CurationSystem

This page may describe proposed implementation or an implementation for an older version of DSpace. For official information, you should refer to the official documentation for your particular DSpace version:

DSpace 1.7.x: Curation System

DSpace 1.8.x: Curation System

DSpace 3.x: Curation System

Curation System for DSpace 1.7

This document is a high-level - but developer-focused - introduction to the curation system being proposed for DSpace 1.7 It presumes knowledge of java and DSpace internals.

An initial, higher level proposal for this Curation System is available at CurationTaskProposal.

Code Availability

The curation system as described here has been implemented in a branch (roughly 1.7 compatible) at:

http://scm.dspace.org/svn/repo/dspace/branches/dspace-curation

This code includes a few demonstration tasks: one to tabulate and display bitstream formats and support-levels thereof, and one that checks whether items have all required metadata fields (as defined by input-forms.xml). You should be able to check out and build this branch to take curation for a test drive, or begin to design and test new tasks.

Also visit pages that describe useful sets of curation tasks that have been developed:

- Virus Scanning: Virus Scan Curation Task
- Content Replication: ReplicationTaskSuite

Tasks

The goal of the curation system ('CS') is to provide a simple, extensible, way to manage routine content operations on a repository. These operations are known to CS as 'tasks', and they can operate on any DSpaceObject (i.e. subclasses of DSpaceObject) - although the first incarnation will only understand Communities, Collections, and Items - viz. core data model objects. Tasks may essentially work on only one type of DSpace object - typically an item - and in this case they may simply ignore other data types (tasks have the ability to 'skip' objects for any reason). The DSpace core distribution ought to provide a number of useful tasks, but the system is designed to encourage local extension - tasks can be written for any purpose, and placed in any java package. What sorts of things are appropriate tasks?

Some examples:

- apply a virus scan to item bitstreams (this will be our example below)
- profile a collection based on format types - good for identifying format migrations
- ensure a given set of metadata fields are present in every item, or even that they have particular values
- call a network service to enhance/replace/normalize an items's metadata or content
- ensure all item bitstreams are readable and their checksums agree with the ingest values

A task can be arbitrary code, but the class implementing it must have 2 properties:

First, it must provide a no-arg constructor, so it can be loaded by the PluginManager. Thus, all tasks are 'named' plugins, meaning that each must be configured in dspace.cfg as:

```
plugin.named.org.dspace.curate.CurationTask = \
org.dspace.curate.ProfileFormats = format-profile \
org.dspace.curate.RequiredMetadata = req-metadata \
org.dspace.ctask.replicate.Audit = audit \
org.dspace.ctask.replicate.Estimate = estimate \
org.dspace.ctask.replicate.Generate = generate \
org.dspace.ctask.integrity.Checksum = checksum \
org.dspace.ctask.integrity.ClamScan = vscan
```

The 'plugin name'(audit, estimate, etc) is called the task name, and is used instead of the qualified class name wherever it is needed (on the cmd line, etc) - the CS always dereferences it.

Second, it must implement the interface 'org.dspace.curate.CurationTask'

The CurationTask interface is almost a 'tagging' interface, and only requires a few very high-level methods be implemented. The most significant is:

```
int perform(DSpaceObject dso);
```

The return value should be a code describing one of 4 conditions:

- 0 : SUCCESS the task completed successfully
- 1 : FAIL the task failed (it is up to the task to decide what 'counts' as failure - an example might be that the virus scan finds an infected file)
- 2 : SKIPPED the task could not be performed on the object, perhaps because it was not applicable
- -1 : ERROR the task could not be completed due to an error

If a task extends the AbstractCurationTask class, that is the only method it needs to define.

Task Invocation

Tasks are invoked using CS framework classes that manage a few details (to be described below), and this invocation can occur wherever needed, but CS offers great versatility "out of the box":

On the command line

A simple tool "CurationCli" provides access to CS via command line. For example, to perform a virus check on collection "4":

```
[dspace]/bin/dspace curate -t vscan -i 123456789/4
```

or

```
[dspace]/bin/dspace dsrun org.dspace.curate.CurationCli -t vscan -i 123456789/4
```

As with other command-line tools, these invocations could be placed in a cron table and run on a fixed schedule, or run on demand by an administrator.

In the admin UI

In the XMLUI, there is a 'Curate' tab (appearing within the 'Edit Community/Collection/Item') that exposes a drop-down list of configured tasks, with a button to 'perform' the task, or queue it for later operation (see section III below). You may filter out some of the defined tasks (not appropriate for UI use), by means of a configuration property. This property also permits you to assign to the task a 'prettier' name than the PluginManager task name.

