The Top 10 Things You Need to Know About Portlets

Martin C. Stoufer
MCStoufer@lbl.gov
Lawrence Berkeley National Lab

Introduction:

As DAQ software grows and matures, so do the number of their published Mbeans. The DAQ Portlet effort has kept up with this supply and has working sets of portlets for both DAQ Dispatch and DAQ Config/Control/Monitor.

This talk is intended to be an introductory guide and review of the important things you need to know in order to make portlets and to make exiting portlets better.

Target audience: All IceCube software developers who have Mbeans in their code and are looking to create usable portlets and/or improve on the ones they already have.

This is a verbal presentation of the *IceCube Portlet Developers Pocketbook.* [URL]

Definitions:

Throughout this guide, you will see mention of the following ideas and names when used to describe components. A brief description here will make for easier understanding later on.

IceAxe ≈ Portal

IceAxe is the name of the Portal that is used by the Icecube software project. It is the all encompassing web based framework that the Portlets are displayed through. By itself, the Portal is little more than a login context with empty pages behind it.

JSP ≈ Portlet

Even though these are not technically the same, The Icecube Portal development team has enforced a one-to-one mapping of a JSP (Java Server Page) to rendered portlet. Some deviation from this is afforded developers when they include 'snippets' of portlet code into other portlets.

Requirements:

To get your portlet project off the ground, you will need a functioning and current BFD Workspace and Mercury System installed. Refer to the **References** Section at the end for pointer URLs.

I. Content

1.Creating the project

Since the portlet project is Java based, DAQ supplies an Ant target to create and build them as well. Similar to the 'createProject' and 'createPackage' targets, these portlet specific targets will create all the files and directory structure you need to get going.

```
> ant –DPROJECT=foo createPortletProject
> ant –DPROJECT=foo –DPACKAGE=bar createPortletPackage
```

Once you have run both of these commands, you will be left with the proper build and portlet config files as well as the stub directory structure for your JSPs and corresponding Java files. Any sub structure underneath is up to you to define and implement. The RPM spec file is pretty empty but at least you have the format and targets to fill in. Refer to the spec file in *daq-display* to see how things are done.

When it comes time to start creating you portlet pages, there is a lot of preamble and DAQ-specific wrapper code that goes into each JSP. So much that an ant target is provided to allow easy creation of a JSP ready for your code.

```
> ant –DPROJECT=foo –DPACKAGE=bar –DCLASS=baz createPortletClass
> ant –DPROJECT=foo –DPACKAGE=bar –DCLASS=baz createPortletFormClass
```

The first will create the boilerplate required to generate your basic JSP; one that doesn't accept input and simply displays information from one or more Mbeans. The later, however, will also create the corresponding java template. You will need to fill in the handling code to suit your needs. Later on you will see some examples.

After filling things in enough, you can use your favorite VCS to store the initial project.

2. Project layout

Lets take a closer look at the files that the ant targets before just created.

a. portlet.xml
   Define the portlets to make available when 'deployed'. This allows you to have portlets in your projects that don't show up in a deployment.
b. project.xml
   This specialized version includes the new entries in the <tools> element to enable portlet generation. No edits are needed here.
c. RPM spec file
   When filled in properly, this will allow the portlet project to be compiled into an RPM for installation in a Portal environment. You should refere to an existing portlet project that has this all filled in. The required edits are few and obvious.
d. resources/jsp-app/classes/icecube/display/*bar*/*bar*.properties
   Create bundle key pair associations. Give example of the long way and how shorthand is so much better. This directory and properties file are created from the 'createPortletProject'.
   When you start JBoss with only DAQ related Mbeans deployed into it, you can visit the *JMX Console* and look at all the bundle resources currently deployed. From here you can create a bundle-key pair for entry into this file. More info is given later on showing how this all works. You can also define a key that will purposefully resolve to multiple Mbeans.
e. resources/jsp-app/*bar*/
   JSPs go under this directory tree. You may create sub directories under here, but the mapping must be followed in the src/icecube/display/*bar*/ directory. The proper dir tree must also be followed when creating entries in the portlet.xml file.
f. resources/jsp-app/images/
   All images used by all jsps go here. Supported formats are JPG,GIF, and PNG.
g. src/icecube/display/*bar*/
   Java action code goes under this tree in same layout as in resources/jsp-app/*bar*/

