Curation System

DSpace supports running curation tasks, which are described in this section. DSpace includes several useful tasks out-of-the-box, but the system also is designed to allow new tasks to be added between releases, both general purpose tasks that come from the community, and locally written and deployed tasks.

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) - which means the entire Site, Communities, Collections, and Items - viz. core data model objects. Tasks may elect to 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 will 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. This gives DSpace sites the ability to customize the behavior of their repository without having to alter - and therefore manage synchronization with - the DSpace source code. What sorts of activities are appropriate for 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 item's metadata or content
- ensure all item bitstreams are readable and their checksums agree with the ingest values

Since tasks have access to, and can modify, DSpace content, performing tasks is considered an administrative function to be available only to knowledgeable collection editors, repository administrators, sysadmins, etc. No tasks are exposed in the public interfaces.

Activation

For CS to run a task, the code for the task must of course be included with other deployed code (to `[dspace]/lib`, WAR, etc) but it must also be declared and given a name. This is done via a configuration property in `[dspace]/config/modules/curate.cfg` as follows:

```
### Task Class implementations
plugin.named.org.dspace.curate.CurationTask = org.dspace.ctask.general.NoOpCurationTask = noop
plugin.named.org.dspace.curate.CurationTask = org.dspace.ctask.general.ProfileFormats = profileformats
plugin.named.org.dspace.curate.CurationTask = org.dspace.ctask.general.RequiredMetadata = requiredmetadata
plugin.named.org.dspace.curate.CurationTask = org.dspace.ctask.general.ClamScan = vscan
plugin.named.org.dspace.curate.CurationTask = org.dspace.ctask.general.MicrosoftTranslator = translate
plugin.named.org.dspace.curate.CurationTask = org.dspace.ctask.general.MetadataValueLinkChecker = checklinks
```

For each activated task, a key-value pair is added. The key is the fully qualified class name and the value is the *taskname* used elsewhere to configure the use of the task, as will be seen below. Note that the curate.cfg configuration file, while in the config directory, is located under "modules". The intent is that tasks, as well as any configuration they require, will be optional "add-ons" to the basic system configuration. Adding or removing tasks has no impact on dspace.cfg.

For many tasks, this activation configuration is all that will be required to use it. But for others, the task needs specific configuration itself. A concrete example is described below, but note that these task-specific configuration property files also reside in `[dspace]/config/modules`

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 the command line. This tool bears the name "curate" in the DSpace launcher. For example, to perform a virus check on collection "4":

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

The complete list of options:

option	meaning
-t taskname	name of task to perform.
-T filename	name of file containing a list of tasknames to be performed.
-e epersonID	(required) email address or netid of the E-Person performing the task
-i identifier	ID of object to curate. May be (1) a Handle, (2) a workflow ID, or (3) 'all' to operate on the whole repository.
-q queue	name of queue to process. -i and -q are mutually exclusive.
-l limit	maximum number of objects in Context cache. If absent, unlimited objects may be added.
-s scope	declare a scope for database transactions. Scope must be: (1) 'open' (default value), (2) 'curation' or (3) 'object'.
-v	emit verbose output
-r filename	emit reporting to the named file. '-r -' writes reporting to standard out. If not specified, report is discarded silently.
-p name=value	set a runtime task parameter `name` to the value `value`. May be repeated as needed. See "Task parameters" below.

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 UI, there are several ways to execute configured Curation Tasks:

1. **From the "Curate" tab/button that appears on each "Edit Community/Collection/Item" page:** this tab allows an Administrator, Community Administrator or Collection Administrator to run a Curation Task on that particular Community, Collection or Item. When running a task on a Community or Collection, that task will also execute on all its child objects, unless the Task itself states otherwise (e.g. running a task on a Collection will also run it across all Items within that Collection).
    - NOTE: Community Administrators and Collection Administrators can only run Curation Tasks on the Community or Collection which they administer, along with any child objects of that Community or Collection. For example, a Collection Administrator can run a task on that specific Collection, or on any of the Items within that Collection.
2. **From the Administrator's "Curation Tasks" page:** This option is only available to DSpace Administrators, and appears in the Administrative side-menu. This page allows an Administrator to run a Curation Task across a single object, or all objects within the entire DSpace site.
    - In order to run a task from this interface, you must enter in the handle for the DSpace object. To run a task site-wide, you can use the handle: `[your-handle-prefix]/0`

