Curation Task Cookbook

Curation Task Cookbook

Notes on design patterns, best practices, pitfalls, workarounds, hacks, etc when working with curation tasks.
Feel free to add your own experience and insights, or comment on existing writeups.

Easy tasks *can* and *should* have simple implementations

A large variety of tasks perform comparatively simple operations on DSpace items. An example: check for the presence of a metadata field (say 'dc.publisher') in each item. The only real responsibility a task has is to return a meaningful status to the calling framework - that's **all** there is to it.

So if we extend AbstractCurationTask, our code (beyond the boilerplate needed to define the class etc) can consist entirely of a single method body, which in this case can be a single line of code:

```
public class SimpleTask extends AbstractCurationTask {
    @Override
    public int perform(DSpaceObject dso) throws IOException {
        // your implementation
        return (dso instanceof Item && ((Item)dso).getMetadata("dc.publisher").length() == 0) ? CURATE_FAIL :
CURATE_SUCCESS;
    }
}
```

It's not pretty, but emphasizes how lean you can make tasks, especially if they are for internal consumption. With code this lightweight, it becomes much easier to imagine 'throwaway' tasks that you might install and run only a few times, or modify as the needs change. There are certainly many ways to make tasks more useful, intelligible, general, shareable, etc. but there is nothing wrong with starting small.

The framework often *does* the right thing with collections and communities

Without any effort on your (the task author's) part, the curation framework manages collections and communities (known as *containers*) on your behalf. Here is how it works:

If you invoke a task where the DSpaceObject is an item, the framework only performs the task on that item.
'Invoking a task' can mean from a command-line, in the admin UI, in workflow, or via the API call 'curator.curate(dso)'.

If you invoke a task where the DSpaceObject is a collection (or other container), the framework first performs the task on the collection itself, then on each of the members (items) of the collection. If a community, it first invokes on the community, then each sub-community, then all the collections, and their respective member items. It thus intuitively distributes the task performance through containers: your task code does not need to detect and iterate.

The framework often does *not* do the right thing with collections and communities

Automatic container iteration is handy to be sure, but there are quite obvious issues with it in many contexts of use. To use our metadata checking example above, which is *diagnostic* in nature, if we invoke the task in the administrative UI against a collection of 3 items, in which the 1st and 3rd items have the publisher field, but the 2nd does not, we will never discover that fact, since the UI will display only the status of the *last* invocation of the sequence (viz. the 3rd item). In a container invocation, there is no accumulation of status codes: only the last one appears to the user.

Fortunately, to paraphrase Apple, 'there's an @Ann (annotation) for that'. If we simply add the @Suspendable annotation to our class definition:

```
@Suspendable
public class SimpleTask extends AbstractCurationTask {
    @Override
    public int perform(DSpaceObject dso) throws IOException {
....
```

then whenever the task returns with a non-success status code, the framework halts (*suspends*) the container iteration, and returns that status to the user. In our imagined case, the task invocation would halt on the second item, and we could see the 'FAIL' status in the UI.

But this approach has consequences that need to be understood: we may now have the behavior we want in the admin UI, but may have disrupted another context of use. If we are also performing this task as a command-line job, we may **not** want to suspend operation in the face of one failure (because, perhaps, we are generating a report of all occurrences). If this is the case, we need to 'decorate' the annotation just a bit:

```
@Suspendable(invoked=Curator.Invoked.INTERACTIVE)
```

This instructs the framework to suspend *only* when the task is invoked interactively. Currently, that flag is set when tasks are invoked in the admin UI, but if you have another interactive use-case, it is a standard part of the curation API to set the invocation mode. But remember, if you know that the task will only be used in the admin UI, go with the simplest solution that works - @Suspendable

Tasks must accomodate the fact that workflow items are a little different

One very useful feature of curation is the ability to attach tasks to steps in workflow. But task authors must keep in mind that items in workflow are a little different than items that have been installed in the repository, and avoid the pitfalls that can arise. For instance, suppose we are doing an ordinary *reporting* task which sets a result string:

```
...
String state = (status == SUCCESS) ? "healthy" : "infected";
setResult("Item: " + item.getHandle() + " found to be " + state);
return status;
```

This very innocuous-looking code will fail if invoked in a workflow context, for the simple reason that items are not assigned handles until they exit workflow, so item.getHandle() returns null. Other assumptions about certain system-assigned item metadata are equally vulnerable. A good practice here - if the task is intended to be run in workflow **and** beyond - would be explicitly to test:

```
String itemId = item.getHandle();
if (itemId == null) {
    itemId = "Workflow item: " + item.getID();
    // or use title, etc
}
```

Use task names to profile your task

Every task that can be invoked is identified with a name (usually referred to as the *logical task name* or simply *task name*) that is used to wire the task into all the contexts in which it is used. The name choice is entirely a local deployment decision, and task names should in general be mnemonically helpful, site-unique, and short: one single word. The task name is used in many places:

- to look up the implementing class (via PluginManager)
- to identify tasks used in workflow (in **workflow-curation.xml**)
- to select tasks to run on the command-line

to name just a few. In general, though, it is not visible in *public* places (like the admin UI) directly - where one is given (via configuration properties) a more descriptive (and possibly language-localized) equivalent. One place it *is* visible, however, is within the task code itself, where it is passed as the 'taskId' parameter to the init() method.

