Retrieving URL parameters from JSR 168 portlets using WebSphere services

Ron Lynn

April 11, 2008
(First published July 25, 2007)

It all started as an everyday event. I had just jumped into a new project to refactor some legacy portlet code and to convert it to JSR 168. I wrote the code years ago and had left it in my successor’s very capable hands.

As a Software Engineer, confronting daily challenges and identifying real problems are my bread and butter; here, I was staring at a road block. The JSR 168 standard does not define a way for a portlet to retrieve plain old URL parameters. The refactoring requirement called for converting the legacy IBM portlet API portlets to JSR 168. The swizzle was that the portlets needed access to URL parameters so that Web applications outside of WebSphere Portal could pass parameters to the portlets.

This article is the result of meeting this trial and overcoming it. It describes how I used the services in IBM® WebSphere® Portal V6 and IBM WebSphere Application Server V6 to enable JSR 168 portlets to read URL parameters. You also see references that address the same or similar situation, but solve it with a different approach.

You should have a good understanding of Java programming; understanding servlet programming and JSR 168 portlet programming would be helpful. Above all, you just need an open mind and the willingness to learn.

Introduction

I recently ran into an interesting problem. I wanted to convert some portlets from the legacy IBM Portlet API to the standard Portlet API 1.0 (JSR 168). These portlets need to be able to retrieve URL parameters. The legacy API was built on top of the Servlet API, and it provided a way for portlets to retrieve parameters that were not specifically intended for the portlet. The JSR 168 API only mirrors the Servlet API; it does not allow retrieval of plain old URL parameters.

In the future (according to the spec), Portlet API 2.0 (JSR 286) will allow shared parameters called public render parameters which, I hope, is just a parameter on a URL. The rationale behind not sharing render parameters in JSR 168 is that parameters are assumed to be private and sharing parameters is expensive.
Why would a portlet developer like to retrieve parameters? Portlets are nice neat little packages that can be wired to other portlets. The portal provides many interesting utilities. While this is certainly all very tidy, the rest of the world isn't. The rest of the world is mired with legacy applications and Web sites that would like to link into the middle of our portal and portlets. Then there's this sticky wicket of being human.

Here's a test for you. Which would you want to put on your business card or recite over the phone?

https://technotest.ibm.com/wps/myportal/Results?searchTerms=foo

https://technotest.ibm.com/wps/myportal/!ut/p/c1/04_SB8K8xLL9MSSzPy8x8z9CP0os3gjCz83f293QwN_FxMzA65AD0t_x5BQlwNL
E588J655dyMcYu688nNT97yI808v00qGw1/1/1/1/L81D00NTQ1fV51YVIR0JSFJBJ
U1nb0FeUtea7ZzL1CkSp3NDUa3Ne5xX3ISEvN1ByOE5GTtHMTBPRQ2MDJQ/5D
1PQVRVMjBTN19zZWFeY2huWZXJtcy9mb28!/1/#28NFOKg180D4662PH90ATU20S6

Believe it or not, these URLs point to exactly the same page with exactly the same parameter passed. Personally, I always pick #1. After all, my business cards are not printed on 4" x 6" cards, and life isn't long enough to recite #2 over the phone. (And besides even if I did recite it, whoever was listening would probably fall asleep before I finished. Ok, I see you there in the back chuckling at me. You say you can decode and encode these URLs in your head? Well good for you! I'm not sure why you're reading this article, please go back to working on Web 3.0 or finish eating your "Ones and Os" cereal.)

The bottom line is that there are at least two reasons for wanting to read parameters:

1. Connectivity with legacy applications.
2. Readable URLs.

So let's get on with it. How do I get my portlet to read those parameters without using some super secret API? Hongging Song and Richard Scott provided one approach in their article "Page-to-page communication between legacy Web pages and portlets." (See Related topics for a link.) It involves using a Servlet (JSP) to construct the big long URL and then redirect the user's request. It's a reasonable approach for a JSR 168 portlet and one that I've kept in my back pocket for some time now. If you read that article, you see that it's pretty easy to grab a parameter in the legacy IBM Portlet API. It's as simple as calling:

**Listing 1. Retrieving parameters in the legacy IBM Portlet API**

```java
PortletRequest.getParameter(<param name>);
```

You might be wondering, "Why, if it's so easy in the IBM API, would you want to convert to the JSR 168 API?" The best reason is that the IBM API in WebSphere Portal is deprecated. This doesn't mean it's going to be dropped like a hot potato any time soon. However, it does mean that we are all better off programming to the standard API. Secondly, JSR 168 is supported by other portals. Portals are now legion. If you want your whiz bang portlet to run on some other portal, then you better code to the standard. (That's the 13th portlet programming commandment.)

