Curation Tasks

This documentation provides a guide for how to programmatically create Curation Tasks. For more information configuring Curation Tasks, see the Curation System section of the documentation

Writing your own tasks

A task is just a java class that can contain arbitrary code, but it must have 2 properties:

First, it must provide a no argument constructor, so it can be loaded by the PluginManager. Thus, all tasks are 'named' plugins, with the taskname being the plugin name.

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 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 is returned to CS by any of a task's `perform` methods. The complete list of values, defined in `Curator`, is:

value	symbol	meaning
-3	CURATE_NOTASK	CS could not find the requested task
-2	CURATE_UNSET	task did not return a status code because it has not yet run
-1	CURATE_ERROR	task could not be performed
0	CURATE_SUCCESS	task performed successfully
1	CURATE_FAIL	task performed, but failed
2	CURATE_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 in Curation System) for display.

Result String

The task may set a string indicating details of the outcome:

```
curator.setResult("Item " + item.getID() + " was painted " + color);
```

CS does not interpret or assign result strings; the task does it. A task may choose not to assign a result, but the "best practice" for tasks is to assign one whenever possible. Code which invokes `Curator.getResult()` may use the result string for display or any other purpose.

Reporting Stream

For very fine-grained information, a task may write to a *reporting* stream. Unlike the result string, there is no limit to the amount of data that may be pushed to this stream.

```
curator.report("Lorem ipsum dolor sit amet,\n");
curator.report("consectetur adipiscing elit,\n");
curator.report("sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.\n");
```

Accessing task output in calling code

The status code, reporting stream, and the result string are accessed (or set) by methods on the Curator object:

```
Curator curator = new Curator();
curator.setReporter(new OutputStreamWriter(System.out));
curator.addTask("vscan").curate(coll);
int status = curator.getStatus("vscan");
String result = curator.getResult("vscan");
```

Task Properties

Task code may configure itself using ConfigurationService in the normal manner, or by the use of "task properties". See Curation System - Task Properties for discussion of the issues for which task properties were invented. Any code which extends AbstractCurationTask has access to its configured task properties.

The entire "API" for task properties is:

```
public String taskProperty(String name);
public int taskIntProperty(String name, int defaultValue);
public long taskLongProperty(String name, long defaultValue);
public boolean taskBooleanProperty(String name, boolean default);
```

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.

Scripted Tasks

DSpace 1.8 introduced 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. See Curation System - Scripted Tasks for information on configuring and running scripted tasks.

Interface

Scripted tasks must implement a slightly different interface than the **CurationTask** interface used for Java tasks. The appropriate interface for scripting tasks is **ScriptedTask** and has the following methods:

```
public void init(Curator curator, String taskId) throws IOException;
public int performDso(DSpaceObject dso) throws IOException;
public int performId(Context ctx, String id) throws IOException;
```

The difference is that `ScriptedTask` has separate `perform` methods for DSO and identifier. The reason for that is that some scripting languages (e.g. Ruby) don't support method overloading.

performDso() vs. performId()

You may have noticed that the `ScriptedTask` interface has both `performDso()` and `performId()` methods, but only performDso is ever called when curator is launched from command line.

There are a class of use-cases in which we want to construct or create new DSOs (DSpaceObject) given an identifier in a task. In these cases, there may be no live DSO to pass to the task.
You actually **can** get curation system to call `performId()` if you queue a task then process the queue - when reading the queue all CLI has is the handle to pass to the task.