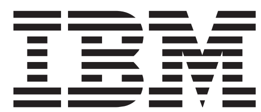
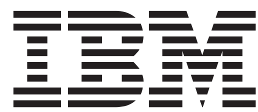
IBM Content Collector
Version 3.0.0.2

Script Connector Implementation Guide



IBM Content Collector
Version 3.0.0.2

Script Connector Implementation Guide



Note

Before using this information and the product it supports, read the information in “Notices” on page 55.

This edition applies to version 3.0 Fix Pack 2 of IBM Content Collector (product number 5724-V57) and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright IBM Corporation 2008, 2012.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

ibm.com and related resources v
How to send your comments vi
Contacting IBM vi

What is the IBM Content Collector Script Connector? 1
Script Connector overview 1
What can you use it for? 1
How does it work? 2
What are the limitations? 2

Getting started 5
Installing the Script Connector 5
The Script Connector template 8
Creating a working folder from the template . . . 8
What happens next? 10

Writing connector scripts 11
The Script Connector API 11
Writing connector scripts in JScript 12
Reading task input parameters 12
Gathering metadata within a script 14
Returning metadata from a task script 14
Submitting metadata from a collector script . . 15
Logging 15
Error handling 16
Updating performance counters 16
Using existing COM objects in script code . . . 16
Upgrading Script Connector scripts from earlier IBM Content Collector versions 18
Testing script code 19
Writing connector scripts in Python 19
Creating tasks 20
Reading task input parameters 21
Gathering metadata within a script 22
Returning metadata from a task script 22
Creating collectors 22
Submitting metadata from a collector script . . 23
Assigning entity IDs 24
Logging 24
Error handling 25
Updating performance counters and Windows event logs 25
Using existing COM objects in script code . . . 26

Preparing for deployment 27
Modifying .wsc files for COM registration 27
Generating new guids for tasks and collectors . . . 28
Creating a registration script. 29
Creating a registration script using JScript . . . 29
Creating a registration script in Python 31
Enabling task and collector configuration in Configuration Manager 33
Customizing configuration screens. 33

Deploying your connector 35
Prerequisite steps 35
Unregistering the existing implementation 35
Deploying the new implementation files. 36
Registering the new implementation 36
After deployment 37

Configuring your connector 39
Connector configuration 39
The Script Connector toolbox 40
Task configuration 40
Collector configuration 42
Troubleshooting configuration issues 43
The Script Connector does not appear in Configuration Manager 43
A new task or collector does not appear in Configuration Manager 43
A task or collector cannot be configured 43
The Script Connector configuration window displays cryptic labels 44

Debugging connector scripts 45

Additional documentation 47
Script Connector API documentation 47
Windows Script Technologies documentation . . . 48

Samples 49
Unix date conversion 49
File processing connector 50
Web service call 51
Python Unix date conversion 52

Notices 55

Index 59

© Copyright IBM Corp. 2008, 2012

ibm.com and related resources

Product support and documentation are available from ibm.com®.

Support and assistance

Product support is available on the web. Simply click Support from the appropriate product website.

IBM® Content Collector
: http://www-01.ibm.com/software/data/content-management/content-collector/

IBM Email Archive and eDiscovery Solution
: http://pic.dhe.ibm.com/infocenter/email/v3r0m0/index.jsp

IBM CommonStore for Exchange Server
: http://www.ibm.com/software/data/commonstore/exchange/

IBM CommonStore for Lotus® Domino®
: http://www.ibm.com/software/data/commonstore/lotus/

IBM Content Manager
: http://www.ibm.com/software/data/cm/cmgr/mp/

IBM FileNet® P8
: http://www.ibm.com/software/data/content-management/filenet-p8-platform/

IBM Enterprise Records
: http://www.ibm.com/software/data/content-management/filenet-records-manager/

IBM Records Manager
: http://www.ibm.com/software/data/cm/cmgr/rm/

IBM WebSphere® Application Server
: http://www.ibm.com/software/webservers/appserv/was/

Lotus Notes® and Domino
: http://www.ibm.com/software/lotus/notesanddomino/

Information center

You can view the IBM Content Collector product documentation in an Eclipse-based information center. See the information center at http://pic.dhe.ibm.com/infocenter/email/v3r0m0/index.jsp.

