Learn all about the second version of the Java Portlet Specification (JSR 286). In Version 2.0, the specification and the APIs more than doubled, and it allows you to implement most use cases without the need to have vendor extensions. The portlet programming model also provides events and public render parameters so that you can build larger composite applications out of your portlets and reuse your portlets in different scenarios.

Portlets are componentized user-facing applications that create some kind of markup. This markup is intended to be aggregated with other markup fragments into a larger application, for example, a portal page as shown in figure 1.

**Figure 1. A sample portal page**

Portlets can therefore be viewed as user interface (UI)-based services that take the Service-Oriented Architecture (SOA) approach all the way up to the user interface.
Java portlets started to become popular after the first version of the Java Portlet Specification, the Java Specification Request (JSR) 168, was finished in 2003 at the Java Community Processes. Since then, nearly all the vendors in the Java portal space, both commercial and open-source vendors, have implemented this standard, and developers have written portlets using the Java Portlet API.

The JSR 168, however, stopped at defining the overall UI component model and did not define any means for building integrated composite applications out of these components. This limitation, and many other things that didn't make it into V1.0 due to time constraints, is now addressed in V2.0.

Work on JSR 286 started in January 2006, and the final version was submitted in February 2008. The expert group of JSR 286 consists of all the major commercial and open-source portal developers, portlet bridge developers, and portlet framework developers. You can find a complete list of the members of the expert group [here](#).

This article provides an overview of the main new features in JSR 286 and some examples of how to leverage these features. This article assumes basic knowledge about the portlet programming model defined in version 1.0; introductory information about JSR 168 is available.

**Building composite applications using coordination**

The main new features of version 2.0 provide capabilities for coordination between different portlets that can be implemented by different parties and packaged in different WAR files. The coordination features of JSR 286 are based on a loosely coupled publish/subscribe model that does not require that the different portlet developers know about each other's work. At development time, you define only the data that a portlet understands, and the actual connections between portlets are created at deployment or runtime. Leveraging these coordination features, portal administrators or business users can now build larger composite applications, out of portlet components without programming. This capability opens the possibility to create new functionality by combining existing components into business mashups in a similar fashion to popular Web 2.0 usage – even in ways that the component authors may never have envisioned themselves.

JSR 286 defines two different means for coordination, addressing two different use cases:

- **Events.** Support for active notifications with action capabilities to the portlet.
- **Public render parameters.** Support for sharing of view state across portlets.

We explain both of these use cases in more detail in the following sections.

**Events**

JSR 286 allows portlets to send and receive events. As mentioned before, the model is a loosely coupled model where the portal application acts as broker between the different portlets and distributes the events.
Figure 2. Event distribution from Portlet 1 to Portlets 2 and 3

Figure 2 depicts the details of how this distribution works. Portlet 1 defines that it can publish some event A in its portlet.xml deployment descriptor. Portlets 2 and 3 define that they are able to receive event A in their portlet.xml deployment descriptors. The portal administrator or business user has now put these three portlets on a page and created connections between the sending portlet Portlet 1 and the receiving portlets Portlet 2 and Portlet 3. Now the user interacts with Portlet 1 and triggers an action on Portlet 1. As a result of this action, the portlet issues event A. After the portlet has finished its action processing the event broker component, which is typically part of a portal runtime, it looks for targets of this event. In our sample, there is a connection between Portlet 1 and Portlets 2 and 3 for this event. Thus, the event broker calls the processEvent life cycle method introduced with version 2.0 on Portlet 2 and 3 with event A as payload. After the action and event processing have finished, the rendering lifecycle starts and the portlets can create new markup based on the state updates that occurred in the action and event phases.

JSR 286 does not specify how connections for portlet coordination are defined and managed. Typically, connections are defined explicitly at page creation time, or they are automatically inferred by the portal at runtime. The first approach allows more control and flexibility, while the second is easier to use. In both cases, the portal needs to know what kind of events a portlet can receive and send. Therefore, you need to provide this information in the portlet.xml deployment descriptor. How would you do this? First, you provide the general event definition, and then you reference this definition in the portlet section with tags that specify that the portlet can either receive or send this event. Each event is uniquely identified by an XML qualified name (QName). QNames are the mechanism for using unique names in XML; they also have a Java representation as the standard library class javax.xml.namespace.QName. A QName consists of a namespace (for example, http://www.ibm.com) and a local part (for example, myEvent).

Listing 1 shows a sample.
Listing 1. Publishing event definition example

```xml
<portlet>
  …
  <supported-publishing-event>
    <qname xmlns:x="http://com.ibm/portal/portlets/ns">x:city</qname>
  </supported-publishing-event>
</portlet>

<event-definition>
  <qname xmlns:x="http://com.ibm/portal/portlets/ns">x:city</qname>
  <value-type>java.lang.String</value-type>
</event-definition>
```

You can specify a full QName for each event, or if you use the same namespace for many events, you can declare it as the default namespace and then specify only the local part of each event.