The general approach you see in this article is:
1. Intercept the user's request with a servlet filter.
2. In the servlet filter, grab the parameter and store it temporarily in a dynamic cache.
3. In the portlet, grab the parameter from the dynamic cache.

Figure 1. The parameter-grabbing approach

That's the "50,000 foot" block-diagram version of the approach. Let's see how to implement it.

Create and set up the cache

You start by creating the cache. Stefan Hepper and Stephan Hesmer talk about using a dynamic cache in portlets in the developerWorks article titled "Caching data in JSR 168 portlets with WebSphere Portal V5.1". (See the Related topics for a link. In the article, see the section titled Leveraging the WebSphere dynacache infrastructure.) They offer a couple of different ways to create and set up the dynacache. The dynamic cache infrastructure is pretty nifty and, if you have a few moments, I'd suggest taking a look at it for any of your caching needs.

For the task at hand, you can just use the Application Server's administration console to enable and create the cache:

1. Login to the admin console. In WebSphere Application Server v6, the URL is something like: http://<server>:9060/ibm/console. The port could be different for your environment. If you don't know the port, ask your system administrator. Or, if you have access to the command line, you can `grep` the serverindex.xml file usually found in the WebSphere install directory under:

```
AppServer/profiles/wp_profile/config/cell/<cellname>/nodes/<nodename>/
```

Look for "WC_adminhost_secure" as an endPointName or try this command (on one line):

```
grep -A 1 "WC_adminhost_secure"
<path>/AppServer/profiles/wp_profile/config/cells/<cellname>/nodes/<nodename>/serverindex.xml
```

(If anyone reading this article knows an easier way to find out which port the admin server is running on, send me email. I'll update the article and lavish kudos upon you.)
2. Now, make sure that the Dynamic Cache Service is enabled. Go to Server => Application servers, and choose WebSphere_Portal.

Figure 2. Select the WebSphere_Portal server

3. Next, select Container services => Dynamic Cache Service.

Figure 3. Select the Dynamic Cache Service

4. You should see that the Dynamic Cache Service is enabled at server startup; if not, enable it.
5. The service is enabled. So now let's create the cache. Expand Resources => Cache instances => Object cache instances.

Figure 5. Select Object cache instances

6. Click New.
7. Fill in the required fields and frob (modify) any of the other fields, if you like. See the WebSphere Application Server Information Center, listed in Related topics, for help with the various settings. That way, you can make an informed decision for your environment. Here's what I set:
   - **Name**: Parameter Cache
   - **JNDI name**: services/cache/catalog/parameters
• Checked User listener context
• Unchecked Dependency ID support

The names are arbitrary. Just make sure you set it to something fairly unique and then remember it for when you create your servlet filter and portlet code.

8. Click OK, and then save the configuration. You have a freshly minted cache to work with.

Figure 6. Your cache should show up in the list of available caches

9. Again, remember the JNDI name or write it down. You will need the JNDI name to do a JNDI lookup in your code. So, either make note of it now or leave this page up in the admin console.

There is an important issue with using the dynamic cache that should be carefully considered. Since it is a cache, it could be completely filled up and not allow one to store anything more. So make sure your cache size is large enough for your application. The range is from 100 to 200,000 cache entries. So, do a little math to figure out how many expected concurrent hits times the number of parameters you'll be caching then arbitrarily double it. Of course, if you really want to find a better number, look in the WebSphere Application Server Info Center and search for "dynamic cache size". You'll end up with a list of performance advisors, monitors, viewers, and troubleshooting tips to assist you in picking the right cache size.

