DocV1.portlet-container

The portlet container service

When a portlet application is deployed, only the portlet.xml and web.xml files are registered into the container. This phase simply uses a ServletListener object that register and unregister the files when the context of the application is deployed and undeployed.

Therefore, when a user sends a request to the portal web application (1), the portal decodes the incoming parameters to extract the portlet application name and portlet name (2) and redirects the request using the RequestDispatcher include() (4) method. What is necessary to understand here is that the request dispatcher is accessed using the portlet application context obtained using the portal ServletContext.

```
ServletContext portletContext = portalContext.getContext("/" +
windowInfos.getPortletApplicationName()); RequestDispatcher dispatcher =
portletContext.getRequestDispatcher(SERVLET_MAPPING); try {
dispatcher.include(request, response); } catch (ServletException e) {
throw new PortletContainerException(e); } catch (IOException e) { throw
new PortletContainerException(e); } finally{
((PortletPreferencesImp)windowInfos.getPreferences()).discard(); }
```

In the portlet application, a servlet used to wrapped portlets is then accessed. It extracts the information on which portlet to invoke with which incoming data and then delegates the work the PortletApplicationHandler class. This object obtains the portlet instance from the PortletApplicationProxy that instantiates it and calls the init() method if this is the first request to call the portlet. Then the handler calls either the processAction() or render() methods of the portlet.

Note that we have splited the ServletWrapper and PortletApplicationHandler in two in order to be able to unit test the portlet-container without launching the application server. This design which implied some more work was really a good choice as we highly used unit tests to develop the container.

The following code content is not that important, as such; it is to show that we have made custom code to avoid request dispatching and consequently the use of a servlet engine.

```
if (Environment.getInstance().getPlatform() == Environment.STAND_ALONE)
{ try { URLClassLoader oldCL = (URLClassLoader)
Thread.currentThread().getContextClassLoader(); URL[] urls = {new
URL(PORTLET_APP_PATH + "WEB-INF/classes/"), new
URL("file:./lib/portlet-api.jar"), new URL(PORTLET_APP_PATH +
"WEB-INF/lib/")}; Thread.currentThread().setContextClassLoader(new
URLClassLoader(urls)); try { return
standAloneHandler.process(portalContext, request, response, input,
output, windowInfos, isAction); } finally {
Thread.currentThread().setContextClassLoader(oldCL); } } catch
(MalformedURLException e) { e.printStackTrace(); } }
```

The eXo portlet container service is a facade to a portlet API compliant portlet container.

The portlet container is only accessed through a simple facade. Of course as the container is a component it can be either injected in another component constructor or looked up using the service locator.

This facade approach, combined with the IoC mechanism, makes it simple for portal vendors to integrate eXo portlet container.

The event package is an extension of the portlet API specification that allows interportlet-communication.

The filter package is also an extension that intoduce the PortletFilter concept.

Several methods are used to set or get the supported modes and states of the portal. Indeed, this is the responsability of the portal to inform the container of custom features it supports. If not, the container may use a pre-defined set of WindowStates and PortletModes.

The getPortletMetaData() returns a map of PortletData objects which contain many information about the portlet.

It is also possible to get ResourcesBundles objects packages with portlets in order to support internationalization (i18n) within your portal admin tools.

Through the set and getPortletPreferences() methods you can query/modify the preferences values of a concrete portlet instance.

Last, but not least, the processAction() and render() methods allows your to call the associated methods defined in the portletAPI.

The Portlet Container Invoker (PCI) package contains value object used to provide information to the portlet container.

The portal is responsible to provide Input and Output objects to the portlet container. According to the type of request those can be of type ActionX or RenderX.

One important thing is to give enougth information to the portlet container so that it is able to produce URLs that fit your portal needs.

To provide a custom URL template you need to provide a PortletURLFactory in the Input object.

```
public interface PortletURLFactory { public static final String RENDER =
"render"; public static final String ACTION = "action"; public
PortletURL createPortletURL(String Type);}
```

The portlet container will then use that factory to create the PortletURL objects to be used in your portlets and therefore will generate the URL you want. To help you in that task, we provide a BasePortletURL abstrat class, you just need to implement the toString() method.

The following source code tells you how to register the factory in the Input object :

```
//prepare the Input object RenderInput input = new RenderInput(); [...]
input.setPortletURLFactory(portletURLFactory);
```

And that's all...you have a portal with a JSR 168 certified portlet container.
DocV1.portlet-container (en)
CrÃ©ateur: XWiki.exo Creation Date: 2005/02/09 21:39
Dernier Auteur: XWiki.exo Last Modification Date: 2005/02/15 23:09