In addition to the name, you need to specify the type of the event payload to allow the portal application to serialize / deserialize the payload in the case that the portlets are using different classloaders or may even run as remote portlets using Web Services for Remote Portlets (WSRP). Thus, the requirement for the payload is that it is either a simple type (for example, String) or that it needs to be Java serializable and Java Architecture for XML Binding (JAXB) serializable. Why do we need this additional JAXB serialization? A portlet may want to send an event to a remote portlet that is running using WSRP. These remote portlets may not be written in Java, and so we need a more general mechanism than the Java serialization. JAXB allows serializing Java objects into XML; all you have to do is annotate your Java object accordingly.

This detail was the hard part; now your work is getting simpler: receiving and sending events through your portlet. You can send an event using the setEvent method on the ActionResponse or the EventResponse. To receive events you need either to extend GenericPortlet or to implement the EventPortlet interface yourself. Extending GenericPortlet has the additional advantage of being able to use annotations for denoting special methods for handling events, processing actions, and rendering a specific portlet mode.

Listing 2 shows a sample.

Listing 2. Event processing example using annotations

```java
@ProcessEvent(qname="{http://com.ibm/portal/portlets/ns}city")
public void cityEvent(EventRequest request, EventResponse response)
  throws IOException, PortletException
{
  Event ev = request.getEvent();
  if ( ev.getValue().equals("Orlando") ) {
    …
  }
  …
}
```

Public render parameters

Public render parameters allow sharing of request parameters across different portlets or other artifacts, like themes or portal pages in IBM WebSphere Portal. Defining a public render parameter is very similar to defining an event: You specify a unique QName and optional alias names in the
portlet.xml deployment descriptor. The QName and alias names are used to connect public render parameters of different portlets. You can view these public render parameters as a shared storage area where all participating portlets can set and get the values.

**Figure 3. Public render parameter coordination example**

![Diagram](http://portal.com/...docid=docum... http://portal.com/...docid=docum...)

Figure 3 depicts an example where a navigator portlet and a content portlet share the same public parameter docid. After you click a link in the navigator portlet that sets the docid to document3, the content portlet displays the content of that document because its view state (the ID of the current document) has been changed from an external influence. Note that public render parameters can be stored in the URL, as in the WebSphere Portal implementation, and thus you can bookmark pages and use the browser's Back and Forward buttons.

How can the portlet get access to these public render parameters? In the portlet itself, you access the public render parameters with the same methods that are used to handle ordinary render parameters defined in version 1.0. Given that these methods take only Strings as keys and not QNames you need to define an identifier in the portlet.xml that you can then use inside the portlet code for addressing the public render parameter.

In most cases, the portlet code does not actually need to change whether a render parameter is public or private, so you can enable your portlet for coordination by simply declaring those parts of the view state information in the portlet.xml that can reasonably be set from external sources as public parameters. If you want to specifically access only public render parameters in your code, then you can also use the method getPublicParameterMap on the request.

**Serving resources**

In the first version of the Java Portlet Specification, you could not serve dynamically generated resources directly through the portlet. Instead, you needed an additional servlet that served the resources. This limitation had some disadvantages:

- You do not have the portlet context available, which means you have no access to render parameters, portlet mode, window state, portlet preferences, portlet session, and more.
- URLs generated in the servlet are outside the portal scope and leave the portal experience. In addition, the current state of the portal page accumulated in the portal URL is lost.
- The servlet does not run under the portal access control and needs to be secured separately.
One advantage of serving resources through a servlet is less overhead because the request does not have to pass through the additional portal framework, for example, when serving large media streams.

Version 2.0 now introduces a new URL type, resource URLs. Resource URLs trigger the lifecycle method, serveResource, on the ResourceServingPortlet interface that you can leverage to create dynamic resources directly in the portlet.

**Resource URLs**

You must have a resource URL to trigger the new serveResource life-cycle method. Let's start by looking into the details of the new resource URL.

Resource URLs can be created similar to other portlet URLs, with the method createResourceURL on the RenderResponse and ResourceResponse, for example,

```java
ResourceURL url = response.createResourceURL();
```

Now you can set parameters on the URL as you do for other portlet URLs. You receive these parameters in the serveResource call. Note that ResourceURLs cannot set new render parameters, portlet mode, or window state. This restriction occurs because serveResource calls do not generate a full new portal page, but return the response of serveResource. Thus, the portal does not have a chance to update other parts of the page where this information may be encoded; for example, in WebSphere Portal all URLs contain this information and so need to be updated.