Create and install the servlet filter

Now that you have a cache in which you can place the parameters, you create the servlet filter that will stash the parameters into the cache. Later, your portlet can retrieve them from the cache.

Create the servlet filter

I used IBM Rational® Software Architect to create my servlet filter; you could use it, Rational Application Developer, or another development environment. The goal is to create a JAR file with your filter class in it.

You first need to create a servlet filter with the code shown in Listing 2 (or you can get from the download), which I describe in some detail below.
### Listing 2. The start of the ServletFilter class

```java
public class ParameterFilter implements Filter {

    private DistributedMap map = null;

    public ParameterFilter() {...}
    public void destroy() {...}

    public void init(FilterConfig arg0) throws ServletException {
        try {
            InitialContext ic = new InitialContext();
            this.map = (DistributedMap) ic.lookup("services/cache/catalog/parameters");
            ic.close();
        } catch(NamingException ne) {
            System.out.println("NamingException error");
            System.out.println(ne.getMessage());
        }
    }

    // ... }
```

The first thing task in the `init` method is to use the JNDI name to look up the cache you created earlier. Assuming you remembered what you used for step 7 above, you'll want to use it here in the `InitialContext.lookup` method.

The real juicy part of the ServletFilter is the `doFilter` method.

### Listing 3. The rest of the ServletFilter class

```java
public class ParameterFilter implements Filter {

    public void doFilter(javax.servlet.ServletRequest request,
                         javax.servlet.ServletResponse response,
                         javax.servlet.FilterChain chain)
                        throws java.io.IOException, javax.servlet.ServletException{

        HttpServletRequest httpRequest = (HttpServletRequest)request;
        HttpSession session = httpRequest.getSession();
        Enumeration parms = httpRequest.getParameterNames();
        while(parms.hasMoreElements()) {
            StringBuffer key = new StringBuffer((String)parms.nextElement());
            String value = httpRequest.getParameter(key.toString());
            //mangle the key so that it's unique for this session
            key.append(".");
            key.append(session.getId());
            map.put(key.toString(), value);
            System.out.println("Parameter Filter: "+key++ " -- "+value);
        }

        // forward the wrapped request to the next filter in the chain
        if (chain != null){
            chain.doFilter(request, response);
        }
    }
```

Get all the parameters from the `HttpSession` and put each one into the `DistributedMap`. Then, mangle the parameter names so that they are unique for each session; otherwise, you could end up overwriting another user's data in the map. This code moves all the parameters. If you would
prefer to only target certain parameters, you can easily make the changes; that task is left as an optional exercise for the reader.

I'll reemphasize that this is a cache. Because we are caching all the parameters that are passed through the URL, the cache could fill up. So, while I leave the exercise of targeting parameters to the reader, you should implement such a mechanism and also enhance what you learn in this article.

For illustrative purposes, the example code includes a `System.out.println` statement so that you can look in the portal `SystemOut.log` file to see your filter in action. You might notice that this code does not handle multiple parameters of the same name. That is, if you had a URL like `http://<server>/wps/portal?parm=foo&parm=bar` you'd end up overwriting `parm` in the dynamic cache. You could overcome this restriction fairly easily in several different ways. One way would be to put parameters in the dynamic cache assuming there are multiples; that is, use dynamic cache names that are indexed. For example: `parm.0.sessionID=foo, parm.1.sessionID=bar`, and so on. Then, you would need to pull them out of the cache in a similar fashion. Another way would be to comma-separate them in the cache. For example: `parm.sessionID=foo,bar`. There are many ways to marshal the data; it is just a matter of choosing what is right for your application.

When you are finished coding (or in this case copying the code into) your ServletFilter class, build this code into a JAR file.

**Install the servlet filter**

Now, you are ready to install the filter.

