Portal: Event Handling (2.1 legacy document)

Table of contents

Table of contents

Warning:
This document was copied as is from the Cocoon 2.1 documentation, but has not yet been fully reviewed or moved to its new home.

1. Overview

This document gives an overview of the event handling of the portal engine.

The sample portal that comes with the Cocoon distribution contains several working samples for event handling.

2. Introduction

The event handling is a central mechanism used in the portal engine. Every change (changes in status or layout, links etc) is propagated through an event. The portal uses the publisher/subscribe paradigm, so each component that is interested in a specific event can subscribe itself for this type of event. In addition each component can send out events.

The processing of a portal request (a request send to the Cocoon portal) is divided into two phases: event handling and rendering. In the first phase all events are processed. For example if the user clicks a link this triggers an event that is published. Any receiver of this event might in turn fire new events that are published as well.

When all events are processed, the first phase is finished and the second phase, the rendering, is started. At this point of time all event handling and all information exchange should be finished.

3. Events and the request/response cycle

In the Portal, an event is represented by a Java object. This event object contains all necessary information to process the event. So, in most cases an event contains the object to modify, what to modify and the value to set. For example, the minimize event for minimizing a coplet, contains the coplet, the information to change the window state and the value "minimize" to set.

There are different types of events: a type for changing the window state, a type for removing a coplet, a type for links that are clicked by the user etc. Each event type is represented by a Java class (or interface).

A component that processes for example the window state request is subscribed to this minimize event (or: the corresponding class/interface) and when such an event is fired, it changes the window state of the coplet to minimize. Every data this component needs is stored in the event. This is a very important detail: the event is not directly processed (in this case) by the object that is changed (the coplet) but by a central subscribed component that changes the coplet. This is because of the publisher/subscribe mechanism used: many components in the portal can subscribe to the same event type if they are interested. So, each component that is interested in an event needs all information about this event. That's why all data is stored in the event itself.

Let's have a look how such an event is created:

```
Event event; event = new ChangeCopletInstanceAspectDataEvent( copletInstanceData,
"size", SizingStatus.STATUS_MINIMIZEDD);
```

Event is just a marker interface, the concrete implementation *ChangeCopletInstanceAspectDataEvent* implements this interface and requires three pieces of information: the CopletInstanceData, the information about what to change (size) and the new value.

All events must implement the marker interface *Event*, so if a component is interested in all Events it could subscribe itself using this event type.

If you want to fire an event, you have to publish it. Therefore you need the event manager, a central portal component. You can lookup this component, fire the event and release the manager again. If you fire the event, the event is directly published to all subscribed components.

```
EventManager manager = null; try { manager =
serviceManager.lookup(EventManager.ROLE); manager.getPublisher().publish(event); }
finally { serviceManager.release(manager); }
```

As noted above, the event will be directly fired. But usually in a portal application, events are not fired directly but are invoked by some user action. This means, the user clicks on a link in the browser, the request is targetted at Cocoon and the portal invokes (fires) the correct events.

For this, a link (or a form action) must know, which event it should fire, if it is clicked. So, in other words, a link is associated with a concrete event. But on the other site, an event is a Java object and we can only use strings in URLs. So how does this work?

The part of the portal that generates the link, creates the event object with all necessary data, transforms this event into a usable URI and this URI is the target of the link. When the user clicks on this link, the portal transforms the URI back into the Java event object and fires the event.

The transformation Event->URI->Event is done by another portal component, the link service. Most portal components (apart from the event manager) are available through another central component, the portal service. So you need to have access to the portal service component. (Renderers e.g. don't have to lookup the service by itself, they get it as a method parameter).

```
PortalService service = null; try { service = serviceManager.lookup(PortalService.ROLE);
LinkService ls = service.getComponentManager().getLinkService(); String uri = getLinkURI(
event ); // create a link that references the uri } finally { serviceManager.release(service); }
```

That's all you have to do: create the event object, get the link service, transform the event into a URI using the service and then create the (html) link using the URI. Everything else is handled by the portal for you.

In addition, you can transform several events into one single link. So you can create links, the user can click, that do several things at the same time (minimizing one coplet and maximizing another one etc.). The link service offers corresponding methods for this.

4. Changing the State of a Coplet

In most cases you want to change the state of a coplet because the user performed an action. The portal engine provides you with some events that you can directly use.

- CopletJXPathEvent
- ChangeCopletInstanceAspectDataEvent

The *CopletJXPathEvent* requires again three pieces of information: the coplet instance data to change, the JXPath expression that defines the data to change and the value:

```
Event event = new CopletJXPathEvent(copletInstanceData, "attributes/username",
username);
```

In the previous chapter, we already saw an example of the usage of the *ChangeCopletInstanceAspectDataEvent*.

It is of course possible that you write your own events for changing the state of a coplet. But in this

case make sure that your own event implements the interface *CopletInstanceEvent*. This helps the portal engine in tracking if a coplet has been changed.

