SimpleEventHandling

Simple Event Handling

RichardRodgers 8-June-06

Approach

- To satisfy the design requirements outlined in the EventMechanism page, but also to avoid incurring performance penalties, a 'hybrid' approach is developed here. That is, we can exploit the power and extensibility of a JMS solution, but are not required to do so.

Logical Model

- Events are managed by 2 entities: *consumers* and *processors*. These can be thought of as roughly analagous to *subscribers* and *publishers* in a standard messaging model, but it is only a loose analogy. The real publisher in this approach, i.e. creators of event messages, is the content implemenation itself, i.e. the classes implementing the content objects (Item, Collection, etc). The job of a processor is simply to hand events to a list of consumers it manages, performing any event filtering that is requested. Thus a complete configuration consists of:
- the name of a class implementing the *Processor* interface
- a list of one or more names of classes implementing the *Consumer* interface
- for each consumer, a filter governing which events that consumer wishes to see in the context of the configuration
- These configurations are themselves named, and any number may be defined. There is a special configuration named 'default', which is used everywhere unless otherwise specified. Configurations are defined in dspace.cfg. For example:

1. the default processor - do not remove
   event.processor.default = org.dspace.event.BasicProcessor
   event.processor.default.consumers = \
   org.dspace.event.SearchConsumer:0407, \
   org.dspace.event.BrowseConsumer:0407, \
   org.dspace.event.HistoryConsumer:0407 }}}

This configuration means: use the class `BasicProcessor` with consumer `SearchConsumer` using the filter 0407, followed by `BrowseConsumer` etc – which is essentially what DSpace currently does without any event mechanism: when an Item is updated, the search and browse indices are updated, and the history system records the change.

- Every DSpace *Context* object contains a list onto which events are posted by the Content classes. Whenever a context is 'completed', the list is passed to the 'default' processor configuration. If special handling is desired, the context can be set to point to another configuration, using `setEventProcessor(String name)` method of `Context`.

- DSpace can be customized to some degree without any coding. Since processor configurations are editable, you can change behavior simply by altering the 'default' config. For example, if you run 'index-all' every night, and don't care about immediate updates, you could simply comment out the first 2 consumer lines, and speed up the system.

- Both consumer and processor are defined with very simple interfaces. For example, the Consumer interface has one method:

```
public void consume(Context context, Event event);
```

Hence for many purposes, using the event mechanism is as simple as writing a class that implements this method. The prototype code includes consumers for search, browse,
history, and even one for the subscriber email notifications.

Where's JMS?

- As noted above, JMS is not required for simple operation where all consumers operate synchronously within a single JVM. But in more complex, distributed scenarios, one may wish DSpace to defer some operations. Here's where JMS comes in. Consider the following configurations:

1. the default processor - do not remove
   event.processor.default = org.dspace.event.BasicProcessor
   event.processor.default.consumers = \
   org.dspace.event.HistoryConsumer:0407, \
   org.dspace.event.JMSConsumer:ffff

1. the JMS processor

```
event.processor.jms = org.dspace.event.JMSProcessor
event.processor.jms.consumers = \
org.dspace.event.SearchConsumer:0407, \
org.dspace.event.BrowseConsumer:0407 }}}
```

Here, only history system updates are performed immediately (i.e. synchronously), and the events are passed to the `JMSConsumer` which simply posts the events as messages
onto a persistent JMS message queue. The `JMSProcessor`, which includes a main method so it can be invoked via the 'dsrun' utility, will then read the queue, and pass the
events onto its consumers, in this case search and browse.

Prototype code to be posted shortly. Comments welcome