You can also set an additional resource ID on the resource URL that clearly identifies your resource. If you extend GenericPortlet, GenericPortlet tries to forward to that resource ID for a serveResource call. You can set the path to your resource as ID, as shown here:

```java
url.setResourceID("WEB-INF/jsp/xmlcontent.jspx");
```

In this case, GenericPortlet automatically dispatches to the given JSP, which can then make use of the portlet state information by including the portlet tag library. Note that static resources like GIF files packaged in your portlet WAR should normally be referenced with static resource URLs such as:

```java
String url = response.encodeURL(request.getContextPath()+"/icons/myigif.gif");
```

Serving static resources using the portlet's serveResource method causes unnecessary performance overhead.

**Serving resources**

To receive a serveResource call triggered by a resource URL, you need to either implement the new ResourceServingPortlet or extend GenericPortlet, which already implements the new interface. In the serveResource call, you get a specific resource request / response pair.

The resource request has similar methods to the render request, but in addition it lets you get access to upload data. In the resource request, the parameter handling lets you have access to three different types of parameters:
• Resource parameters set on the resource URL that are available only for this resource call, accessible through the getPrivateParameterMap call
  • Private render parameters of the portlet, accessible through the getPrivateRenderParameterMap
  • Public render parameters, accessible through the getPublicParameterMap call

The ordinary parameter access methods return the merged set of these three parameter maps. Parameters that have the same key first get the resource parameter values merged and then either the private or public render parameter values, depending on whether the parameter key was declared as a public render parameter in portlet.xml.

With resource URLs, you can leverage all HTTP methods, not only GET, as you do in render. This capability means that you can use methods such as POST or DELETE to change state in the serveResource call. These state changes, though, should be limited to private state of the portlet: portlet-scoped session data and portlet preferences. You should not modify state that affects other portlets because the portal framework does not have a chance to update other parts of the portal page for serveResource calls, so that updates to page-level state may not be visible.

NOTE: Render parameters, portlet mode, and window state fall into this category as some portal implementations, such as WebSphere Portal, store this information in the URL to support the ability to bookmark and the use of Back and Forward buttons in the browser. This implementation means that changing a render parameter, for example, requires an update of all the URLs on the page, which the portal cannot perform, as the processing of the response data on the client is done entirely by the portlet.

Because the markup returned from serveResource is not aggregated with other markup from the portal framework, the resource response allows full control over the output stream. For example, the portlet can set a HTTP status code.

Caching levels of resources

There are many different use cases for serving resources; therefore, there are also different requirements for the ability to cache serveResource calls. You can influence the HTTP caching behavior of resource requests with the setCacheability method on the ResourceURL. This method gives a hint to the portal about how much information the target serveResource call requires. Therefore, the portal can remove irrelevant data from the URL, which results in more stable URLs and more HTTP caching hits for resource requests. Note that setting the cacheability of a resource response makes sense only if you allow the response to be cached by specifying HTTP caching headers for the returned response. This specification means that you must set at least an expiration time using response.getCacheControl().setExpirationTime(). If the returned content is not user specific, you should additionally set the cache control to public scope.

JSR 286 supports the following scenarios:

• Full cacheable resources, which do not depend on interaction state. An example is a programmatically generated SVG view that depends on the portlet preferences for back-end access. To mark a serveResource call as fully cacheable, you need to set the cacheability on
the resource URL to FULL. This setting means that the portal can generate a URL that does not contain any interaction state of the page and the portlets on the page. Thus, the browser can cache the returned markup of the serveResource call for at least as long as the user interacts with the current page.

As a result of not encoding any state of the page and portlets on the page, you cannot count on getting the current render parameters, portlet mode, or window state in your portlet. Keep in mind that because of the missing state information, you are restricted in the output of serveResource: You can include only resource URLs that are fully cacheable, and you cannot include action or render URLs.

- Full and shared cacheable resources such as shared static resources, for example, JavaScript libraries. You can mark a FULL cacheable serve resource URL as shared by adding the additional SHARED property on the URL with the name of the shared resource as a value. The name should be a QName that identifies the resource uniquely, for example {http://dojotoolkit.org/v1.0}dojo.js.

  Specifying the shared property enables the portal to use only one version of such shared resources per page. It still requires that the portlet packages these resources in its WAR file in case the portal has not already loaded the resource.

  Leveraging portal resources that do not need to be provided by the portlet is vendor specific; for example, in WebSphere Portal you can use the ResourceURLAccessor to create resource URLs.

- Portlet-level cacheable resource that depends on portlet interaction state, for example, a dynamically generated PDF output of the current portlet view. If you need to have access to the portlet state, for example the render parameter, portlet mode, or window state, but you are not going to put action or render URLs in your response or in later responses, you set the cacheability level to PORTLET. This level allows refreshing the resource for each interaction with the portlet, but also allows using the cached resource when interacting with other portlets on the page.

- Page-level cacheable resources such as resources providing action or render URLs, for example, the markup returned for an Ajax call. PAGE-level cacheability is the default setting. You have no restrictions in your serveResource method, which means that you can generate action or render links. It also means that your resource cannot be cached across any interaction with the portal page that changes a part of the state of the page.