PDF publications

You can view a PDF version of the IBM Content Collector installation and configuration guide by using the Adobe Acrobat Reader for your operating system. The guide is available from the IBM Publications Center. If you do not have the Acrobat Reader installed, you can download it from the Adobe website at http://www.adobe.com.

© Copyright IBM Corp. 2008, 2012

How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information.

Send your comments by using the online reader comment form at https://www14.software.ibm.com/webapp/iwm/web/signup.do?lang=en_US &source=swg-rcf.

Contacting IBM

To contact IBM customer service in the United States or Canada, call 1-800-IBM-SERV (1-800-426-7378).

To learn about available service options, call one of the following numbers:

- In the United States: 1-888-426-4343
- In Canada: 1-800-465-9600

For more information about how to contact IBM, see the Contact IBM website at http://www.ibm.com/contact/us/.

What is the IBM Content Collector Script Connector?

Script Connector overview

The Script Connector is an IBM Content Collector connector that enables customers and service providers to implement custom tasks and collectors that extend standard IBM Content Collector functionality. This makes the Script Connector very useful for integrating IBM Content Collector with existing business processes, prototyping and proof of concept studies, and for customization "in the field" by implementers and customers.

The Script Connector uses popular scripting languages to provide a straightforward but flexible way of implementing custom collectors and tasks. Using scripting enables rapid delivery of comparatively simple and self-contained functionality. However, the use of scripting also means that the Script Connector is not suitable for all scenarios, especially those that have tightly constrained performance requirements.

What can you use it for?

The Script Connector is frequently used to implement a custom task that provides additional metadata to existing entities in a task route.

For example, a Script Connector task can be implemented to:

- Perform simple data conversion duties based solely on the input data, such as converting integer Unix dates to internal DateTime values.
- Manage complex data conversion operations, such as anonymizing or encrypting personally identifiable information.
- Call web services or perform database queries to provide additional metadata, such as using a customer identifier to retrieve the corresponding customer name and address.
- Use third-party APIs to read extended properties from the file system and from specific file types such as Microsoft Office documents and CAD drawings.
- Handle custom postprocessing operations, such as sending an email message.

A Script Connector task might also add additional entities to the task route. For example, a Script Connector task can be implemented to:

- Extract files from a .zip archive and submit them to the task route, or create a new .zip file packaging files processed by the task route.
- Convert files from one format to another, such as converting .doc files to .rtf files.
- Use an XSL style sheet to render XML files as HTML.

The Script Connector can also be used to implement custom collectors to find and submit items into a task route. You might consider creating a custom collector to:

- Process metadata files that are not in a format that is supported by the File System Connector, such as fixed length records, base64 encoded data, or text files containing name/value pairs.
- Extract files or other content from a third-party system.

© Copyright IBM Corp. 2008, 2012

Implementing a custom collector is optional and is generally more complex than implementing a custom task. If possible, try to use an existing collector to submit items for processing, then use a custom task to fill the gaps in metadata or functionality.

How does it work?

The Script Connector is based on Windows COM technology. To perform work, the Script Connector creates a COM object using information in the Windows registry, then uses COM late binding to call a method named **execute** on that object. To use the Script Connector to provide custom functionality, you must provide a COM component that implements the execute method and provide the registry information to call it.

The Script Connector is specifically intended to work with the scripting languages provided by the Windows Scripting Runtime (JScript and VBScript). To interface with script code, the Script Connector uses Windows Scripting Components (WSC), a bridging technology that allows script code to be registered and called as COM objects. The script code can then leverage other COM components developed in any language, including other scripts that are registered as COM components using WSC. The Script Connector also supports the use of Python scripts registered as COM objects, leveraging the functionality provided by standard and third-party Python libraries.

The Script Connector also provides an API to allow interaction with the IBM Content Collector infrastructure. This allows scripts to retrieve passed configuration and input parameters, to add metadata to existing entities, to submit new files and other entities, to return task status, to log messages, to update performance counters, and to register the script code so that it can be used. The Script Connector also supports the creation and registration of custom metadata sources.

The Script Connector provides a simple plug-in user interface to the IBM Content Collector Configuration Manager to add tasks and collectors to task routes, provide basic task and collector configuration, and to specify task input parameters. The text labels and tooltip help shown to users can be customized by updating a localization file.