Each of the above pages exposes a drop-down list of configured tasks, with a button to 'perform' the task, or queue it for later operation (see section below). Not all activated tasks need appear in the Curate tab - you filter them by means of a configuration property. This property also permits you to assign to the task a more user-friendly name than the PluginManager *taskname*. The property resides in `[dspace]/config/modules/curate.cfg`:

```
curate.ui.tasknames = profileformats = Profile Bitstream Formats
curate.ui.tasknames = requiredmetadata = Check for Required Metadata
```

When a task is selected from the drop-down list and performed, the tab displays both a phrase interpreting the "status code" of the task execution, and the "result" message if any has been defined. When the task has been queued, an acknowledgement appears instead. You may configure the words used for status codes in curate.cfg (for clarity, language localization, etc):

```
curate.ui.statusmessages = -3 = Unknown Task
curate.ui.statusmessages = -2 = No Status Set
curate.ui.statusmessages = -1 = Error
curate.ui.statusmessages = 0 = Success
curate.ui.statusmessages = 1 = Fail
curate.ui.statusmessages = 2 = Skip
curate.ui.statusmessages = other = Invalid Status
```

Report output from tasks run in this way is collected by configuring a Reporter plugin. You must have exactly one Reporter configured. The default is to use the FileReporter, which writes a single report of the output of all tasks in the run over all of the selected objects, to a file in the reports directory (configured as report.dir). See `[DSpace]/config/modules/submission-configuration.cfg` for the value of `plugin.single.org.dspace.curate.Reporter`. Other Reporter implementations are provided, or you may supply your own.

As the number of tasks configured for a system grows, a simple drop-down list of **all** tasks may become too cluttered or large. DSpace 1.8+ provides a way to address this issue, known as *task groups*. A task group is a simple collection of tasks that the Admin UI will display in a separate drop-down list. You may define as many or as few groups as you please. If no groups are defined, then all tasks that are listed in the *ui.tasknames* property will appear in a single drop-down list. If at least *one* group is defined, then the admin UI will display **two** drop-down lists. The first is the list of task groups, and the second is the list of task names associated with the selected group. A few key points to keep in mind when setting up task groups:

- a task can appear in more than one group if desired
- tasks that belong to no group are *invisible* to the admin UI (but of course available in other contexts of use)

The configuration of groups follows the same simple pattern as tasks, using properties in `[dspace]/config/modules/curate.cfg`. The group is assigned a simple logical name, but also a localizable name that appears in the UI. For example:

```
# ui.taskgroups contains the list of defined groups, together with a pretty name for UI display
curate.ui.taskgroups = replication = Backup and Restoration Tasks
curate.ui.taskgroups = integrity = Metadata Integrity Tasks
.....
# each group membership list is a separate property, whose value is comma-separated list of logical task names
curate.ui.taskgroup.integrity = profileformats, requiredmetadata
....
```

In workflow

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

```
<taskset-map>
    <mapping collection-handle="default" taskset="cautious" />
</taskset-map>
<tasksets>
    <taskset name="cautious">
        <flowstep name="editstep">
            <task name="vscan">
                <workflow>reject</workflow>
                <notify on="fail">$flowgroup</notify>
                <notify on="fail">$colladmin</notify>
                <notify on="error">$siteadmin</notify>
            </task>
        </flowstep>
    </taskset>
</tasksets>
```

This markup would cause a virus scan to occur during the "editstep" of workflow for any collection, and automatically reject any submissions with infected files. It would further notify (via email) both the reviewers ("editstep" role/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: `[dspace]/config/emails/flowtask_notify`. This may be language-localized or otherwise modified like any other email template.