In summary, these cacheability levels give you the ability to provide as many hints about the cacheability as possible to the runtime. The runtime also can enhance the cacheability of resources in other ways, for example by keeping track of the state changes on the client and re-rendering only the parts affected by state changes. An example for this is the Client-side Aggregation theme in WebSphere Portal V6.1 Beta.

**Using Ajax**

In version 1.0, support for Ajax use cases was rather minimal: You had only the ability to provide an additional servlet that serves the Ajax fragment. This servlet, however, was then addressed directly and not through the portal framework, so it did not have the portlet state provided, nor was it under the security protection of the portal, as you can see in figure 4. The only way to share
data between the portlet and the servlet is to use the application-scope session or the parameter on the URL that invokes the servlet.

**Figure 4. Ajax solution in JSR 168: Serving the Ajax data using an additional servlet**

How does the new portlet specification support Ajax use cases?

As we have seen, JSR 286 provides you with the means to serve resources directly using the portlet. Thus, you can issue XmlHttpRequests to ResourceURLs and, on the server side, get complete access to the portlet context such as render parameter, portlet mode, window state, portlet preferences, and portlet session.

We also learned that you can make some state changes in the serveResource call:

- Change the portlet preferences
- Change data in the portlet session scope

You can now implement additional use cases such as asynchronous updates that make your user interface more responsive.

**Figure 5. Ajax solution in JSR 286: Serving the Ajax data directly from the portlet**

Figure 5 depicts the solution in JSR 286. You can see that the Ajax calls go through the portal servlet and thus are under the control of the portal. You can also leverage resource URLs to provide the links to your JavaScript library.
Closing the gap to the servlet programming model

The first version of the Java Portlet Specification restricted the portlet programming model in some areas compared to the servlet programming model. This restriction was done because portlets are aggregated on a page and all concepts that assume that you are the only component on the page are not easily applied to the portlet programming model.

The second version tackles these issues and provides solutions so that the portlet programming model has nearly the same capabilities as the servlet one, plus the portlet specific extensions.

Cookies, document head section elements, and HTTP headers

In the first version of the Java Portlet Specification, the portlet could not contribute to sections of the portal page outside its portlet window. In the second version, you can set cookies, document head section elements (for example, HTML ones such as meta, link, or style) and HTTP headers (such as application-specific Pragma headers).

Two-part rendering life-cycle call

To overcome the problem that version 1.0 faced with contributing content outside the current portlet window, version 2.0 added a two-part render call:

1. RENDER_HEADER part that allows the portlet to return content outside the current portlet window such as portlet title, preferred next possible portlet modes, cookies, document head section elements, and HTTP headers.
2. RENDER_MARKUP part that allows the portlet to return its normal markup.

These two parts are needed as some portal implementations, such as WebSphere Portal, stream the page and portlet output directly to the client to avoid buffering overhead. In this case, the portlet rendering cannot add anything to the head section of the document, as the head section is already written. Thus, portlets that want to contribute to the head section or to set a portlet title should turn on the two-part rendering feature with the setting shown in listing 3 in the portlet deployment descriptor.

Listing 3. Two-part rendering lifecycle enablement

```xml
<portlet>
  ...
  <container-runtime-option>
    <name>javax.portlet.renderHeaders</name>
    <value>true</value>
  </container-runtime-option>
</portlet>
```

Implementing two-part rendering is straightforward if you extend GenericPortlet and override the methods that GenericPortlet provides, such as getTitle and getHeaders. The GenericPortlet class does the rest for you.

Using cookies
You can set cookies at the response of each life-cycle method (processAction, processEvent, render, and serveResource) with this code: `response.addProperty(javax.servlet.http.Cookie cookie)`

The cookie can then be accessed in all life-cycle methods using: `request.getCookies()`

Using cookies in portlets is different from using cookies in servlets in these ways:

- As mentioned, to set cookies in the render method, the renderHeaders option should be turned on and the cookies should be set in the RENDER_HEADERS part by using overriding doHeaders() of GenericPortlet, for example.
- Cookies may not be accessible at the client because they are stored at the portal server or they are put in a different namespace when the portlet runs as a remote portlet through WSRP.
- Cookies are not guaranteed to be shared across different portlets.

**Request dispatcher include and forward**

In the first version of the Java Portlet Specification, you had only the option of including servlets or JSPs from the render life-cycle call. The second version lets you use forward and include and lets you use them in all life-cycle methods.

This addition means that you can now dispatch to action or event logic written in a servlet, or you can do forwards when serving resources try to forward to the resource ID that you have set on a ResourceURL.