Example portlet.xml entries:

When the project is initially created, you will have an portlet.xml file without any entries. Here are two examples. One that just displays info and the latter for when your portlet uses one or more <form> elements in it:

```
<portlet>
    <description>EventBuilder Monitor Summary</description>
    <portlet-name>EventBuilderMonitorSummary</portlet-name>
    <display-name>EventBuilder Monitor Summary</display-name>

    <portlet-class>gov.lbl.henpcg.stormlet.JspRenderer</portlet-class>

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

    <portlet-info>
      <title>EventBuilder Monitor Summary</title>
    </portlet-info>

    <portlet-preferences>
      <preference>
        <name>base-dir</name>
        <value>/WEB-INF/bar/EventBuilder/monitor</value>
      </preference>
      <preference>
        <name>view-jsp</name>
        <value>EventBuilderMonitorSummary.jsp</value>
      </preference>
    </portlet-preferences>
  </portlet>


<portlet>
    <description xml:lang="EN">DAQControl System Configuration Create</description>
    <portlet-name>DAQControlSystemConfigureCreate</portlet-name>
    <display-name>DAQControl System Configuration Create</display-name>

    <portlet-class>gov.lbl.henpcg.stormlet.Jsp**Form**Renderer</portlet-class>

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

    <portlet-info>
      <title>DAQ Control System Configuration Create</title>
    </portlet-info>

    <portlet-preferences>
      <preference>
        <name>base-dir</name>
        <value>/WEB-INF/bar/DAQControl/systemConfigure</value>
      </preference>
      <preference>
        <name>view-jsp</name>
        <value>DAQControl_SystemConfigureCreate.jsp</value>
      </preference>
      <preference>
        <name>form-handler</name>
<value>icecube.display.DAQ.DAQControl.systemConfigure.DAQControlSystemConfigureCreate
</value>
      </preference>
    </portlet-preferences>
  </portlet>
```

3. Resolving Mbean(s)

In order to find out what beans are deployed and waiting for you to build portlets from, you need to have JBoss running and have DAQ-PROD deployed therein. Instructions for doing so are referenced at the end. Once you have everything running, you can point your web browser at the local address:

http://localhost:8080/jmx-console

This will bring up the JMX Console page. Not very interesting, but you will see an endless list of mbean resource strings that this node knows about. By clicking on one, you will see the Mbean display page that will show all the attributes and operations the mbean provides. You are advised to become comfortable with this console, as it will be an integral part of development and debugging of your portlets

By using the bundle key pairs, we can then leverage some portlet tags developed by DAQ to make the resolving of Mbeans across an SP*S cluster easier.

Lets start by looking at a few example entries from a portlet.xml file that are used here:

```
daq.eventbuilder.attributeConfiguration=
iceCube.daq:acme-aspect=configure,
id=0,name=eventBuilderAttributes,type=eventBuilder

daq.stringprocessor.attributeConfigurations=
iceCube.daq:acmeaspect=configure,
name=stringProcessorAttributes,type=stringProcessor,*
```

Due to layout constraints, each of these entries should always be on one line.

On the left is the key that you think up and on the right is the resource bundle that is listed on the JMX Console. The dot notation for the key is purely artifical and only servers as some sort of hierarchy for humans to understand.

One major difference between these two bundle key pairs are the trailing “,*” and the missing “id=0” keys for the stringprocessor resource. Whenever you leave out one or more items from the resource, you must use the “,*” wildcard at the end. This will cause JBoss to resolve as many Mbeans that match this sub-query as possible.