Tasks wired in this way are normally performed *as soon as the workflow step is entered*, and the outcome action (defined by the 'workflow' element) immediately follows. It is also possible to delay the performance of the task - which will ensure a responsive system - by queuing the task instead of directly performing it:

```
...
    <taskset name="cautious">
        <flowstep name="editstep" queue="workflow">
...
```

This attribute (which must always follow the "name" attribute in the flowstep element), will cause all tasks associated with the step to be placed on the queue named "workflow" (or any queue you wish to use, of course), and further has the effect of **suspending** the workflow. When the queue is emptied (meaning all tasks in it performed), then the workflow is restarted. Each workflow step may be separately configured,

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

As with task invocation from the administrative UI, workflow tasks need to have a Reporter configured in `submission-configuration.cfg`.

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.setReporter(System.out);
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).

The above directs report output to standard out. Any class which implements Appendable may be set as the reporter class.

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 curate -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, it could modify DSpace content silently, etc. But the CS runtime does provide a few pieces of information whenever a task is performed:

Status Code

This was mentioned above. This is returned to CS whenever a task is called. The complete list of values:

```
 -3 NOTASK  - CS could not find the requested task
 -2 UNSET   - task did not return a status code because it has not yet run
 -1 ERROR   - task could not be performed
  0 SUCCESS - task performed successfully
  1 FAIL    - task performed, but failed
  2 SKIP    - task not performed due to object not being eligible
```

In the administrative UI, this code is translated into the word or phrase configured by the *ui.statusmessages* property (discussed above) for display.

Result String

The task may define a string indicating details of the outcome. This result is displayed, 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. A task may not assign a result, but the "best practice" for tasks is to assign one whenever possible.

Reporting Stream

For very fine-grained information, a task may write to a *reporting* stream. This stream may be sent to a file or to standard out, when running a task from the command line. Tasks run from the administrative UI or a workflow use a configured Reporter class to collect report output. Your own code may collect the report using any implementation of the Appendable interface. Unlike the result string, there is no limit to the amount of data that may be pushed to this stream.

Task Properties

DSpace 1.8 introduces a new "idiom" for tasks that require configuration data. It is available to any task whose implementation extends `AbstractCurationTask`, but is completely optional. There are a number of problems that task properties are designed to solve, but to make the discussion concrete we will start with a particular one: the problem of hard-coded configuration file names. A task that relies on configuration data will typically encode a fixed reference to a configuration file name. For example, the virus scan task reads a file called "`clamav.cfg`", which lives in `[dspace]/config/modules`. It could look up its configuration properties in the ordinary way. But tasks are supposed to be written by anyone in the community and shared around (without prior coordination), so if another task uses the same configuration file name, there is a name **collision** here that can't be easily fixed, since the reference is hard-coded in each task. In this case, if we wanted to use both at a given site, we would have to alter the source of one of them - which introduces needless code localization and maintenance.

Task properties gives us a simple solution. Here is how it works: suppose that both colliding tasks instead use the task properties facility instead of ordinary configuration lookup. For example, each asks for the property `clamav.service.host`. At runtime, the curation system **resolves** this request to a set of configuration properties, and it uses the *name the task has been configured as* as the prefix of the properties. So, for example, if both were installed (in, say, `curate.cfg`) as:

```
org.dspace.ctask.general.ClamAv = vscan,
org.community.ctask.ConflictTask = virusscan,
....
```

then the task property `foo` will resolve to the property named `vscan.foo` when called from ClamAv task, but `virusscan.foo` when called from ConflictTask's code. Note that the "`vscan`" etc are locally assigned names, so we can always prevent the "collisions" mentioned, and we make the tasks much more portable, since we remove the "hard-coding" of config names.

Another use of task properties is to support multiple task profiles. Suppose we have a task that we want to operate in one of two modes. A good example would be a mediafilter task that produces a thumbnail. We can either create one if it doesn't exist, or run with "-force" which will create one regardless. Suppose this behavior was controlled by a property in a config file. If we configured the task as "thumbnail", then we would have in (perhaps) `[dspace]/config/modules/thumbnail.cfg`:

```
...other properties...
thumbnail.thumbnail.maxheight = 80
thumbnail.thumbnail.maxwidth = 80
thumbnail.forceupdate=false
```

The thumbnail generating task code would then resolve "forcedupdate" to see whether filtering should be forced.