1. Place the JAR file you created above in the `shared/app` directory under the portal install directory:
   ```bash
   ${portalserver}/shared/app
   ```

   On the Linux® server, I used:
   ```bash
   /opt/ibm/WebSphere/PortalServer/shared/app
   ```

2. Configure the filter in WebSphere Portal's `web.xml` file:
   ```bash
   ${appserver}/profiles/wp_profile/config/cells/${server}/applications/wps.ear/
   deployments/wps/wps.war/WEB-INF
   ```

   On Linux, I used:
   ```bash
   /opt/ibm/WebSphere/AppServer/profiles/wp_profile/config/cells/icat28/
   applications/wps.ear/deployments/wps/wps.war/WEB-INF
   ```

3. Add the following code to the `web.xml` file:
This code does two things. First, it defines the filter in the Application Server. Second it tells the Application Server that before you pass the request on to WebSphere Portal, call the Parameter filter class. Because the URL filter mapping filters both "/portal/" and "/myportal/", the filter will get called whether or not you are logged on. (The "/myportal/" is the logged-in portal url.)

4. Restart the portal server.
5. After WebSphere Portal is up, you can test the filter. Access the portal by typing something like: http://<servername>/wps/portal?parm=quux
6. Then, look in the SystemOut.log file in the portal server's log directory. You should see a line that looks similar to the following, (assuming, of course, that you put a System.out.println statement in your ServletFilter code):

```
[7/7/07 16:30:24:177 EDT] 0000007c SystemOutO Parameter Filter: parm.dl9vOW6IhS68o0GqTjZS7QQ – quux
```

It's broken, how do I fix it?

This didn't work the first time for me either. If it worked for you, YaY! Then I guess this article did its job. For the rest of us, here are the problems I ran into and their fixes.

1. Error 500, some sort of problem with the filter. For some reason your filter blew up. It happens to all of us. What happened in my case first time I ran it, was that I changed from using a String for my dynamic cache key to a StringBuffer, but I forgot to change the map.put code. Inspect your code, it shouldn't be terribly long and whatever is broken should be easy to spot. If this doesn't work, capture all exceptions and output them to a log file.
2. My filter never seems to get executed. In my case, I defined the filter in the wrong place so it never got executed. Go back and look at which web.xml file you modified. It's the one in the profiles/wp_profile/config directory, NOT the one in the profiles/wp_profile/installedApps directory! I know seems like a silly mistake, but it happens.

Create the portlet

You can use any old portlet for this part. What you will do is pretty non-invasive. You add a few lines to the init method and then add a getParameter method.

1. Add the following lines of code to your init method:

```xml
<filter>
  <filter-name>Parameter Filter</filter-name>
  <filter-class>com.ibm.catalog.filters.ParameterFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name> Parameter Filter</filter-name>
  <url-pattern>/myportal/*</url-pattern>
</filter-mapping>
<filter-mapping>
  <filter-name> Parameter Filter</filter-name>
  <url-pattern>/portal/*</url-pattern>
</filter-mapping>
```
public abstract class ParameterPortlet extends GenericPortlet {

    private DistributedMap parameterMap = null;

    public void init() throws PortletException{
        super.init();
        try {
            InitialContext ic = new InitialContext();
            parameterMap = (DistributedMap)
                ic.lookup("services/cache/catalog/parameters");
            ic.close();
        } catch (NamingException e) {
            System.out.println("NamingException in the portlet");
        }
    }
}

This code simply looks up the same cache that you defined at the beginning of this article and
assigns the DistributedMap to a variable.

2. Next, add the getParameter method.

public abstract class ParameterPortlet extends GenericPortlet {

    protected String getParameter(RenderRequest request, String name){
        StringBuffer key = new StringBuffer(name);
        key.append(".");
        key.append(request.getPortletSession().getId());
        String parm = (String)parameterMap.get(key.toString());
        System.out.println("Portlet parm: "+key+ " -- "+parm);
        parameterMap.remove(key.toString());
        return parm;
    }

}

This code simply grabs the parameter out of the DistributedMap and returns it. It removes
the parameter from the map once this method is called; you may or may not want to do that.
It's just depends on what you plan to do with the parameters. In this case, I moved them to a
request attribute so I'd have further access to them in the rest of the portlet. This is easy to do
by adding these few lines of code to the getParameter method:

    String parm = (String)parameterMap.get(key.toString());
    if(null != parm){
        request.setAttribute(name, parm);
    }

That's all there is to the portlet, other than using the parameters. You could implement other
methods in addition to getParameter. Perhaps you need to mirror all the getParameter
type methods. Maybe you want this to be more transparent and don't care where your
parameters come from. You could use the getParameter method above as wrapper to the
request.getParameter method to look in the request and then in the dynamic cache for
parameters.