Now that we have the bundle keys ready, we can now put code into JSPs that will give us access to one or more beans. The following tags do just that:

```
<jmxext:resolveMBean
           bundle="icecube.display.bar.bar"
           beanKey="daq.eventbuilder.attributeConfiguration"
           mbean="configMbean"
           server="mbeanConfigServer"/>

<jmxext:resolveMBeans
      bundle="icecube.display.bar.bar"
      beanKey="daq.stringprocessor.attributeConfigurations"
      comparator="icecube.display.DAQ.IdComparator"
      mbeans="mbeans"
      servers="mbeanServers"/>
```

The first tag is pretty straight forward. Let's break it down argument by argument:

i. bundle
   This is the java classpath that points to your "*bar*.properties" file starting from the resource/jsp-app/classes directory. Consider this to be a static string for all your resolveMBean(s) tags. Unless you have multiple properties files in the same project. Doing so will work, but maintenance and further development by others may prove too confusing.
ii. beanKey
   This is the literal key from bundle-key pair found in the bundle above and should be unique to each resolveMBean(s) call on a page. Both the bundle and the beanKey are crucial in locating the Mbeans you want to work with.
iii. mbean
   Whenthe backend code resolves a Mbean that matches the bundle and beankey, it stores a reference to the mbean into the argument. In this case, 'configMbean' is now a variable name you may call later on in the JSP. The argument could be any string you want, but must be unique as you may be resolving multiple single mbeans in a page.
iv. server
   In the distributed environment of DAQ and JBoss, there are multiple servers running all providing a subset of Mbeans you may want to use. When an mbean is resolved, we also need to keep track of what server has it. This way when we go to interact with the mbean, we can explicitly tell which server to contact. This saves a lot of time and resources in the long run. Like mbean argument, the reference to the server is stored in the 'mbeanConfigServer'.

Notice that the second tag is "pluralized". This will cause all the code behind the scenes to resolve *ALL* mbeans that match the bundle and beanKey combination. Similar to its single mbean resolve call, the mbeans now stores a list of all mbeans found and a hash of servers in mbeanServers keyed by the items in the mbeans list. Therefore, this multiple mbean query would resolve to the following example resource keys:

```
daq.stringprocessor.attributeConfigurations=iceCube.daq:acmeaspect=configure,id=0,
name=stringProcessorAttributes,type=stringProcessor

daq.stringprocessor.attributeConfigurations=iceCube.daq:acmeaspect=configure,id=1,
name=stringProcessorAttributes,type=stringProcessor

daq.stringprocessor.attributeConfigurations=iceCube.daq:acmeaspect=configure,id=2,
name=stringProcessorAttributes,type=stringProcessor
```

We will discuss later on how to iterate over this list and hash.

The optional comparator argument is used when you want to order the list of returned mbeans with respect to some aspect of the bundle key. This should only be used when you are working with multiple mbeans and is discussed later.

4. Getting attributes

So now we have access to an mbean and its server, in order to get access to one of its attributes, we can use the 'getAttribute' tag. The format is easy to use.

```
<jmx:getAttribute mbean="${configMbean}"
                  server="${mbeanConfigServer}"
                  attribute="ConfigurationId"
                  id="configId"/>
```

Notice how the mbean and server arguments are now used as inputs. The 'attribute' is an attribute in the Mbean and the 'id' argument holds the name of the variable which holds the value of the attribute. It is up the developer to understand what type of value it is and code appropriately.

If you are starting from square one, you will need to click on a few mbean resource listings on the JMX Console to see what attributes and operations a specific Mbean has.

To iterate over the list and hash of mbeans and servers, the 'forEach' tag does this quite easily.

```
<c:forEach items="${mbeans}" var="mbean">
    <jmx:getAttribute mbean="${mbean}"
                      server="${mbeanServers[mbean]}"
                      attribute="ComponentId"
                      id="spid"/>
    <!—Do something with the ${spid} variable here --!>
</c:forEach>
```

Again, the 'id' argument stores the attribute into 'spid'. Inside the 'forEach' block, you may do anything you want with this variable.

5. Invoking Operations

Simply extracting attributes from an Mbean is too restrictive when it comes to developing interactive portlets. We can also invoke operations on the Mbean that will change the internal state of the bean and/or return information to the portlet. Again, by viewing the mbean through the JMX Console, you can see the operations supported. Some take arguments, some do not. All are designed to return some value to inform the calling context (in our case the portlet) of the operation status. In most cases the return value will be something to be used in the portlet.