**Leveraging the servlet life-cycle listeners**

In V1.0, the servlet HttpSession listeners can be also used for the PortletSession as the PortletSession is a façade on top of the HttpSession. Version 2.0 extends the list to all servlet life-cycle listeners defined in the Java Servlet Specification V2.5, and it enforces a correspondence between portlet and servlet objects:

- `javax.servlet.ServletContextListener`. For notifications about the servlet context and the corresponding portlet context.
- `javax.servlet.ServletContextAttributeListener`. For notifications about attributes in the servlet context or the corresponding portlet context.
- `javax.servlet.http.HttpSessionActivationListener`. For notifications about the activation or passivation of the HTTPSession or the corresponding PortletSession.
- `javax.servlet.http.HttpSessionAttributeListener`. For notifications about attributes of the HTTPSession or the corresponding PortletSession.
- `javax.servlet.http.HttpSessionBindingListener`. For notifications about binding of the object to the HTTPSession or the corresponding PortletSession.
- `javax.servlet.ServletRequestListener`. For notifications about changes to the HttpServletRequest or the mirrored portlet request of the current Web application.
- `javax.servlet.ServletRequestAttributeEvent`. For notifications about changes to the attributes of the HttpServletRequest or the mirrored portlet request of the current Web application.
A servlet request listener can distinguish a plain servlet request targeted to a servlet from a wrapped servlet request targeted to a portlet by looking at the request attribute `javax.portlet.lifecycle_phase`. This attribute is set on a request targeted to a portlet, indicating the current portlet life-cycle phase of this request.

Access to all these life-cycle listeners gives you many hook points for managing objects related to these lifecycles, which is a great feature for many frameworks. These hook points, though, come with a cost, especially the request lifecycle listeners, which add significant processing overhead for each request. Therefore, use them with care.

**Extending JSR 286**

The expert group of JSR 286 worked to make JSR 286 extensible so that you can add features and functionality in our frameworks on top of JSR 286 in a noninvasive and container-independent manner. You can add:

- Servlet lifecycle listeners
- Portlet filters for wrapping the request / response
- PortletURL listeners for manipulating the URL before it is written to the output stream
- Portlet-managed modes allowing the portlet to provide its own portlet modes

Some extension points require that the container supports this extension point for the portlet to take advantage of the extension. These extension points are:

- Request / response properties from V1.0 for setting extension properties on request or response. Version 2.0 adds several new spec-defined extension properties, such as the predefined caching properties.
- PortletURL properties for extending the PortletURL, for example, the predefined SHARED property for ResourceURLs.
- Container runtime options for leveraging additional container behavior, such as the two-part rendering.

Let’s take a deeper look into the new features for extending the Java Portlet Specification in the next sections.

**Portlet filters**

The new portlet filter functionality allows you to plug filters around any life-cycle call of the portlet. Following the common decorator pattern, filters can do pre- or post-processing, and they can modify or wrap the request and response objects that are passed to the portlet. Typical applications of portlet filters could include these:

- Passing information from additional sources to the portlet as attributes or parameters
- Output filtering for security enforcement or markup compliance
- Collecting diagnostic information
- Bridging between Web application frameworks
WebSphere Portal V6.1, for example, uses a filter approach to augment the markup of portlets with additional semantic information for client-side click-to-action.

The portlet filter programming model is modeled on the servlet filter model:

1. Define the filters in the deployment descriptor. This definition is done using the `<filter>` element, where you also need to state the life-cycle call to which the filter should be applied.
2. Implement the corresponding Filter interface in your filter. You can also list multiple life-cycle entries and implement multiple Filter interfaces with your class.
3. Provide a filter-mapping element where you describe to which portlets your filter should be applied (you can also use an asterisk as a wildcard if it should be applied to all portlets in the application).

The order of the filter-mapping elements in the deployment descriptor also defines the order of the filters that are applied to the portlet. Listing 4 shows an example for a deployment descriptor entry.

**Listing 4. Portlet filter definition example**

```xml
// filter declaration
<filter>
  <filter-name>PortletFilter</filter-name>
  <filter-class>com.example.PortletFilter</filter-class>
  <lifecycle>RENDER</lifecycle>
</filter>

// filter mapping
<filter-mapping>
  <filter-name>PortletFilter</filter-name>
  <portlet-name>MyPortlet</portlet-name>
</filter-mapping>
```

At runtime, a filter chain of all the filters is applied to the portlet. Each filter gets the current request and response, or a wrapped version created by a preceding filter, and the filter chain. After doing its preprocessing, the filter implementation can either terminate the request processing or call the next element in the filter chain, passing in either the received request and response or additional wrappers. The last element in the filter chain is the portlet itself.

Listing 5 is an example of a filter that does some pre- and post-processing and provides the portlet with a wrapped request.