Custom connector development using the Script Connector is much simpler than developing custom connectors in C++. In fact, these scripts can be developed with nothing more complex than a standard text editor (although a text editor with syntax highlighting such as Notepad++ is helpful); you do not require a fully featured development environment such as Microsoft Visual Studio.

What are the limitations?

Currently, only one Script Connector can be installed on a system. This one instance can be configured to perform multiple unrelated tasks, but this limitation complicates mixing and matching tasks from different Script Connector implementations. A failure in an unreliable task or collector that causes the Script Connector to crash, will also stop all Script Connector tasks and collectors from working.

Implementing a collector is optional, but the Script Connector can implement a maximum of one collector. However, this collector can be used in multiple task routes and you can configure multiple instances of the collector in a single task

route. The Script Connector can implement multiple tasks, irrespective of whether a collector is implemented. Obviously, it must implement at least one task or a collector to be useful.

The Script Connector user interface components in Configuration Manager are generic because they have to support all possible uses of the Script Connector. The user interface is not user configurable, except for changing the display text and tooltips.

The generic user interface means that collector configuration must be defined as a single string for each collector instance. However, this string can contain any information that can be meaningfully represented as a string. For example, simple configuration parameters can be specified on multiple lines as name/value pairs; a hierarchical data structure can be represented as XML, and binary data can be serialized using base64 encoding. For more complex configuration requirements, the configuration string can also be used to specify an external configuration file, which can use any format you want.

The generic user interface also means that static task configuration is limited to a single string for each task instance. Again, this can contain any information that can be represented as a string. Tasks also support named input parameters that can be used to pass constant values or use an input expression to derive input values from the metadata from upstream collectors and tasks.

The Script Connector does not implement caching and the script object is re-created for each call to a task or collector. This means that you cannot create a Script Connector task that is optimized for high throughput. The lack of caching also means that the Script Connector is not suitable for creating tasks that use APIs that are expensive to initialize (in terms of time, such as connecting to a remote database over a WAN, or in terms of memory, such as loading a large XML DOM object into memory), or have limitations on the creation rate of top level classes (such as the Domino COM API). However, limitations that are not acceptable for tasks, can be acceptable for collectors, where the collector might connect once and run continuously for minutes or even hours.

Getting started

Before you start working with the Script Connector, review the information about prerequisites, installation, and the Script Connector template.

Prerequisites

IBM Content Collector must be installed and configured.

If you want to write connector scripts in Python, download and install the following software:

1. ActiveState ActivePython for Windows (x86) (32-bit)

   The free Community Edition is suitable. The provided Script Connector samples were developed and tested using ActivePython version 2.7.2.
2. Python for Windows extensions (pywin32)

   Ensure that you download the Windows 32-bit version of this software for the specific Python release installed at the previous step.

Installing the Script Connector

Because the majority of IBM Content Collector customers will not need the Script Connector, it is not automatically installed and configured as part of the normal installation process. However, to ensure the Script Connector is readily available for those customers that do need it, the Script Connector installer is copied into the `ctms\ResourceKit` subdirectory of the IBM Content Collector installation directory.

If IBM Content Collector is installed to the default location, you can find the Script Connector package at this location:

On 64-bit Windows, at:

```
C:\Program Files (x86)\IBM\ContentCollector\ctms\ResourceKit\
ScriptConnector
```

On 32-bit Windows, at:

```
C:\Program Files\IBM\ContentCollector\ctms\ResourceKit\ScriptConnector
```

The `ScriptConnector` folder initially contains the following files:

Table 1. Contents of the Script Connector folder before installation

Folder	File	Contents
	`readme.txt`	Installation details and known issues.
`Install`	`Install Script Connector`	Installation shortcut that provides pre-configured installation parameters.

© Copyright IBM Corp. 2008, 2012

Table 1. Contents of the Script Connector folder before installation (continued)

Folder	File	Contents
Install	ScriptConnectorConnectorSetup.msi	Installation package for the Script Connector service, API, and customization kit including template, samples, and API documentation. **Tip:** Do not attempt to install the Script Connector by running this file directly. The installation will fail with an error indicating that the TARGETDIR parameter has not been set. However, after installation, you can uninstall the Script Connector by running this file.

