DocV1.PortletAPIextensions

Porlet API extensions

First, the eXo platform supports most of the non mandatory features and suggestion defined in the specifications such as :

- Caching:

Each portlet content can be cached in a per user map to reduce portal page creation time. The implementation of this feature uses Aspect Oriented Programmation (AOP) and the AspectJ language. At build time we weave several aspects, as described in the previous article, to the class that calls the portlets instances, the cache aspect is one of these. When cache is enabled we look in the advice to see if the content has already been generated. If so we directly return it, if not we move to the next aspect. The cache is discarded when the processAction method is called or when the expiration period has elapsed.

The portlet API specifications lets you define cache in the portlet.xml file :

```
<portlet> [...] <expiration-cache>0</expiration-cache> [...] </portlet>
```

You can use several values :

- -1 means that cache never expires
- 0 means that cache is disabled
- n represents any second integer before the cache is discarded.

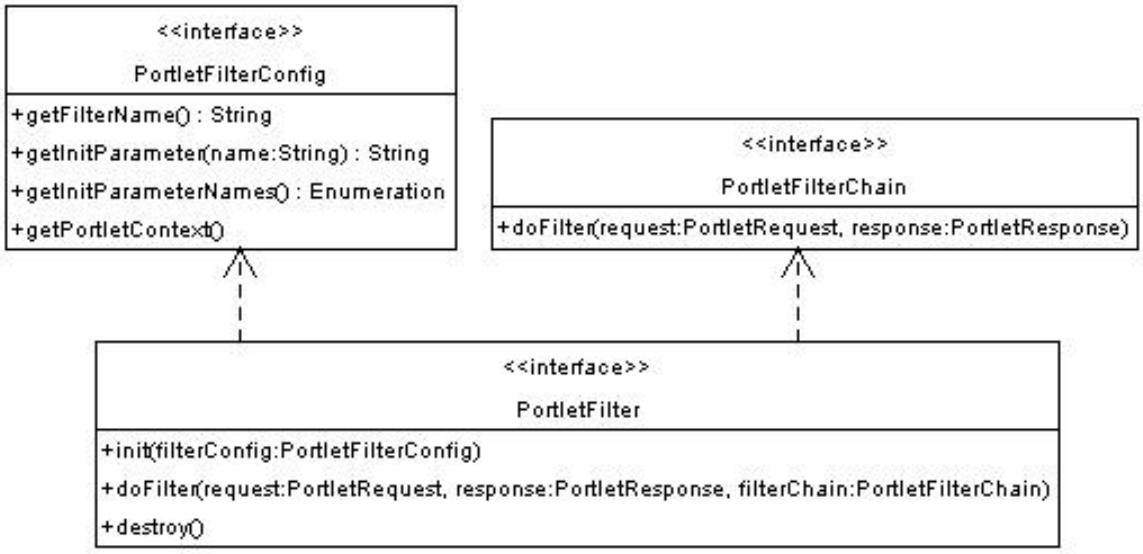
Cache can be completely disabled by defining it in the portlet-container.xml configuration file :

```
<cache> <enable>false</enable> </cache>
```

Note that we also cache PortletPreferences object once they have been extracted from the underlying storage system to avoid calls to the database on each portlet request.

- Portlet filters:

filters are on the list, in the suggestions chapter, for the next version of the portlet specifications. We have implemented this feature reproducing as well as possible the servlet filter's API. The interfaces used are very similar to the servlet ones.