5. Subscribing to Events

If you are interested in events, you can subscribe to a specific event type. As events are Java objects, you subscribe for all events of a specific interface or class (and all of the subclasses). Subscribing is done using the event manager:

```
EventManager manager = null; try { manager =
serviceManager.lookup(EventManager.ROLE); manager.getRegister().subscribe(
myComponent ); } finally { serviceManager.release(manager); }
```

The component you subscribe must implement the Subscriber interface:

```
Subscriber interface: public Class getEventType(); public void inform( Event event );
```

The getEventType() method returns the class/interfaces of the events the component is interested and each time such an event occurs, the inform() method is invoked.

For example one central component in the portal subscribes for all events dealing with coplets, so it returns *CopletInstanceEvent* as the class (interface) in getEventType().

6. Inter Coplet Communication

A very interesting feature of the portal is inter-coplet communication. The demo portal already has a simple sample where the name of an image selected in an image gallery is transfered to a different coplet.

Now, there is only one (minor) problem: in the cocoon portal coplets (or more precisly CopletInstanceData objects) are not components but just data objects. So, a coplet can't directly register itself as a subcriber for events.

Remember that we mentioned earlier on a central component that processes the change events for coplets? So, this is basically one possibility: if you want to pass information from one coplet to another one, create a CopletJXPathEvent and pass the information to the other coplet.

Imagine a form coplet where the user can enter a city. When this form is processed by the form coplet, it can generate one (or more) CopletJXPathEvents and push the entered city information to a weather coplet and a hotel guide coplet. So, these two coplets display the information about the selected city.

7. The Coplet Transformer

Apart from the possibility to create events from within your Java code, it's also possible to create events from within a pipeline by using for example the coplet transformer. It listens for elements with the namespace "http://apache.org/cocoon/portal/coplet/1.0".

7.1. The coplet element

The coplet element has nothing to do with events :) It can be used to include information about the current coplet in the SAX stream:

```
... <coplet:coplet select="attributes/name"/> ...
```

The coplet element can only be used inside a coplet pipeline, but not in the main portal pipeline. The select attribute defines an JXPath expression that is used to fetch the value that is included in the

stream.

7.2. The link element

The link element creates a link that will trigger an event if the user clicks this link:

```
... <coplet:link coplet="COPLET_ID" path="JXPATH" value="TO_SET"/> <coplet:link
layout="LAYOUT_ID" path="JXPATH" value="TO_SET"/> ...
```

This element generates an HTML link which will eiter trigger an event to change a coplet instance data or a layout based on the JXPath and the value provided.

8. Configuring Subscribers

In the previous chapters we saw one possibility to subscribe: dynamically in some Java code. This requires that - of course - this code is executed at some point of time. This is a solution for dynamic subscribers, which means a subscriber that is only "available" if a specific feature of the portal is used. If the feature is available, the "feature" subscribes itself (or another component).

However, this adds an exta burdon to the development of own events and their subscribers. Therefore it is possible to configure subscribers in the cocoon.xconf. These subscribers are instantiated by the portal engine on startup of Cocoon and subscribed by the portal engine.

You have two possibilites, you can either subscribe Avalon components or classes. In the first case, you configure the role of the component. Then the portal engine looks up this component and subscribes it.

If you configure a class, the portal engine creates an instance of this class using the no-argument constructor and subscribes this instance. For convenience, this instance can implement the Avalon lifecycle interface like LogEnabled or Serviceable.

The configuration takes place in the cocoon.xconf as a configuration for the event manager:

```
... <component class="org.apache.cocoon.portal.event.impl.DefaultEventManager"
logger="portal" role="org.apache.cocoon.portal.event.EventManager"> ... <!-- add a new
instance of each class as a subscriber: --> <subscriber-classes> <class
name="org.apache.cocoon.portal.event.subscriber.impl.DefaultJXPathEventSubscriber"/>
</subscriber-classes> <!-- add each component as a subscriber (the component should be
thread safe): --> <subscriber-roles> <role
name="org.apache.cocoon.portal.samples.location.LocationEventSubscriber"/>
</subscriber-roles> </component> ...
```

In the sample configuration above, one class is subscribed (the *DefaultJXPathEventSubscriber*) and one Avalon component (the *LocationEventSubscriber*).

So, if you write your own events and your own subscribers you can either dynamically add them during execution or statically add them by configuration as shown above.

9. Further Information

The event.impl package contains all currently processed events, so you can study the events and see how to create them. In general most events are created inside the renderers, especially the renderer aspects that render specific details (e.g. the sizing buttons for a coplet). So, you can have a look at the code as well.

There are several transformers that help in creating events inside a Cocoon pipeline. For example the

coplet transformer can be used to create links that contain events to change the status of a coplet or a layout object. The gallery sample uses this transformer as a demo.

1. Comments

add your comments