To install the Script Connector:

1. Double-click the `Install Script Connector` shortcut in the `Install` folder. This runs `ScriptConnectorSetup.msi` with the required command line parameters to install the Script Connector executable files and API to the correct location. The `ScriptConnector` folder now contains the following additional files and folders:

Table 2. Contents of the Script Connector folder after installation

Folder	File	Contents
Documentation	ScriptConnectorAPIDocs.zip	Zip archive containing HTML Script Connector API documentation See the Script Connector API documentation for details.
Install\Logs		Installation logs.
Samples		Script Connector samples. See "Unix date conversion" on page 49, "File processing connector" on page 50, "Web service call" on page 51, and "Python Unix date conversion" on page 52 for details.
Template		Script Connector template. See "The Script Connector template" on page 8 for more information.
Tools	GuidGenerator.js	Script for generating new globally unique identifiers (guids). See "Generating new guids for tasks and collectors" on page 28 for more information.

The Script Connector installer also installs the following Python support files into the IBM Content Collector installation directory:

Folder	File	Usage
ctms\ibm	`__init__.py`	Defines the namespace **ibm** to enable importing.
ctms\ibm\ctms	`__init__.py`	Defines the namespace **ibm.ctms** to enable importing. Exposes enumeration types in `ConfigurableConnectorDLL` to Python scripts.
ctms\ibm\ctms	util.py	Contains Python utilities for connector development. Provides these classes: • JavaScript like DateTime class for returning date/time metadata to simplify translation of JScript code to Python. • ACLEntry class for returning ACL entry metadata • Expando object class used for passing extended error information and ACL entry metadata to scripts • Registration helper classes • Enumeration like classes for LogLevels, Win32LogLevels, TypeSystem2TypeIds, and TaskTypes. Enabled by: `import ibm.ctms.util`
ctms\ibm\ctms	template.py	Provides a template for Python tasks and collectors. Because Python supports inheritance, the template provides base classes for both tasks and collectors, encapsulating the common functionality. Enabled by: `import ibm.ctms.template`

After the installation, the Script Connector does not appear in the list of available connectors in Configuration Manager until you register a task or a collector.

2. Complete one of these tasks:
   - Deploy one of the samples.
   - Use the Script Connector template to create a custom implementation, then deploy it.

     All instructions for setting up a Script Connector apply to new Script Connector implementations that are written using the Script Connector template.

After deployment and registration, you can configure task routes that use the Script Connector, and the IBM Content Collector Task Routing Engine service will be able to run them.

If you need to uninstall the Script Connector, double-click `ScriptConnectorSetup.msi`. This gives you the option of repairing or uninstalling the Script Connector. Alternatively, you can use the Windows Control Panel **Program and Features** dialog to uninstall or repair the Script Connector. The Script Connector is listed as **IBM Content Collector Script Connector**.

The Script Connector template

To minimize the work you need to do, the `ConnectorTemplate` folder provides a template for creating a new JScript Script Connector implementation.

This template contains these files:

Table 3. List of template files

File	Usage
`ConnectorConfiguration.js`	JScript template for registering tasks, collectors, and custom metadata.
`ConnectorConfiguration.wsc`	Used to register the Script Connector registration script in `ConnectorConfiguration.js` as a COM object that can be called by the Script Connectorexecutable during registration.

`CustomTask1.js`
`CustomTask2.js`
`CustomCollector.js`
	JScript templates for custom tasks and collectors. These templates also contain useful code snippets that demonstrate how to use the Script Connector API to read configuration and submit entity metadata.

`CustomTask1.wsc`
`CustomTask2.wsc`
`CustomCollector.wsc`
	Used to register task and collector scripts as COM objects that can be called by the Script Connector.

`ConfigurableConnector.adf`
`CustomTask1.adf`
`CustomTask2.adf`
`CustomCollector.adf`
	Adds Script Connector user interface components to Configuration Manager.

`ConfigurableConnector.Resource.en-US.xml`	Provides English language labels and tooltip text for the Script Connector user interface components.

All instructions for setting up a Script Connector apply to new Script Connector implementations that are written using the Script Connector template.

Creating a working folder from the template

Always create a working folder before you start to customize the template scripts.