The main difference is that filters are defined per portlet. To add a filter the portlet developer must define it in the portlet.xml file.

```
<portlet> <filter> <filter-name>LoggerFilter</filter-name>
<filter-class>exo.services.portletcontainer.test.filter.LoggerFilter</filter-class>
<init-param> <name>default-param</name>
<value>default-param-value</value> </init-param> </filter> [...]
</portlet>
```

For example, it is good practise to use such filters to log access to portlets with non intrusive code. The class would look like :

```
public class LoggerFilter implements PortletFilter{ public void
init(PortletFilterConfig portletFilterConfig) throws PortletException {
if(!'default-param-value'.equals(portletFilterConfig.getInitParameter('default-param
throw new PortletException(); } public void doFilter(PortletRequest
portletRequest, PortletResponse portletResponse, PortletFilterChain
filterChain) throws IOException, PortletException { // do something
before the portlet is reached filterChain.doFilter(portletRequest,
portletResponse) ; // do something after the portlet is reached } public
void destroy() { } }
```

The filter chain is created from the list of filters defined in the portlet.xml file. The implementation is here also done using an AspectJ aspect. After any aspect has been proceeded, the filter aspect launches the portlet filter in a recurisve way.

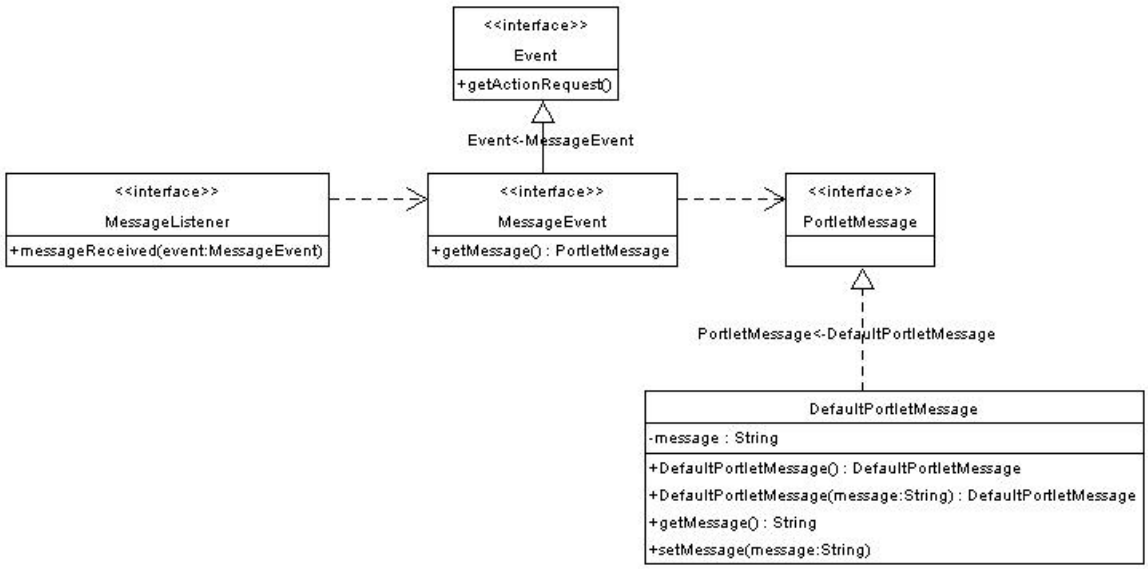
- Portlet inter-communication:

this feature is also a suggestion for the next version of the specifications. It lets a portlet sends an event to another one or broadcast the event within the scope of the portlet application. This message mechanism may, for instance, be used when a user clicks on the node of a file tree included in a portlet. An event can then be sent to another portlet that would modify its state and render, for example, the content of the node.

Here again, we extend the specifications' portlet.xml file and ask the developer to reference a MessageListener class.

```
<portlet> <message-listener>
<listener-name>SimpleMessageListener</listener-name>
<listener-class>exo.services.portletcontainer.test.listeners.SimpleMessageListener</
<description>a simple example</description> </message-listener>
<portlet-name>PortletThatReceivesMessage</portlet-name> [...] </portlet>
```

By convention, the portlet that sends the Message object must be aware of the type of the objects the listener can receive.