The single mbean invoke should look quite familiar by now.

```
<jmx:invoke mbean="${configMbean}"
            server="${configMbeanServer}"
            method="fetchConfigurationIdList"
            id="varConfigList"/>
```

When you want to invoke the same operation across a set of all Mbeans (and you know that each mbean provides the operation), you can use the same 'forEach' tag to iterate over each mbean.

```
<c:forEach items="${mbeans}" var="mbean">
   <jmx:invoke mbean="${mbean}"
               server="${attributeConfigMbeanServers[mbean]}"
               method="fetchInitialStateTransition"
               id="transition">
     <jmx:setParameter type="int">
       ${cfgid}
     </jmx:setParameter>
   </jmx:invoke>
</c:forEach>
```

We also show here the format when the operation requires arguments. We need to declare the type of the required parameter . The backend code does the casting for you if possible. There is a serious restriction here in that the order of parameters you pass in **must** be in the same order as they are listed in the Mbean display from the JMX Console.

6. Sorting results

Since Mbean discovery by the Jboss servers occurs in an indeterminate order (wrt when things are deployed and announced), it is impossible to enforce any type of order to things when mbeans are resolved in a JSP. DAQ has added some backing code that allows for a resulting list (from an invoke call) to be put into some type of order. The motivation for this feature is when you want to iterate over a set of Mbeans in some logical and repeatable manner.

Recall the comparator argument in step 3. This points to a user defined class located in

/src/icecube/display/*bar*

named IdComparator.java

The guts are as such:

```
public class IdComparator
       implements Comparator
{
    public IdComparator() {}

    public final int compare(Object o1, Object o2)
    {
        ObjectName obj1 = (ObjectName) o1;
        ObjectName obj2 = (ObjectName) o2;

        int id1 = Integer.parseInt(obj1.getKeyProperty("id"));
        int id2 = Integer.parseInt(obj2.getKeyProperty("id"));

        if ( id1 > id2 ) {
            return 1;
        } else if ( id1 == id2 ) {
            return 0;
        } else {
            return -1;
        }
    }

    public final boolean equals(Object o1, Object o2)
    {
        ObjectName obj1 = (ObjectName) o1;
        ObjectName obj2 = (ObjectName) o2;

        int id1 = Integer.parseInt(obj1.getKeyProperty("id"));
        int id2 = Integer.parseInt(obj2.getKeyProperty("id"));

        return id1 == id2;
    }
}
```

Therefore the resolveMBeans call in step 3 will always return a list of mbeans order wrt their ‘id’ field. Since the ‘id’ field is really the distinguishing key of a bundle-key pair, re-use of this one class will get you quite far. If you want to compare bundle keys against some other key, you need to implement the ‘compare’ and ‘equals’ class appropriately.

This is not needed when you are doing a ‘getAttribute’ or ‘invoke’ against a specific Mbean. The mbean developer should of coded things as to return a sorted list for you.

7. Forms and Action Java code

In the use case where you want to update information in a Mbean, the JSP needs to include a <form> context in which to display/accept new information to send to the Mbean via an invoke call. As discussed in step 2, the java code should go in the analogue location [Show tree again] for its calling JSP file.

In the DAQServers.jsp file:

```
<form action="<portlet:actionURL/>" method="post">
  <button type="SUBMIT"
          name="resetMD"
          value="Pressed"
          class="portlet-form-button">
    Reset MultiCast Detector
  </button><br>
</form>
```

In the corresponding java class:

```
public class DAQServers
        implements JspFormHandler {

    public DAQServers() {
    }

 public void handleForm(FormParameters  request)
            throws JmxExtException,
            ReflectionException,
            InstanceNotFoundException,
            MBeanException,
            AttributeNotFoundException,
            InvalidAttributeValueException {
    if (null != request.getParameter("resetMD")) {
        //Insert your code here
    }
}
```

What goes into the java file is really dependent on what needs to be done and processed. This topic is discussed more thoroughly in the *IceCube Developers Guide*.

8. Session variables:

As your portlets become more complex and exercise multiple Mbeans, it will become evident that you will need some manner in which to share bits of information across portlets. This is where the session attribute comes into play.