To set up a working folder:

1. Create a working folder for your script components.

   Important: Do not put the working folder in the `ContentCollector\ctms` folder or any of its subfolders.

2. Copy these files from the template:

Table 4. List of files to be copied to the working folder

Source file	Target location
`Template\ConnectorConfiguration.js`	*Working Folder*
`Template\ConnectorConfiguration.wsc`	*Working Folder*
`Template\adf\ConfigurableConnector.adf`	*Working Folder*`\adf`
`Template\localization\ConfigurableConnector.Resource.en-US.xml`	*Working Folder*`\localization`

3. If you are implementing custom tasks, copy files as follows.

   Tip: When you copy the template files for collectors and tasks, you do not need to rename the template files, but using a unique filename prefix that is specific to each task and collector simplifies combining tasks from multiple sources into a single Script Connector deployment package.

   - For one or more custom tasks:

Table 5. Files required for implementing one or more custom tasks

Source file	Target location
`Template\CustomTask1.js`	*Working Folder*`\`*TaskName*`.js`
`Template\CustomTask1.wsc`	*Working Folder*`\`*TaskName*`.wsc`
`Template\adf\CustomTask1.adf`	*Working Folder*`\adf\`*TaskName*`.adf`

   - For two or more custom tasks:

Table 6. Files required for implementing two or more custom tasks

Source file	Target location
`Template\CustomTask2.js`	*Working Folder*`\`*TaskName2*`.js`
`Template\CustomTask2.wsc`	*Working Folder*`\`*TaskName2*`.wsc`
`Template\adf\CustomTask2.adf`	*Working Folder*`\adf\`*TaskName2*`.adf`

   - For three or more custom tasks, copy the same files as for two or more custom tasks, renaming the files appropriately.

4. If you are implementing a custom collector, copy files as follows:

Table 7. Files required for implementing a custom collector

Source file	Target location
`Template\CustomCollector.js`	*Working Folder*`\`*CollectorName*`.js`
`Template\CustomCollector.wsc`	*Working Folder*`\`*CollectorName*`.wsc`
`Template\adf\CustomCollector.adf`	*Working Folder*`\adf\`*CollectorName*`.adf`

What happens next?

After you have set up your working folder from the Script Connector template, you must customize the files.

Complete these customization steps.

Table 8. Customization tasks

File	What to do
Task scripts	For each task, replace the body of the ProcessEntity() function with your own custom task implementation. See "Writing connector scripts" on page 11 for more information.
Collector scripts	If implementing a collector, replace the body of the PerformCollection() function with your own custom collector implementation. See "Writing connector scripts" on page 11 for more information.
.wsc files	Replace the values of the description, classid (guid), and progid attributes in the registration element. Update the src attribute in the script element to reference the corresponding .js script file. See "Modifying .wsc files for COM registration" on page 27 for more information.
`ConnectorConfiguration.js`	You must provide the body of the registration script. The task and collector guids specified in the script must match classid elements in the corresponding .wsc files. See "Creating a registration script" on page 29 for more information.
`ConnectorConfiguration.wsc`	If you renamed `ConnectorConfiguration.js`, update the src attribute in the script element to reference the renamed .js script file.
`ConfigurableConnector.adf`	Do not change.
Task and collector .adf files	Update with the task IDs of your custom tasks (as registered in `ConnectorConfiguration.js`). See "Enabling task and collector configuration in Configuration Manager" on page 33 for more information.
`ConfigurableConnector.Resource.en-US.xml`	Customize the labels and tooltips displayed to the user in the Script Connector configuration screens in Configuration Manager. See "Customizing configuration screens" on page 33 for more information.

Writing connector scripts

To complete a custom Script Connector implementation, use the Script Connector template and API to write scripts that integrate the required custom task and collector functionality. The functions that are defined in these scripts can be called by the Script Connector and can interact with the CTMS framework. The following information covers writing connector scripts in both JScript and Python.

The Script Connector API

The Script Connector API allows two-way interaction between the Script Connector and the scripts that do the work. Your scripts will use the Script Connector API to read input parameters, to return task output metadata, to submit collected items to log messages, and to update performance counters.