This fact can be exploited to address a common pattern in task authoring. Frequently it is the case that one wants a set of closely related functions that operate as essentially different *profiles*. For example, task A may examine whether a given item's metadata field conforms to a specific value or pattern, whereas task B may actually *change* the field to normalize it. There is obviously a lot of common logic in these 2 tasks, so an implementation would likely try to factor out the common code into either a base class or helper class that each task class uses or inherits. But this often leaves the task class very sparse: intuitively, we really want the ability to determine which profile we are using *within* a single task class:

```
if ("report".equals(profile)) {
    // just set the task status and result
    setStatus(...
    setResult(...
} else if ("update".equals(profile)) {
    // update object
    dso.update();
    setStatus(..
    setResult("Updated object:...
}
```

And in fact, that profile data can easily be encoded in the task name itself! Let's see how this could work. First let us define (in **curate.cfg**) two distinct aliases or profiles for our task:

```
plugin.named.org.dspace.curate.CurationTask = \
    org.dspace.curate.ProfiledTask = profile-report, \
    org.dspace.curate.ProfiledTask = profile-update, \
    ....
```

Then within the implementation class *ProfiledTask*, we simply key off the task name, which encodes the profile semantics:

```
if (taskId.endsWith("report")) {
    // just set the task status and result
    setStatus(...
    setResult(...
} else if (taskId.endsWith("update")) {
    // update object
    dso.update();
    setStatus(..
    setResult("Updated object:...
}
```

This small convention can help avert a lot of boiler-plate coding. Of course, if you intend to redistribute your task, be sure to document this behavior.

Extend AbstractCurationTask and use Task Properties for Portability

There's rarely a good reason to have your task directly implement the **org.dspace.curate.CurationTask** interface: extend **org.dspace.curate.AbstractCurationTask** instead. It is extremely lightweight, and offers many handy methods that encode common operations on tasks, even if you do not use the logic it provides for distributing tasks through containers. One great example (available only in DSpace 1.8) is the *taskProperty* family of methods. Here is how it works: tasks frequently rely on configuration properties to determine how they will operate. One would expect the code to manage this to look like:

```
String expectedVersion = ConfigurationManager.getProperty("versions", "version.required");
```

This code would look for a file in /dspace/config/modules with the name 'versions.cfg' and read it's properties, returning a value for the property named 'version.required', but there are several problems with this approach, perhaps the most serious of which is that your task *hard-codes* the name of the configuration file. But this name could conflict with a pre-existing configuration file from some other task, and your task becomes uninstallable/non-portable. AbstractCurationTask gives you a more attractive alternative:

```
String expectedVersion = taskProperty("version.required");
```

This method uses a flexible/portable means of resolving the configuration property file: it uses the **task name** as the config file name. Thus if you have named the task in question 'myTask', the method will look for /dspace/config/modules/myTask.cfg. And remember that the task name is a locally assigned (site-specific) value, which means that name collisions can always be avoided. There is also a companion set of methods for convenient invocation:

- taskIntProperty(String name, int defaultValue)
- taskBooleanProperty(String name, boolean defaultValue)
- taskLongProperty(String name, long defaultValue)

Task properties have many other useful features (explored in later posts), including profiling, property overriding, etc. And remember that you can mix invocations freely: use ConfigurationManager(module, name) if the property in question is in a well-known or pre-establised location, and use taskProperty (name) for the task's specific properties.

Understanding Mutative Tasks (Part 1)

In the parlance of the curation task framework, a task is said to be *mutative* if it can change the state of (mutate) the object passed to the perform() method (or possibly multiple objects, if the passed object is a container). This does not mean, by the way, that it always **will** change an object: it may skip some objects, but not others, or operate only on collections, but not items, etc. In DSpace content API terms, when it does mutate an object, it generally means that an update() method is called on it. When your task is mutative in this sense, it is considered good practice to annotate the implementation class with @Mutative - this informs the framework that your task has that capability. Currently (as of DSpace 1.8), the framework makes no direct use of this annotation, but it opens the door to several interesting possibilities in the future (e.g. the framework could sequence tasks in an order that understands what changes to make first). However, as a *consumer* of tasks written by others, it is an invaluable piece of information that the task 'wears on its sleeve': you might very well scrutinize much more carefully a task that can change your content than one that cannot. So try to respect other task users, and remember to use the @Mutative annotation when appropriate.