**Listing 5. Example filter implementation**

```
public class PortletFilter implements RenderFilter {
    public void doFilter(RenderRequest req, RenderResponse resp, FilterChain chain) throws .. {
        PrintWriter pw = resp.getWriter();
        pw.write("Pre-processing");

        MyRenderResponseWrapper resWrapper =
            new MyRenderResponseWrapper(res);
        chain.doFilter(req, resWrapper);

        pw.write("Post-processing");
    }
}
```
Note that when you use wrappers, you should stick to the wrapper pattern and only override existing behavior. Do not add new methods on the wrapper; there may be other wrappers in the chain that you don't know of, and thus the portlet may not be able to access your new method. If you want to provide the portlet with additional capabilities, set an object providing access to these capabilities as a request attribute.

**Portlet URL listeners**

In some cases, you may want to centrally manage setting specific properties on portlet URLs or to enhance the portlet URL creation of existing portlets. One example involves implementing a mapping of resources to shared IDs. You implement this mapping only once in the listener, and the listener checks if the resource URL is targeting a resource for which a shared QName exists and then sets the SHARED property with the corresponding QName. For these use cases, the JSR 286 specification gives you the portlet URL listeners.

You need to register such a listener with the listener element in the portlet deployment descriptor, and your class needs to implement the PortalURLGenerationListener interface that defines a callback method for each type of portlet URLs: action, render, and resource.

**Portlet-managed modes**

In JSR 168, the portlet could leverage only portlet modes that are supported by the portal framework running the portlet. In some use cases, the portlet wants to offer portlet-specific functionality with the same user look-and-feel as portlet modes supported by the portal (for example, with context menus on the portlet window). For example, a ShowShoppingCart portlet mode lists all entries that you currently have in your shopping cart.

To support these use cases, JSR 286 introduces the portlet-managed modes that are not known to the portal, but are managed by the portlet itself. The portlet can declare such a mode in the portlet deployment descriptor with the code shown in listing 6.

**Listing 6. Defining a custom portlet managed mode**

```
<custom-portlet-mode>
  <description>Show shopping cart</description>
  <portlet-mode>ShowShoppingCart</portlet-mode>
  <portal-managed>false</portal-managed>
</custom-portlet-mode>
```

The important point here is to set the portal-managed mode to false. This setting indicates to the portal that it should treat this mode just like the standard View mode concerning all aspects, including how it provides the portlet with preferences and from the perspective of access control. The portal should provide UI controls that allow the portlet to switch to this mode.

The portal can define localized names for the decoration using the resource bundle entry: `javax.portlet.app.custom-portlet-mode.<name>.decoration-name`.

Because the portal does not know specific semantics of the portlet-managed modes, it does not know when it makes sense to present a decoration to switch to a portlet-managed mode and when it does not. Therefore, the JSR 286 specification lets you indicate for
which portlet modes the portal should display UI controls as part of the render response using the method `setNextPossiblePortletModes`. When you use `GenericPortlet`, you can override `getNextPossiblePortletModes`, and `GenericPortlet` takes care of setting these in the `RENDER_HEADER` part. Note that you need to set this list of modes on each response and that you should enable the container runtime option `javax.portlet.renderHeaders` mentioned in the next section.

**Container runtime options**

Container runtime options allow the portlet to supply specific options to the portlet container that either change default behavior defined in the Java Portlet Specification or add additional behaviors. We've already seen an example of such a container runtime setting: the `renderHeaders` option. JSR 286 defines a list of container runtime options that are all optional except for `actionScopedRequestAttributes`. In addition to these options, specific portlet container implementations may provide their own options.

You indicate that your portlet requires a specific container runtime option in the deployment descriptor using the code shown in listing 7.

**Listing 7. Container runtime option template**

```xml
<portlet>
  ...
  <container-runtime-option>
    <name>NAME_OF_THE_OPTION</name>
    <value>optionally you can have one or more parameters</value>
  </container-runtime-option>
</portlet>
```

These options are predefined in JSR 286:

- **`javax.portlet.escapeXml`**. Allows you to turn off the XML escaping of portlet URLs for each default, in case you have written a JSR 168 portlet that assumes that URLs are not XML escaped and have migrated this to a JSR 286 portlet.
- **`javax.portlet.renderHeaders`**. Enables the two-part rendering mechanism that lets you set headers in streaming-based portal implementations.
- **`javax.portlet.servletDefaultSessionScope`**. Allows you to change the default scope of the session object provided to servlets or JSPs that are called from the portlet through forward or include from application scope to portlet scope. That way, these resources access the same session data using portlet and servlet APIs, which is particularly convenient if your JSPs are written for a servlet-based Web framework.
- **`javax.portlet.actionScopedRequestAttributes`**. Allows Web frameworks to pass complex objects from the action or event phase to the render phase through the request. You are able to access these attributes until the next request with an action semantic (which may be an action triggered through an Action URL or an event) occurs. This feature likely is implemented by the portlet container by storing these attributes in the session. Therefore, use this feature only if you cannot avoid it, as it will probably cause some performance degradation. This option is the only one that JSR 286 requires to be supported by all containers.
Note that if you use a container runtime option and the container on which your portlet is deployed does not support this option, the container may reject the deployment of the portlet. Use these options with care if you want to develop a portlet that can run on many different portlet container implementations.

