Curation System

As of release 1.7, 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.

1 Changes in 1.8
2 Tasks
3 Activation
4 Writing your own tasks
5 Task Invocation
5.1 On the command line
5.2 In the admin UI
5.3 In workflow
5.4 In arbitrary user code
6 Asynchronous (Deferred) Operation
7 Task Output and Reporting
7.1 Status Code
7.2 Result String
7.3 Reporting Stream
8 Task Properties
9 Task Annotations
10 Scripted Tasks
10.1 Interface
10.1.1 performDso() vs. performId()
11 Bundled Tasks
11.1 MetadataWebService Task
11.1.1 ISSN to Publisher Name
11.1.2 HTTP Headers
11.1.3 Transformations
11.1.4 Result String Programatic Use
11.1.5 Limits and Use
11.2 NoOp Curation Task
11.3 Bitstream Format Profiler
11.4 Required Metadata
11.5 Virus Scan
11.5.1 Setup the service from the ClamAV documentation.
11.5.2 DSpace Configuration
11.5.3 Task Operation from the Administrative user interface
11.5.4 Task Operation from the Item Submission user interface
11.5.5 Task Operation from the curation command line client
11.5.5.1 Table 1 – Virus Scan Results Table
11.6 Link Checkers
11.6.1 Basic Link Checker
11.6.2 Metadata Value Link Checker
11.7 Microsoft Translator
11.7.1 Configure Microsoft Translator

Changes in 1.8

- *New package:* The default curation task package is now **org.dspace.ctask**. The tasks supplied with DSpace releases are now under **org.dspace.ctask.general**
- *New tasks in DSpace release:* Some additional curation tasks have been supplied with DSpace 1.8, including a link checker and a translator
- *UI task groups:* Ability to assign tasks to groups whose members display together in the Administrative UI
- *Task properties:* Support for a site-portable system for configuration and profiling of tasks using configuration files
- *New framework services:* Support for context management during curation operations
- *Scripted tasks:* New (experimental) support for authoring and executing tasks in languages other than Java

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:

```
plugin.named.org.dspace.curate.CurationTask = \
org.dspace.ctask.general.NoOpCurationTask = noop, \
org.dspace.ctask.general.ProfileFormats = profileformats, \
org.dspace.ctask.general.RequiredMetadata = requiredmetadata, \
org.dspace.ctask.general.ClamScan = vscan, \
org.dspace.ctask.general.MicrosoftTranslator = translate, \
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`

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 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 arguments:

```
-t taskname: name of task to perform
-T filename: name of file containing list of tasknames
-e epersonID: (email address) will be superuser if unspecified
-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 - emit reporting to standard out
```

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`:

```
ui.tasknames = \
     profileformats = Profile Bitstream Formats, \
     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):

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

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
ui.taskgroups = \
  replication = Backup and Restoration Tasks, \
  integrity = Metadata Integrity Tasks, \
  .....
# each group membership list is a separate property, whose value is comma-separated list of logical task names
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="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>
</tasksets>
```

This markup would cause a virus scan to occur during step one of workflow for any collection, 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: `[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="step1" 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.

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 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 is sent to standard out, so is only available when running a task from the command line. Unlike the result string, there is no limit to the amount of data that may be pushed to this stream.

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

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

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`. And thus in the implementation one would find:

```
host = ConfigurationManager.getProperty("clamav", "service.host");
```

and similar. 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 this method provided by AbstractCurationTask in their task implementation code (e.g. in virus scanner):

```
host = taskProperty("service.host");
```

Note that there is no name of the configuration file even mentioned, just the property name whose value we want. At runtime, the curation system **resolves** this call to a configuration file, and it uses the *name the task has been configured as* as the name of the config file. So, for example, if both were installed (in `curate.cfg`) as:

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

then "`taskProperty()`" will resolve to `[dspace]/config/modules/vscan.cfg` when called from ClamAv task, but `[dspace]/config/modules/virusscan.cfg` 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.

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);
```

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 `[dspace]/config/modules/thumbnail.cfg`:

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

Then, following the pattern above, the thumbnail generating task code would look like:

```
if (taskBooleanProperty("forceupdate")) {
    // do something
}
```

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 called: `thumbnail.force.cfg`. In this file, we put only **one** property:

```
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 reads the config file thumbnail.cfg and operates in "non-force" profile (since the value is false), but when we run the task "`thumbnail.force`" the curation system **first** reads `thumbnail.cfg`, **then** reads `thumbnail.force.cfg` which **overrides** the value of the "`forceupdate`" property. 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.

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

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`:

```
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.

