporary storage:

- a. Reuse temporary variables as much as possible.
 - b. Declare temporary variables outside loops.
 - c. Instead of `String`, use `StringBuffers`, which are optimized for parsing and string assembly.
 - d. Declare collection classes (`Vectors`, `Arrays`) with an initial size that is the average maximum size, to prevent growth and reallocation of space on the heap.
4. **Avoid synchronized methods.** Synchronization causes a bottleneck through your portlet, or a path that only allows one thread at a time to pass through. Besides slowing the entire portal system, the possibility of a deadlock occurring is also high, which could hang the portlet, or portal, for all users.
 5. **Avoid long-running loops.** Simply put, portlets need to be fast. Long running loops consume a large number of CPU cycles and cause the portal page to wait for this one portlet to finish. If a long-running loop seems necessary, re-inspect the design to see if it can be accomplished by some other means, such as through block/notification logic, or breaking the large loop into several shorter ones.
 6. **Use JSPs instead of XML/XSLT.** JavaServer pages are more efficient than XML/XSLT in general. Because JSPs are compiled into servlets, they run much faster than having to parse XML and apply XSL stylesheets on every request. Also, the portal infrastructure is optimized around JSPs, allowing for easy expansion into other markups, languages, and browser support by simply adding new directory structures of JSPs.

In general, XML parsing and XSLT processing is expensive. XSL processing, for example, causes hundreds of transient String objects to be created and destroyed, which is expensive in CPU cycles and heap management. If XML/XSLT is a requirement, then make use of caching as much as possible to reduce the amount of parsing and XSLT processing which must take place.

7. **Use caching as much as possible.** If portlet output is static in nature or is valid for some period of time before it is updated, then your portlet should enable caching. Define a default expiration time in the portlet's deployment descriptor using the

`<expiration-cache>` tag. During runtime the portlet can attach an expiration time for each response individually using the `RenderResponse.setProperty(EXPIRATION_CACHE, time)` call.

Security considerations

1. **Use the Credential Vault to store sensitive data.** The Credential Vault is designed to securely store “secrets”, or pieces of data necessary for accessing applications, such as user IDs and passwords. Entries in the vault are called slots, and slots can be shared or non-shared. Shared means that the secret can be used by any user of that portlet. Non-shared means that every user must have his or her own secret stored in the slot.

There are two types of credentials: passive and active. Passive credentials give the portlet programmer access to the actual user ID and password. Active credentials give the portlet programmer access only to services enabled by the credential, such as a secure connection through an authentication proxy; the does not actually see the secret itself.

Using the Credential Vault enables single sign-on to avoid multiple login challenges and improve the user’s experience with the portal.

2. **Appropriately handle data which is passed to the client.** HTTP connections may not be secured, and certain browsers, especially on mobile devices, may not secure transmitted data effectively. So, if you use HTTP headers, such as cookies, to pass application data to the client, be sure that it is either encrypted appropriately, or is benign in nature and does not require securing.
3. **Enable single sign-on as appropriate.** Many customers gain access to the portal through a trust association interceptor (TAI) which translates user identity as established by some secure network intermediary into a portal identity. Your portlet needs to be able to either pass third-party user identity information to back-end applications that are enabled for the same authority, or to translate the portal user identity into external application user identity, such as through the use of the Credential Vault.

Remote portlet considerations

In order to seamlessly support offering a portlet as a Web Service for Remote Portlets (WSRP) service, the portlet only needs to adhere to the JSR 168 specification and must not use RenderURLs as HTTP form POSTs (see [Portlet coding guidelines, item 15](#)).

Portlet documentation

Even the best written portlet can be useless if it is not documented correctly. Use the following guidelines for documenting how administrators can use your portlet.

1. **Document all context parameters and configuration parameters.** Even if the values should not be changed, the parameters are displayed under Portal Administration, which allows administrators to change the values or delete the parameters altogether.
2. **Document how the portlet uses caching.** This information might allow the administrator to further optimize the portlet caching. For JSR 168 portlets, inform the administrator that the portlet should not be placed on a page with portlets that are cachable.
3. **Document portlet session requirements.** If the portlet requires the existence of a valid session, instruct the administrator not to place it on an anonymous page.

Summary

This paper described best practices for portlet development related to JSR 168. The practices were presented in different levels and provide guidance on how to:

- Structure your portlet application
- Program portlets in a portable, high performing, and maintainable fashion
- Program your JSPs that are included from your portlets.