This mechanism can only occurs in a processAction method call, in other words before any render methods is called so that the state of portlets is consistent.

```
public class SimpleMessageListener implements MessageListener{ public
void messageReceived(MessageEvent messageEvent) throws PortletException
{ DefaultPortletMessage message = (DefaultPortletMessage)
messageEvent.getMessage(); } }
```

To be able to send a message you need to cast the PortletContext object to the ExoPortletContext interface which extends the specifications.

```
public class PortletThatSendsMessage extends GenericPortlet{ public void
processAction(ActionRequest actionRequest, ActionResponse
actionResponse) throws PortletException, IOException { ExoPortletContext
context = (ExoPortletContext)
actionRequest.getPortletSession().getPortletContext();
context.send('PortletThatReceivesMessage', new
DefaultPortletMessage('message sent'), actionRequest);
actionResponse.setRenderParameter('status', 'Everything is ok'); }
public void render(RenderRequest renderRequest, RenderResponse
renderResponse) throws PortletException, IOException { ExoPortletContext
context = (ExoPortletContext)
renderRequest.getPortletSession().getPortletContext();
context.send('PortletThatReceivesMessage', new
DefaultPortletMessage('message sent'), renderRequest); PrintWriter w =
renderResponse.getWriter(); w.println('Everything is ok'); } }
```

Note that this code is extracted from our unit tests set. Therefore the sent in the processAction() method will be executed as expected while the call done within the render() method will throw a PortletException.

We have built the portlet container as a real open source production choice and we have t herefore added several features to improve response time and memory management :

- Pooling :

the portlet specification is built on the top of the servlet specification. Therefore PortletRequest, PortletResponse, PortletSession objects and many wrappers must be created for each portlet call. When you have a large amount of portlets within a page, and many concurrent requests, the number of objects to instanciate can be very important. To avoid these problems we use pooled objects.

When a request comes to the portlet container, we borrow portlet objects and wrapper from the pool, fill them with the incoming information and call the portlet instance. When the portlet has generated the content and the request goes back to the portal, we release the pooled object.

You can configure the number of objects in the pool using the portlet-container.xml file :

```
<object-pool> <instances-in-pool>500</instances-in-pool> </object-pool>
```

- Support of shared sessions:

some application servers such as IBM WebSphere or Tomcat (4.1.29), in its default mode, support shared session. This simply means that when a request is dispatched to another web or portlet application context (corresponding to a WAR) the session of the first context is propagated to the context where the request is dispatched to.

This behaviour is not the one defined by the servlet specifications which impose the creation of a new session per context :

HttpSession objects must be scoped at the application (or servlet context) level. The underlying mechanism, such as the cookie used to establish the session, can be the same for different contexts, but the object referenced, including the attributes in that object, must never be shared between contexts by the container.

To illustrate this requirement with an example: if a servlet uses the RequestDispatcher to call a servlet in another Web application, any sessions created for and visible to the servlet being called must be different from those visible to the calling servlet.

To reduce the number of session object we decided to also support this mode and even make it the default one. Of course we tested our implementation with the TCK test suite and we claim compliance for this mode too.

This feature required substancial work and imagination as the portlet API imposes a unique session per portlet application. We therefore had to partition the main session with encoded attributes to simulate independent sessions' objects.

You may configure the use of share session in the portlet-container.xml file. Note that you also need to configure the underlying servlet container to the same session mode.

```
<shared-session> <enable>false</enable> </shared-session>
```

- Portlet lazy loading:

to manage memory resources we have decided to instantiate and init portlets only when they are called for the first time.

Here, we leverage pico-container capabilities. When portlets are deployed in the container, we reference them in what we call the ServiceManager. This is a singleton wrapper around a pico container instance.
DocV1.PortletAPIextensions (en)
CrÃ©ateur: XWiki.benjamin.mestrallet Creation Date: 2005/02/09 14:12
Dernier Auteur: XWiki.benjamin.mestrallet Last Modification Date: 2005/02/15 22:55