Here's one example of a super-duper getParameter method:
protected String getParameter(RenderRequest request, String name) {
    StringBuffer key = new StringBuffer(name);
    key.append(".");
    key.append(request.getPortletSession().getId());
    String parm = (String)parameterMap.get(key.toString());
    if(null == parm) {
        parm = request.getParameter(name);
    }
    if(null == parm) {
        parm = (String)request.getAttribute(name);
    }
    if(null != parm) {
        request.setAttribute(name, parm);
    }
    System.out.println("Portlet parm: "+key+" -- "+parm);
    parameterMap.remove(key.toString());
    return parm;
}

You might want to change the order of precedence; for example, a portlet specific parameter might be more important than a URL parameter.

**Call a page with parameters**

So far, your portlets can read URL parameters, but that doesn't do you much good unless you can get to the page on which the portlet resides. Those of you who know about URL Mappings and Custom Unique Names, can skip to the Conclusion, if you so desire.

The problem with what we have done so far is that unless your portlet is on the default portal page, you still have a very long URL on which to add parameters. You can use URL Mapping to get a URL that points directly to your page.

Here's what you need to do:

1. Login to WebSphere Portal as an administrator.
2. Go to the Administration page.
3. Select **Portal Settings** => **Custom Unique Names** => **Pages**.
4. Search for the page that contains your portlet.
5. Click the pencil icon and give the page a unique name.

**Figure 8. Set a unique name for your page**

6. Now that your page has a unique name, you can set a URL mapping for that page. In my case, I wanted a URL that points directly to my Results page, and I wanted something easy like: http://<servername>/wps/portal/Results
You can specify what you want by selecting **Portal Settings => URL Mapping**, and clicking the **New Context** button.
7. Choose a label; in this case, I chose **Results**, as you can see in the URL above.
8. Now, click the Edit mapping button (-pencil) in the same row as the Context label you chose.
9. Find and select the page with the unique name you selected in step 5. You don't really need a custom Unique Name, but it sure is easier that remembering that your page's Unique Identifier is "6_28NF0KG100D4602PH90ATU2083".

10. Save everything and you are done.

Your page should now be available by just tacking on the Context label you chose. If you add parameters onto the end of that URL, your portlet will pick them up. For example you could pass a query parameter to the "Results" page above to show the results of that query:

http://<servername>/wps/portal/Results?query=search+me

Conclusion

Well folks, that's what happened to me and how I approached a solution. I hope this article has given one way of overcoming the lack of URL parameter support in JSR 168. You can see that it's quite easy to configure and use a dynamic cache from a portlet running in WebSphere Portal v6. It's also easy to implement and install a servlet filter that executes before WebSphere Portal gets called. I am already scheming and planning how I might use servlet filters for other purposes. While reflecting on this article, I wonder if this approach could be used outside of WebSphere Portal. One would only have to replace the dynamic cache with a generic caching mechanism or shared memory mechanism for temporarily sharing the parameters. If you happen to do this, drop me a line and let me know. I'd be interested to hear about your experiences. Thank you for spending your time reading this article. I hope it has helped you in some small way.

Acknowledgements

I'd like to thank the following people (in no specific order) for reviewing this article:

- Stefan Hepper
- Joey Bernal
- Karl Bishop
- Doug Phillips
- Mike Dockter
- Lan Chen
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code samples</td>
<td>parameter-samples.zip</td>
<td>6 KB</td>
</tr>
</tbody>
</table>
Related topics

- Popular trials and downloads from developerWorks
- Developing a multi-locale site using WebSphere Portal V5.1.0.1. See the section named "Developing and installing the servlet filter”.
- Caching data in JSR 168 portlets with WebSphere Portal V5.1
- Page-to-page communication between legacy Web pages and portlets
- WebSphere Application Server documentation in IBM Knowledge Center
- Writing Servlet Filters

© Copyright IBM Corporation 2007, 2008
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)