But an obvious use-case would be to want to run force mode **and** non-force mode from the admin UI on different occasions. To do this, one would have to stop Tomcat, change the property value in the config file, and restart, etc However, we can use task properties to elegantly rescue us here. All we need to do is go into the config/modules directory, and create a new file perhaps called: `thumbnail.force.cfg`. In this file, we put the properties:

```
thumbnail.force.thumbnail.maxheight = 80
thumbnail.force.thumbnail.maxwidth = 80
thumbnail.force.forceupdate=true
```

Then we add a new task (really just a new name, no new code) in curate.cfg:

```
org.dspace.ctask.general.ThumbnailTask = thumbnail
org.dspace.ctask.general.ThumbnailTask = thumbnail.force
```

Consider what happens: when we perform the task "`thumbnail`" (using taskProperties), it uses the `thumbnail.*` properties and operates in "non-force" profile (since the value is false), but when we run the task "`thumbnail.force`" the curation system uses the `thumbnail.force.*` properties. Notice that we did all this via local configuration - we have not had to touch the source code at all to obtain as many "profiles" as we would like.

See Task Properties in Curation Tasks for details of how properties are resolved in task code.

Task Parameters

New in DSpace 7, you can pass parameters to a task at invocation time. These runtime parameters will be presented to the task as if they were task properties (see above) and, if present, will override the value of identically-named properties. Example:

Task parameters

```
bin/dspace curate -t reticulate -i 123456789/36 -p foreground=red -p background=green
```

Scripted Tasks

The procedure to set up curation tasks in Jython is described on a separate page: Curation tasks in Jython

DSpace 1.8 includes limited (and somewhat experimental) support for deploying and running tasks written in languages other than Java. Since version 6, Java has provided a standard way (API) to invoke so-called scripting or dynamic language code that runs on the java virtual machine (JVM). Scripted tasks are those written in a language accessible from this API. The exact number of supported languages will vary over time, and the degree of maturity of each language, or suitability of the language for curation tasks will also vary significantly. However, preliminary work indicates that Ruby (using the JRuby runtime) and Groovy may prove viable task languages.

Support for scripted tasks does **not** include any DSpace pre-installation of the scripting language itself - this must be done according to the instructions provided by the language maintainers, and typically only requires a few additional jars on the DSpace classpath. Once one or more languages have been installed into the DSpace deployment, task support is fairly straightforward. One new property must be defined in `[dspace]/config/modules/curate.cfg`:

```
curate.script.dir = ${dspace.dir}/scripts
```

This merely defines the directory location (usually relative to the deployment base) where task script files should be kept. This directory will contain a "catalog" of scripted tasks named `task.catalog` that contains information needed to run scripted tasks. Each task has a 'descriptor' property with value syntax:

`<engine>|<relFilePath>|<implClassCtor>`

An example property for a link checking task written in Ruby might be:

```
linkchecker = ruby|rubytask.rb|LinkChecker.new
```

This descriptor means that a "`ruby`" script engine will be created, a script file named "`rubytask.rb`" in the directory `<script.dir>` will be loaded and the resolver will expect an evaluation of "`LinkChecker.new`" will provide a correct implementation object. Note that the task must be configured in all other ways just like java tasks (in `ui.tasknames, ui.taskgroups`, etc).

Script files may embed their descriptors to facilitate deployment. To accomplish this, a script must include the descriptor string with syntax: `$td=<descriptor>` somewhere on a comment line. For example:

```
# My descriptor $td=ruby|rubytask.rb|LinkChecker.new
```

For reasons of portability, the <relFilePath> component may be omitted in this context. Thus, "`$td=ruby||LinkChecker.new`" will be expanded to a descriptor with the name of the embedding file.