Understanding Distributive Tasks (Part 1)

We have observed in several notes that the curation framework will *naturally* act as a container iterator, in the sense that if the DSpace object passed to a perform() method is a collection, community (or *site*, as of DSpace 1.8), the framework will invoke the task on each member of the container. We also saw that there were ways to alter or condition this behavior using the @Suspendable annotation; but there is an even more 'radical' approach that tasks can take: they can inform the curation framework that they want to **entirely opt out of container iteration**, and the way to do this is to annotate the task class with @Distributive. This annotation essentially tells the framework: "Look, I'm going to worry about how (or even whether) to 'distribute' myself in containers, you just butt out". As a result, the most fundamental meaning of the @Distributive annotation is this: it guarantees that the framework will call 'perform(dso)' *exactly once*, whether the passed dso is an item, collection, community, etc. The annotation does *not*, however, make any guarantee that the task will in fact be distributed - that becomes entirely the responsibility of the task implementation. In subsequent notes we will explore why this is a useful capability (and it is for a variety of reasons), but here we will merely note that **AbstractCurationTask** provides a very convenient set of helper methods if your task does *in fact* want to manage its own container iteration. Here is the design pattern:

```
// perform any set up
distribute(dso);  // method supplied by AbstractCurationTask
// perform any follow-up and set status, result, etc
```

All the task has to do is override a method that 'distribute' calls for each Item:

```
void performItem(Item item)
```

The practical effect of all this is that the task doesn't need to worry about iteration details, it merely needs to specify how each Item is handled.

Curation Context Management - Object Cache Issues

A common 'gotcha' in the DSpace programming model arises when a given *Context* (which is the object that provides access to the DB, inter alia) is used in a long-running process. Each time a DSpace object is requested, a copy of it is placed in a memory cache managed by that context. If there are enough objects requested, the cache will consume all the memory available in the Java virtual machine, and ugly out-of-memory crashes ensue. If you want to write a task that might be used in this way (i.e. that might be invoked against all items in the repository or a very large collection), there are various means of preventing disaster. Perhaps the easiest is simply to include in your perform() code:

```
if (dso instanceof Item)
{
    Item item = (Item)dso;
    // do some work with this item
    item.decache();
}
```

The decache() call removes the item from the cache, so it never grows significantly. However, this approach has some downsides. First, it might simply be unnecessary in most cases - you might mostly run the task against collections that never reach a problematic cache size. Second, it might introduce significant inefficiencies. Suppose your task is run together with several others, each of which use the same 'decache' strategy. Then when task1 empties the cache, task2 must reread from the database (since it's now out of cache), and same with task3, etc.

There is a better way as of DSpace 1.8, however. A new API method is available on the *Curator* object called 'setCacheLimit(int limit)' that will instruct the framework to empty the cache if it exceeds the passed limit. Thus, if the caller knows that this will be a **big run**, she simply sets the limit:

```
Curator curator = new Curator();
curator.addTask("task1").addTask("task2").setCacheLimit(1000);
curator.perform(context, id);
```

Since often long-running tasks are relegated to command-line execution, the CurationCli tool exposes a new switch '-l' (with number or objects as argument) that calls the 'setCacheLimit()' method before invoking the tasks.

A few important caveats/restrictions to note in using this approach:

- The context must be *visible* to the curation framework in order for the method to have any effect. In other words, the caller **must** use the signature 'perform(context, id)' **not** 'perform(dso)'
- One of the tasks being performed must not be @Distributive (if they all are, the framework does not *see* the iteration) - these tasks manage their own context caches

Curation Context Management - Transaction Issues

A closely related issue occurs with database transactions. The Context object mentioned above also manages the scope of any database transactions. Typically, we don't need to worry about when the 'commit' occurs, since the curation operation is part of a larger thread of execution. But if a task that updates it's objects participates in a very long-running process, there can occur a situation where the transaction becomes too large. And this is usually quite unneeded: there is no transactional relationship between each object update (they are unrelated). For this reason, the curation system (as of DSpace 1.8) offers an API method on the *Curator* object to manage transactional scope. It is 'setTransactionScope(TxScope scope)' and the values of scope are:

- open - this is the default: it means that no commits will be issued by the framework
- object - this means that after each 'perform()' is called, the framework will call 'context.commit()'.
- curation - this means that after the entire curation (iterating perhaps over a container, and involving possibly multiple tasks), the framework will commit the transaction

As with the cache management, there is a new switch in CurationCli that allows you to set a scope for a command-line task invocation. It is:
'-s <scope>' where scope can be one of "open", "object" or "curation". This will cause setTransactionScope(scope) to be called on the Curation object.

Note that the same caveats/restrictions apply as for the context cache method:

- The context must be *visible* to the curation framework in order for the method to have any effect. In other words, the caller **must** use the signature 'perform(context, id)' **not** 'perform(dso)'
- One of the tasks being performed must not be @Distributive (if they all are, the framework does not *see* the iteration) - these tasks manage their own context caches