You saw best practices for transient and persistent storage of data, internationalization, performance and security considerations.

You can use these guidelines to program JSR 168 portlets without falling into pitfalls that will cause you great trouble later on when you need to enhance your portlet application, run it as remote portlet, or run on servers with a high load.

Resources

The following resources provide additional information related to programming portlets using JSR 168 and WebSphere Portal.

Standards and specifications

JSR 168 Portlet Specification

<http://jcp.org/en/jsr/detail?id=168>

Web Services for Remote Portlets Specification

<http://www.oasis-open.org/committees/download.php/3343/oasis-200304-wsrpspecification-1.0.pdf>

Java Community Process

<http://jcp.org>

WebSphere Portal

WebSphere Portal V5.02 InfoCenter

<http://publib.boulder.ibm.com/pvc/wp/502/index.html>

Portlet Development Guide for IBM Portlets: Portlet Development Best Practices and Coding Guidelines

<http://www.ibm.com/developerworks/websphere/zones/portal/portlet/portletdevelopmentguide.html>

Portlet API Javadoc for WebSphere Portal 5.0

<http://www.ibm.com/websphere/developer/zones/portal/portlet/5.0api/WPS/>

Portlet JSP Tag Library Syntax

<http://www.ibm.com/developerworks/websphere/zones/portal/portlet/V41jsptaglib.html>

Portlet JSP Tag Quick Reference (Version 5.02)

http://publib.boulder.ibm.com/pvc/wp/502/ent/en/InfoCenter/wps/wpsbsoutput.html#jsp_tags

developerWorks WebSphere Portal zone

<http://www.ibm.com/developerworks/websphere/zones/portal/>

Struts support for portal

OpenSource

<http://jakarta.apache.org/struts/index.html>

Struts in WebSphere Portal 4.1

<http://www.ibm.com/support/docview.wss?rs=0&org=SW&doc=7002247>

Developing and Deploying a Struts Application as a WebSphere Portal V5 Portlet

http://www.ibm.com/developerworks/websphere/library/techarticles/0401_hanis/hanis.html

Related topics

WebSphere Portal-related Redbooks

<http://publib-b.boulder.ibm.com/cgi-bin/searchsite.cgi?Query=Portal%20OR%20portlet&SearchMax=4999&SearchOrder=4&SearchFuzzy=FALSE>

Building a Portlet within the Model-View-Controller Paradigm using WebSphere Portal

http://www.ibm.com/developerworks/websphere/library/techarticles/0210_kwong/kwong.html

WebSphere Portal Programming: Portal Aggregation for Pervasive Devices

http://www.ibm.com/developerworks/websphere/techjournal/0210_godwin/godwin.html

WebSphere Portal Programming: Pervasive Portlet Development

http://www.ibm.com/developerworks/websphere/techjournal/0207_wanderski/wanderski.html

Comparing the Java Portlet Specification JSR 168 with the IBM Portlet API

http://www.ibm.com/developerworks/websphere/library/techarticles/0312_hepper/hepper.html

Design Patterns



E. Gamma, R. Helm, R. Johnson, J. Vlissides. Addison-Wesley, 1995.

Accessibility standards

IBM Accessibility Center

<http://www.ibm.com/able/index.htm>

About the authors

	<p>Stefan Hepper is a software architect in the WebSphere Portal team at the IBM Boeblingen Lab in Germany. He is responsible for Java standards and is one of the two specification leads for JSR 168. He has conducted lectures at international conferences, published papers, filed patents, and was a co-author of the book <i>Pervasive Computing</i> (Addison-Wesley 2001). You can reach Stefan at sthepper@de.ibm.com.</p>
	<p>Marshall Lamb is a Senior Software Engineer and Chief Programmer for WebSphere Portal V5. He lead the Business Portlet development team for WebSphere Portal and the WebSphere Transcoding Publisher product for several years. Marshall started in networking software development, working through the Host Integration product line, before moving into the Pervasive Software Division (WTP), and finally into Lotus Software with WebSphere Portal. You can contact Marshall at mailto:wsdd@us.ibm.com.</p>

Trademarks

- DB2, IBM, and WebSphere are trademarks or registered trademarks of IBM Corporation in the United States, other countries, or both.
- Windows and Windows NT are registered trademarks of Microsoft Corporation in the United States, other countries, or both.
- Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
- Other company, product, and service names may be trademarks or service marks of others.

IBM copyright and trademark information: <http://www.ibm.com/legal/copytrade.phtml>