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.

Bundled Tasks

DSpace bundles a small number of tasks of general applicability. Those that do not require configuration (or have usable default values) are activated to demonstrate the use of the curation system. They may be removed (deactivated by means of configuration) if desired without affecting system integrity. Those that require configuration may be enabled (activated) by means editing DSpace configuration files. Each task - current as of DSpace 4.0 - is briefly described below.

MetadataWebService Task

DSpace item metadata can contain any number of identifiers or other field values that participate in networked information systems. For example, an item may include a DOI which is a controlled identifier in the DOI registry. Many web services exist to leverage these values, by using them as 'keys' to retrieve other useful data. In the DOI case for example, CrossRef provides many services that given a DOI will return author lists, citations, etc. The MetadataWebService task enables the use of such services, and allows you to obtain and (optionally) add to DSpace metadata the results of any web service call to any service provider. You simply need to describe what service you want to call, and what to do with the results. Using the task code, you can create as many distinct tasks as you have services you want to call. Each description lives in a configuration file in 'config/modules', and is a simple properties file, like all other DSpace configuration files. The name of the configuration file is the task name you assign to it in config/modules/curate.cfg. There are a few required properties you must configure for any service, and for certain services, a few additional ones. An example will illustrate best.

ISSN to Publisher Name

Suppose items (holding journal articles) include 'dc.identifier.issn' when available. We might also want to catalog the publisher name (in 'dc.publisher'). The cataloger could look up the name given the ISSN in various sources, but this 'research' is tedious, costly and error-prone. There are many good quality, free web services that can furnish this information. So we will configure a MetadataWebService task to call a service, and then automatically assign the publisher name to the item metadata. As noted above, all that is needed is a description of the service, and what to do with the results. Create a new file in 'config/modules' called 'issn2pubname.cfg' (or whatever is mnemonically useful to you). The first property in this file describes the service in a 'template'. The template is just the URL to call the web service, with parameters to substitute values in. Here we will use the 'Sherpa/Romeo' service:

```
template=http://www.sherpa.ac.uk/romeo/api29.php?issn={dc.identifier.issn}
```

When the task runs, it will replace '{dc.identifier.issn}' with the value of that field in the item, If the field has multiple values, the first one will be used. As a web service, the call to the above URL will return an XML document containing information (including the publisher name) about that ISSN. We need to describe what to do with this response document, i.e. what elements we want to extract, and what to do with the extracted content. This description is encoded in a property called the 'datamap'. Using the example service above we might have:

```
datamap=//publisher/name=>dc.publisher,//romeocolor
```

Each separate instruction is separated by a comma, so there are 2 instructions in this map. The first instruction essentially says: find the XML element 'publisher name' and assign the value or values of this element to the 'dc.publisher' field of the item. The second instruction says: find the XML element 'romeocolor', but do not add it to the DSpace item metadata - simply add it to the task result string (so that it can be seen by the person running the task). You can have as many instructions as you like in a datamap, which means that you can retrieve multiple values from a single web service call. A little more formally, each instruction consists of one to three parts. The first (mandatory) part identifies the desired data in the response document. The syntax (here '//publisher/name') is an XPath 1.0 expression, which is the standard language for navigating XML trees. If the value is to be assigned to the DSpace item metadata, then 2 other parts are needed. The first is the 'mapping symbol' (here '=>'), which is used to determine how the assignment should be made. There are 3 possible mapping symbols, shown here with their meanings:

```
'->' mapping will add to any existing value(s) in the item field
'=>' mapping will replace any existing value(s) in the item field
'~>' mapping will add *only if* item field has no existing value(s)
```