**Backward compatibility**

Java Portlet Specification V2.0 was designed to avoid breaking binary code compatibility with V1.0 and to maintain compatible behavior for all API methods. This design point means that all portlets written against the V1.0 specification should run unchanged on a V2.0 container. The only exceptions to this rule are the following slight behavior changes that normally should not break any portlets:

- RenderResponse.setContentType is no longer required before calling getWriter or getOutputStream. Calling getWriter or getOutputStream without previously setting the content type no longer results in an IllegalStateException in V2.0.
- getProtocol for included servlets / JSPs no longer returns null, but instead returns HTTP/1.1 in V2.0.

This backward compatibility statement also includes the deployment descriptor in which V2.0 added new entries, but did not change existing ones. This behavior means that you can turn most JSR 168 portlets into a JSR 286 portlet by changing the one line in the portlet deployment descriptor that references the schema to the new V2.0 portlet deployment descriptor schema.

In this section, we cover various smaller additions that can make your life easier and that can support new use cases, such as the new Java 5 features leveraged in the API, the new caching features, the changes in the runtime IDs that you can access, and the tag lib additions.

**Java 5 features**

JSR 286 leverages some of the new Java 5 features:

- Using Generics
- Introducing a new enum class for all user profile attributes

GenericPortlet also lets you use the following annotations to make dispatching life-cycle calls to specific handler code easier:

- ProcessAction. This annotation allows you to annotate a method that should process a specific action. To allow dispatching, the Action URL needs to include the parameter javax.portlet.action.
- ProcessEvent. This annotation allows you to annotate methods that should process specific events.
- RenderMode. This annotation allows you to annotate methods that should render a specific portlet mode.

Listing 8 shows how you use the ProcessAction annotation.
Listing 8. Leveraging the ProcessAction annotation

```java
// generating the URL
PortletURL url = response.createActionURL();
url.setParameter(ActionRequest.ACTION_NAME, "actionA");
url.write(output);

// method for handling the action
@ProcessAction(name="actionA")
void processMyActionA(ActionRequest req, ActionResponse resp)
throws PortletException, java.io.IOException { ...
}
```

Because some portal implementations still depend on Java 1.4, only features that can easily be removed for Java 1.4-based platforms have been added to JSR 286. For Java 1.4-based platforms, a JAR file of the JSP 286 APIs compiled with Java 1.4 compliance settings in which the preceding features have been removed.

New caching features

JSR 286 adds new caching features that help making portlets scale better:

- Public caching scope, which marks cache entries shared across users
- Support for multiple cached views per portlet
- Validation-based caching for validating expired cache entries

In JSR 168, all cache entries were private for each user. If you have portlets that are not customizable and don't display different content for different users, such as a news-of-the-day portlet, then you get a cache entry for each user, even if all entries are the same. Now, in JSR 286, you can tell the portlet container that the cache entry can be shared using the cache-scope element in the deployment descriptor. Thus, you can dramatically reduce the memory footprint for these portlets, as the code in listing 9 shows.

Listing 9. Public caching scope example

```xml
<portlet>
  ...
  <expiration-cache>60</expiration-cache>
  <cache-scope>public</cache-scope>
  ...
</portlet>
```

A portlet can also set the cache scope programmatically using a response property or the new CacheControl interface. We recommend that you do not change the cache scope programmatically, though, as it may be difficult for portlet containers to ensure that cache entries are invalidated correctly if your portlet mixes responses with cache scope public and private for the same user.

The JSR 168 specification demanded that portlet containers must invalidate the cache for each user interaction with a portlet (render or action URL), so that there could be only a single cache entry per portlet window and user. With the introduction of public render parameters and shared
caching, the specification has been enhanced to allow the container to cache multiple portlet views based on render parameters. This enhancement means that you can use render parameters so that users can navigate between different views in a shared cacheable portlet while the output still remains cached. Only the action and event processing calls, which can cause side-effects that are not known to the container, now require a cache invalidation.

Validation-based caching is useful for situations in which you do not want to recompute the markup of a portlet often because it is an expensive operation. Ideally, set the expiration time to a high value so that the markup is cached for a long time. On the other hand, you may want to respond quickly to changes in your back-end system and provide an updated view of the portlet. Validation-based caching solves this dilemma: It allows you to define a short expiration time so that your portlet gets called often and can check for changes in the back-end system. If nothing has changed in the back-end system, you validate that the expired cache content is still valid to use by setting the CacheControl.setUseCachedContent(true) and setting a new expiration time. If something has changed in the back-end system, then you produce new markup and set the setting to false.

How do you know which back-end state the currently cached markup maps to? You can set a specific validation token, called ETag after the same tag in the HTTP specification, on the response. The portlet container can then provide you with this ETag in the request of the next render or serveResource call, indicating that it still has the cached content available, which can be revalidated.