In workflow

CS provides the ability to attach any number of tasks to standard DSpace workflows. Using a configuration file (workflow-curation.xml), you can declaratively (without coding) wire tasks to any step in a workflow. An example:

```
<taskset name="cautious">
  <flowstep name="step1">
    <task name="vscan">
      <workflow>reject</workflow>
      <notify on="fail">$flowgroup</notify>
      <notify on="fail">$colladmin</notify>
      <notify on="error">$siteadmin</notify>
    </task>
  </flowstep>
</taskset>
```

This markup would cause the virus scan to occur during step one of workflow, and automatically reject any submissions with infected files. It would further notify (via email) both the reviewers (step 1 group), and the collection administrators, if either of these are defined. If it could not perform the scan, the site administrator would be notified.

The notifications use the same procedures that other workflow notifications do - namely email. There is a new email template defined for curation task use (in dspace/config/emails): 'flowtask_notify'. This may be language-localized or otherwise modified like any other email template.

Like configurable submission, you can assign these task rules per collection, as well as having a default for any collection.

In arbitrary user code

If these pre-defined ways are not sufficient, you can of course manage curation directly in your code. You would use the CS helper classes. For example:

```
Collection coll = (Collection)HandleManager.resolveToObject(context, "123456789/4");
Curator curator = new Curator();
curator.addTask("vscan").curate(coll);
System.out.println("Result: " + curator.getResult("vscan"));
```

would do approximately what the command line invocation did. the method 'curate' just performs all the tasks configured
(you can add multiple tasks to a curator).

Asynchronous (Deferred) Operation

Because some tasks may consume a fair amount of time, it may not be desirable to run them in an interactive context. CS provides a simple API and means to defer task execution, by a queuing system. Thus, using the previous example:

```
Curator curator = new Curator();
curator.addTask("vscan").queue(context, "monthly", "123456789/4");
```

would place a request on a named queue "monthly" to virus scan the collection. To read (and process) the queue, we could for example:

```
[dspace]/bin/dspace dsrun org.dspace.curate.CurationCli -q monthly
```

use the command-line tool, but we could also read the queue programmatically. Any number of queues can be defined and used as needed.
In the administrative UI curation 'widget', there is the ability to both perform a task, but also place it on a queue for later processing.

Task Output and Reporting

Few assumptions are made by CS about what the 'outcome' of a task may be (if any) - it. could e.g. produce a report to a temporary file. But the CS runtime does provide a few pieces of information that a task can assign:

Status Code

This was mentioned above. This is returned to CS whenever a task is called. In addition to the task-assigned codes, there are values:

```
      NOTASK - CS could not find the requested task
      UNSET  - task did not return a status code because it has not yet run
```

Result String

The task may define a string indicating details of the outcome. This result is displayed, e.g. in the 'curation widget' described above:

```
       "Virus 12312 detected on Bitstream 4 of 1234567789/3"
```

CS does not interpret or assign result strings, the task does it.

Reporting Stream

This is not currently fully implemented, just writes to standard out. But if more details should be recorded, they can be pushed to this stream.

All 3 are accessed (or set) by methods on the Curation object:

```
Curator curator = new Curator();
curator.addTask("vscan").curate(coll);
int status = curator.getStatus("vscan");
```

Task Annotations

CS looks for, and will use, certain java annotations in the task Class definition that can help it invoke tasks more intelligently. An example may explain best. Since tasks operate on DSOs that can either be simple (Items) or containers (Collections, and Communities), there is a fundamental problem or ambiguity in how a task is invoked: if the DSO is a collection, should the CS invoke the task on each member of the collection, or does the task 'know' how to do that itself? The decision is made by looking for the @Distributive annotation: if present, CS assumes that the task will manage the details, otherwise CS will walk the collection, and invoke the task on each member. The java class would be defined:

```
@Distributive
public class MyTask implements CurationTask
```

A related issue concerns how non-distributive tasks report their status and results: the status will normally reflect only the last invocation of the task in the container, so important outcomes could be lost. If a task declares itself @Suspendable, however, the CS will cease processing when it encounters a FAIL status. When used in the UI, for example, this would mean that if our virus scan is running over a collection, it would stop and return status (and result) to the scene on the first infected item it encounters. You can even tune @Supendable tasks more precisely by annotating what invocations you want to suspend on. For example:

```
@Suspendable(invoked=Curator.Invoked.INTERACTIVE)
public class MyTask implements CurationTask
```

would mean that the task would suspend if invoked in the UI, but would run to completion if run on the command-line.

Only a few annotation types have been defined so far, but as the number of tasks grow, we can look for common behavior that can be signaled by annotation. For example, there is a @Mutative type: that tells CS that the task may alter (mutate) the object it is working on.