The third part (here 'dc.publisher') is simply the name of the metadata field to be updated. These two mandatory properties (template and datamap) are sufficient to describe a large number of web services. All that is required to enable this task is to edit 'config/modules/curate.cfg', add 'issn2pubname' to the list of tasks:

```
plugin.named.org.dspace.curate.CurationTask = \
... other defined tasks
org.dspace.ctask.general.MetadataWebService = issn2pubname, \
... other metadatata web service tasks
org.dspace.ctask.general.MetadataWebService = doi2crossref, \
```

If you wish the task to be available in the Admin UI, see the Invocation from the Admin UI documentation (above) about how to configure it. The remaining sections describe some more specialized needs using the MetadataWebService task.

HTTP Headers

For some web services, protocol and other information is expressed not in the service URL, but in HTTP headers. Examples might be HTTP basic auth tokens, or requests for a particular media type response. In these cases, simply add a property to the configuration file (our example was 'issn2pubname.cfg') containing all headers you wish to transmit to the service:

```
headers=Accept: application/xml||Cache-Control: no-cache
```

You can specify any number of headers, just separate them with a 'double-pipe' ('||').

Transformations

One potential problem with the simple parameter substitutions performed by the task is that the service might expect a different format or expression of a value than the way it is stored in the item metadata. For example, a DOI service might expect a bare prefix/suffix notation ('10.000/12345'), whereas the DSpace metadata field might have a URI representation ('http://dx.doi.org/10.000/12345'). In these cases one can declare a 'transformation' of a value in the template. For example:

```
template=http://www.crossref.org/openurl/?id={doi:dc.relation.isversionof}&format=unixref
```

The 'doi:' prepended to the metadata field name declares that the value of the 'dc.relation.isversionof' field should be *transformed* before the substitution into the template using a transformation named 'doi'. The transformation is itself defined in the same configuration file as follows:

```
transform.doi=match 10. trunc 60
```

This would be read as: exclude the value string up to the occurrence of '10.', then truncate any characters after length 60. You may define as many transformations as you want in any task, although generally 1 or 2 will suffice. They keywords 'match', 'trunc', etc are names of 'functions' to be applied (in the order entered). The currently available functions are:

```
'cut' <number> = remove number leading characters
'trunc' <number> = remove trailing characters after number length
'match' <pattern> = start match at pattern
'text' <characters> = append literal characters (enclose in ' ' when whitespace needed)
```

When the task is run, if the transformation results in an invalid state (e.g. cutting more characters than there are in the value), the un-transformed value will be used and the condition will be logged. Transformations may also be applied to values returned from the web service. That is, one can apply the transformation to a value before assigning it to a metadata field. In this case, the declaration occurs in the datamap property, not the template:

```
datamap=//publisher/name=>shorten:dc.publisher,//romeocolor
```

Here the task will apply the 'shorten' transformation (which must be defined in the same config file) before assigning the value to 'dc.publisher'.

Result String Programatic Use

Normally a task result string appears in a window in the admin UI after it has been invoked. The MedataWebService task will concatenate all the values declared in the 'datamap' property and place them in the result string using the format: 'name:value name:value' for as many values as declared. In the example above we would get a string like 'publisher: Nature romeocolor: green'. This format is fine for simple display purposes, but can be tricky if the values contain spaces. You can override the space separator using an optional property 'separator' (put in the config file, with all other properties). If you use:

```
separator=||
```

for example, it becomes easy to parse the result string and preserve spaces in the values. This use of the result string can be very powerful, since you are essentially creating a map of returned values, which can then be used to populate a user interface, or any other way you wish to exploit the data (drive a workflow, etc).

Limits and Use

A few limitations should be noted. FIrst, since the response parsing utilizes XPath, the service can only operate on XML, (not JSON) response documents. Most web services can provide either, so this should not be a major obstacle. The MetadataWebService can be used in many ways: showing an admin a value in the result string in a UI, run in a batch to update a set of items, etc. One excellent configuration is to wire these tasks into submission workflow, so that 'automatic cataloging' of many fields can be performed on ingest.