An example of using validation-based caching is shown in listing 10.

**Listing 10. Validation-based caching example**

```java
protected void doView(RenderRequest request, RenderResponse response)
    throws PortletException, java.io.IOException
{
    if ( request.getETag() != null ) { // validation request
        if ( markupIsStillValid(request) { // markup is still valid
            response.getCacheControl().setExpirationTime(30);
            response.getCacheControl().setUseCachedContent(true);
            // no need to write any output
            return;
        }
    }
    // create new content with new validation tag
    response.getCacheControl().setETag(computeETag(request));
    response.getCacheControl().setExpirationTime(60);
    PortletRequestDispatcher rd = getPortletContext().getPortletRequestDispatcher("jsp/view.jsp");
    rd.include(request, response);
}

private boolean markupIsStillValid(PortletRequest request) {
    // check if backend state still matches validation token
    return computeETag(request).equals(request.getETag());
}

private String computeETag(PortletRequest request) {
    // return some backend state indicator like a last update timestamp
}
```

What's new in the Java Portlet Specification V2.0 (JSR 286)
Using validation-based caching is beneficial only if the operations for creating the markup are expensive compared to the operations for checking the back-end state. If, for example, the request to the back-end server takes up 90 percent of the time to render the portlet, it does not make sense to use validation-based caching to save only the remaining 10 percent rendering time.

**Runtime IDs**

Runtime IDs can be used to scope data that your portlet handles, so that you can avoid collisions between multiple occurrences of a portlet within the same portal, and maybe even on the same page. Most portlet API objects, for example, portlet session or portlet preferences, do implicit namespacing, but you need to do explicit namespacing when you access shared data stores or want to assign unique IDs to output elements. This practice is common for Ajax applications, which can particularly benefit from the two enhancements that version 2.0 provides in this area.

First, JSR 286 extended the lifetime of the namespace runtime ID, using the response.getNamespace method, from being valid for only one request to now being stable for the lifetime of the portlet window. This extension now allows you to use the namespace to also create namespace form IDs if, for example, your portlet is intended to be aggregated in a forms-based portal server. You can also leverage the namespace for JavaScript function calls that are embedded in Ajax responses, and you can reuse namespaced JavaScript functions provided by a previous render call.

Second, JSR 286 introduced a new API call that allows you to get to a unique ID for the portlet window using the request.getWindowID method. This call now allows you to use this ID as key for data that you want to create namespace per portlet window; for example, the portlet wants to cache data that it received from a back-end system per portlet window. Note, however, that no lifecycle calls are defined around the portlet window, so if you use the portlet window ID for specifying namespace entries in a persistent data store, you also need to clean up these entries yourself.

**Taglib additions**

The JSR 286 tag library has its own namespace so that new additions do not interfere with portlets using the old JSR 168 tag library. You now need to include the new tag library with: `<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>`

The define objects tag is now enhanced and gives you the following additional variables beyond the request, response, portletConfig from the V1.0:

- `portletSession`. For access to the portlet-scoped session.
- `portletSessionScope`. For access to the portlet-scoped session attribute key/values map.
- `portletPreferences`. For access to the portlet preferences.
- `portletPreferencesValues`. For access to the portlet preferences key/values map.

Each of the URL generation tags has the following additional attributes:

- `copyCurrentRenderParameters`. For copying the current private render parameters to the URL.
- **escapeXml.** For turning the default XML escaping of the URL. As mentioned previously, in JSR 168 it was not defined if a URL is XML-escaped or not. JSR 286 now has defined the same behavior as the Java Standard Tag Library: by default all URLs are escaped, but you can turn it off using the escapeXml attribute.

For action URLs, there is also the additional name attribute that allows you to set the javax.portlet.action parameter that is evaluated by the GenericPortlet for dispatching to the ProcessAction annotated methods.

Other additions made in the JSR 286 specification include these:

- Adding a new resourceURL tag for generating resource URLs
- Adding the new propertyTag that can be used inside portlet URL tags for attaching properties to URLs

**Conclusion**

As you have seen, the second version of the Java Portlet Specification added a lot of new content and abilities. The specification and the APIs more than doubled compared to version 1.0. Now the Java Portlet Specification has grown up, and it allows you to implement most use cases without the need to have vendor extensions. Note that some of the features described in this article are optional, so they may not be supported on all JSR 286-compliant platforms, but the standard ensures that those platforms that provide extended capabilities do so in a consistent and well-defined manner.

The portlet programming model now provides you with events and public render parameters so that you can build larger composite applications out of your portlets and reuse your portlets in different scenarios. Finally, you get better support for Ajax-based use cases with JSR 286 as you are now able to serve resources directly through the portlet.
Related topics

- Refer to the JSP 286 specification.
- Refer to the JSP 186 specification.
- Consult the IBM WebSphere Portal page.

© Copyright IBM Corporation 2008
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)