This is the boilerplate JSP to add to your portlet page:

```
<%@ taglib prefix="renderer"
uri="http://henpcg.lbl.gov/taglibs/JspRenderer" %>
<renderer:actionException var="actionException"/>
...

<renderer:applicationSessionAttr
              name="daq.stringproc.selectedStringProcId"
              var="id"/>
...
<form action="<portlet:actionURL/>" method="post">
             <select name="selectStringProcId" size='1'>
                 <c:forEach var="mbean" items="${mbeans}">
                      <jmx:getAttribute mbean="${mbean}"
                                        server="${mbeanServers[mbean]}"
                                        attribute="ComponentId"
                                        id="spid"/>
                             <option value="${spid}">${spid}</option>
                 </c:forEach>
             </select>
             <button type="SUBMIT"
                     name="submitStringProcId"
                     value="Pressed"
                     class="portlet-form-button">
                Select StringProc
             </button>
</form>
```

In the corresponding java file, more boilerplate:

```
public void handleForm(FormParameters request)
    {
        if (null != request.getParameter("submitStringProcId")) {
              request.addApplicationSessionAttr(
                 "daq.stringproc.selectedStringProcId",
                  request.getParameter("selectStringProcId")
                  );
         }
    }
```

II. Presentation

1. Images

Text and tables sometimes just can't get across what is needed. Adding images helps with the visual cues and helps improve the overall look and feel of the portlet. It makes it more inviting and intuitive to users.

All images go in the *images* directory as mentioned in step 2. To reference the image in a jsp:

```
<c:set var="image" value="/images/red_x.gif"/>
<img src='<c:out
value="<%=renderRequest.getContextPath()%>"/>${image}'>
```

This is nasty to look at, but this boilerplate works and will work every place.

2. Deploying to the Portlet Harness

Now that you have a working portlet or a partial one in need of some debugging, we can now deploy the portlet into the Portlet Harness. The harness is a purely DAQ invention and designed to display and operator portlets in the smallest and simplest environment possible. Only one portlet can be displayed at a time, but all the form, session attribute, and images will work the same as if it was deployed into IceAxe.

With a running jboss locally, you can deploy your portlet project.

```
> ant deploy.plt-app
```

So where is this Portlet Harness exactly? Due to its small size and simplicity, it is actually built into your portlet project before it is deployed into JBoss. In order to view the harness, just point your web browser at the local address:

http://localhost:8080/*foo*-display

You will see a listing for all your portlets on the left and a description frame on the right. By clicking on one of your portlets, it will render (hopefully) in the right frame. If you see you made a mistake or are getting errors, you can fix things in the project and then redeploy the portlet with the same deploy ant target. JBoss is smart enough to undeploy the old version before the new one deploys. If you want to remove the portlet project (and the portlet harness as well):

```
> ant undeploy.plt-app
```

3. Deploying to the IceAxe Portal

Once you are confident that your portlet project is stable and functional, you can prepare it to be deployed into IceAxe. This time you must have JBoss running, and both DAQ-PROD and IceAxe deployed before continuing.

```
> ant portlets              # Will build just the WAR file
> ant deploy.portlets       # Will build and deploy into IceAxe
> ant undeploy.portlets *
```

* The last is highly dangerous as the portlets are undeployed from a running and deployed Portal context. The worst thing is that you will crash all or part of the Portal. This is bad sportsmanship if others are using the Portal to develop/debug their own code. It is easier to simply redeploy a new version.

11. Building the portlets into an RPM.

You need a .spec file. Use the template spec file or copy one in from an existing DAQ portlet project. Edit to suit your needs. CAUTION: This spec file and the existing ant command will only build portlet projects into an rpm designed for use with IceAxe.

```
ant build.portlet.rpm
```

III. References

BFD

Mercury Platform

The Icecube Software Developers Guide

Other topics for discussion:

a. Proper layout of JSPs to allow for better compile and code readability.
b. Merging in other display project. The portlet project layout makes this easy.
c. Using the Icecube Stylesheet. Minimal rework to existing portlets.