NoOp Curation Task

This task does absolutely nothing. It is intended as a starting point for developers and administrators wishing to learn more about the curation system.

Bitstream Format Profiler

The task with the taskname 'formatprofiler' (in the admin UI it is labeled "Profile Bitstream Formats") examines all the bitstreams in an item and produces a table ("profile") which is assigned to the result string. It is activated by default, and is configured to display in the administrative UI. The result string has the layout:

```
10 (K) Portable Network Graphics
5  (S) Plain Text
```

where the left column is the count of bitstreams of the named format and the letter in parentheses is an abbreviation of the repository-assigned support level for that format:

```
U  Unsupported
K  Known
S  Supported
```

The profiler will operate on any DSpace object. If the object is an item, then only that item's bitstreams are profiled; if a collection, all the bitstreams of all the items; if a community, all the items of all the collections of the community.

Required Metadata

The "`requiredmetadata`" task examines item metadata and determines whether fields that the web submission (`input-forms.xml`) marks as required are present. It sets the result string to indicate either that all required fields are present, or constructs a list of metadata elements that are required but missing. When the task is performed on an item, it will display the result for that item. When performed on a collection or community, the task be performed on each item, and will display the *last* item result. If all items in the community or collection have all required fields, that will be the last in the collection. If the task fails for any item (i.e. the item lacks all required fields), the process is halted. This way the results for the 'failed' items are not lost.

Virus Scan

The "`vscan`" task performs a virus scan on the bitstreams of items using the ClamAV software product.
Clam AntiVirus is an open source (GPL) anti-virus toolkit for UNIX. A port for Windows is also available. The virus scanning curation task interacts with the ClamAV virus scanning service to scan the bitstreams contained in items, reporting on infection(s). Like other curation tasks, it can be run against a container or item, in the GUI or from the command line. It should be installed according to the documentation at http://www.clamav.net. It should not be installed in the dspace installation directory. You may install it on the same machine as your dspace installation, or on another machine which has been configured properly.

Setup the service from the ClamAV documentation.

This plugin requires a ClamAV daemon installed and configured for TCP sockets. Instructions for installing ClamAV (http://www.clamav.net/doc/install.html)

NOTICE: The following directions assume there is a properly installed and configured clamav daemon. Refer to links above for more information about ClamAV.
The Clam anti-virus database must be updated regularly to maintain the most current level of anti-virus protection. Please refer to the ClamAV documentation for instructions about maintaining the anti-virus database.

DSpace Configuration

In `[dspace]/config/modules/curate.cfg`, activate the task:

- Add the plugin to the comma separated list of curation tasks.

```
### Task Class implementations
plugin.named.org.dspace.curate.CurationTask = \
org.dspace.ctask.general.ProfileFormats = profileformats, \
org.dspace.ctask.general.RequiredMetadata = requiredmetadata, \
org.dspace.ctask.general.ClamScan = vscan
```

- Optionally, add the vscan friendly name to the configuration to enable it in the administrative it in the administrative user interface.

```
ui.tasknames = \
profileformats = Profile Bitstream Formats, \
requiredmetadata = Check for Required Metadata, \
vscan = Scan for Viruses
```

- In `[dspace]/config/modules`, edit configuration file clamav.cfg:

```
service.host = 127.0.0.1
Change if not running on the same host as your DSpace installation.
service.port = 3310
Change if not using standard ClamAV port
socket.timeout = 120
Change if longer timeout needed
scan.failfast = false
Change only if items have large numbers of bitstreams
```

- Finally, if desired virus scanning can be enabled as part of the submission process upload file step. In `[dspace]/config/modules`, edit configuration file `submission-curation.cfg`:

```
virus-scan = true
```

Task Operation from the Administrative user interface

Curation tasks can be run against container and item dspace objects by e-persons with administrative privileges. A curation tab will appear in the administrative ui after logging into DSpace:

1. Click on the curation tab.
2. Select the option configured in ui.tasknames above.
3. Select Perform.

Task Operation from the Item Submission user interface

If desired virus scanning can be enabled as part of the submission process upload file step. In `[dspace]/config/modules`, edit configuration file `submission-curation.cfg`:

```
virus-scan = true
```

Task Operation from the curation command line client

To output the results to the console:

```
[dspace]/bin/dspace curate -t vscan -i <handle of container or item dso> -r -
```

Or capture the results in a file:

```
[dspace]/bin/dspace curate -t vscan -i <handle of container or item dso> -r - > /<path...>/<name>
```

Table 1 – Virus Scan Results Table

GUI (Interactive Mode)	FailFast	Expectation
Container	T	Stop on 1st Infected Bitstream
Container	F	Stop on 1st Infected Item
Item	T	Stop on 1st Infected Bitstream
Item	F	Scan all bitstreams
Command Line		
Container	T	Report on 1st infected bitstream within an item/Scan all contained Items
Container	F	Report on all infected bitstreams/Scan all contained Items
Item	T	Report on 1st infected bitstream
Item	F	Report on all infected bitstreams

Link Checkers

Two link checker tasks, BasicLinkChecker and MetadataValueLinkChecker can be used to check for broken or unresolvable links appearing in item metadata.

This task is intended as a prototype / example for developers and administrators who are new to the curation system.

These tasks are not configurable.

Basic Link Checker

BasicLinkChecker iterates over all metadata fields ending in "uri" (eg. dc.relation.uri, dc.identifier.uri, dc.source.uri ...), attempts a GET to the value of the field, and checks for a 200 OK response.
Results are reported in a simple "one row per link" format.

Metadata Value Link Checker

MetadataValueLinkChecker parses all metadata fields for valid HTTP URLs, attempts a GET to those URLs, and checks for a 200 OK response.
Results are reported in a simple "one row per link" format.

Microsoft Translator

Microsoft Translator uses the Microsoft Translate API to translate metadata values from one source language into one or more target languages.
This task cab be configured to process particular fields, and use a default language if no authoritative language for an item can be found. Bing API v2 key is needed.

MicrosoftTranslator extends the more generic AbstractTranslator. This now seems wasteful, but a GoogleTranslator had also been written to extend AbstractTranslator. Unfortunately, Google has announced they are decommissioning free Translate API service, so this task hasn't been included in DSpace's general set of curation tasks.

Translated fields are added in addition to any existing fields, with the target language code in the 'language' column. This means that running a task multiple times over one item with the same configuration could result in duplicate metadata.

This task is intended as a prototype / example for developers and administrators who are new to the curation system.

Configure Microsoft Translator

An example configuration file can be found in [dspace]/config/modules/translator.cfg.

```
#----------------------------------------------------------------#
#----------TRANSLATOR CURATION TASK CONFIGURATIONS--------------#
#----------------------------------------------------------------#
# Configuration properties used solely by MicrosoftTranslator   #
# Curation Task (uses Microsoft Translation API v2)             #
#----------------------------------------------------------------#
## Translation field settings
##
## Authoritative language field
## This will be read to determine the original language an item was submitted in
## Default: dc.language

translate.field.language = dc.language

## Metadata fields you wish to have translated
#
translate.field.targets = dc.description.abstract, dc.title, dc.type

## Translation language settings
##
## If the language field configured in translate.field.language is not present
## in the record, set translate.language.default to a default source language
## or leave blank to use autodetection
#
translate.language.default = en

## Target languages for translation
#
translate.language.targets = de, fr

## Translation API settings
##
## Your Bing API v2 key and/or Google "Simple API Access" Key
## (note to Google users: your v1 API key will not work with Translate v2,
## you will need to visit https://code.google.com/apis/console and activate
## a Simple API Access key)
##
## You do not need to enter a key for both services.
#
translate.api.key.microsoft = YOUR_MICROSOFT_API_KEY_GOES_HERE
translate.api.key.google = YOUR_GOOGLE_API_KEY_GOES_HERE
```