The API defines the following classes:

ICOMCaptureUtilForScripts
This is the main API class. It is used to:

- Test if the connector is shutting down or if the collector is stopping.
- Add task status, file, and other metadata to entities.
- Log events.
- Update performance counters.
- Return task, collector, and custom metadata details for registration.

An instance is passed to the execute() method of task and collector scripts and to the describeMethods() method of registration scripts.

ICOMCollectorSubmitProxyForScripts
This class is only used by collectors. It is used to:

- Associate metadata with entities found by the collector.
- Submit the entity metadata to the task route for processing.

The collector obtains an instance by calling ICOMCaptureUtilForScripts.createCollectorSubmitStubProxy().

This class is new in IBM Content Collector V3.0. In earlier versions, this functionality was handled by ICOMCaptureUtilForScripts.

IComTaskInput
This class is a container for task input. It is used to:

- Pass the configuration string containing static configuration data to tasks and collectors.
- Pass a collection containing the entity IDs and entity data for each entity routed to a task.
- Pass context information on the current task route node (task route and task ids and names).

A COM SafeArray containing instances of this class (usually one) is passed as input to the execute() method of tasks and collectors.

IEntityData
Type-safe property bag used to contain a set of named properties. It is used to:

© Copyright IBM Corp. 2008, 2012

- Pass the evaluated results of input parameter expressions for each entity passed in task inputs.
- Return output metadata for an entity, from either a task or a collector.

New instances can be created in a script:

```
var customMetadata =
  new ActiveXObject("ConfigurableConnectorDLL.EntityData");
```

The API also defines enumerations containing error codes from calls to methods in these classes. For more information, see the Script API documentation. How to access the API documentation is described in "Script Connector API documentation" on page 47.

Writing connector scripts in JScript

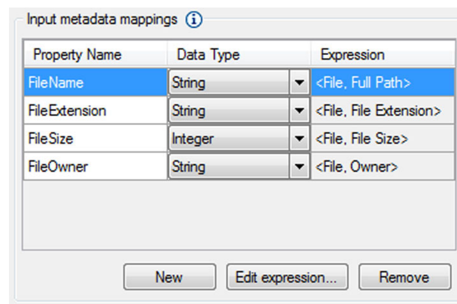
The Script Connector template provides JScript templates for both tasks (`CustomTask`*x*`.js`) and collectors (`CustomCollector.js`).

These templates handle the mechanics of marshalling data between the Script Connector and the script code. In many cases, implementing a task is simply a matter of replacing the body of the ProcessEntity() method in the template script with your custom implementation. Similarly, replace the body of PerformCollection() to implement a collector. These template scripts and the provided code snippets also show the key script operations that you need to use to make your scripts interact with the Script Connector and IBM Content Collector. The samples provide additional information.

Reading task input parameters

When you configure a task, you specify property names and data types and provide an expression to provide a value.

The input metadata mappings look similar to this sample:



These passed parameters can be retrieved from an IEntityData object passed as part of the task input. To access these parameters you could use the following script:

```
// Read passed input parameters
// - The name of the parameter must match the name specified in the Task Configuration
// - You must use the getEntityData...() method appropriate for the parameter datatype
// - The task configuration must specify an expression that provides a value
var fileName = entityData.getEntityDataStringValue("FileName");
var fileExtension = entityData.getEntityDataStringValue("FileExtension");
var fileSize = entityData.getEntityDataInt64Value("FileSize");

// Determine if owner property exists in the input metadata and conditionally read data
var fileOwner = "";
```

```
if (entityData.haveParameter("FileOwner"))
{
    fileOwner = entityData.getEntityDataStringValue("FileOwner");
}
```

You must ensure the property names specified in the task configuration match the property names specified in the script (the comparison is case sensitive).

Date input parameters require special handling as the passed value is a COM object, rather than a JScript Date object:

```
// Get Date input
var dateIn = entityData.getEntityDataDateTimeValue("FileDate");

// New JScript Date
var fileDate = new Date();

// Set Date properties from input parameter object
// Input parameter date object properties are UTC
fileDate.setUTCFullYear( dateIn.year );
fileDate.setUTCMonth( dateIn.month - 1 ); // JScript Date, month is 0 based
fileDate.setUTCDate( dateIn.day );
fileDate.setUTCHours( dateIn.hour );
fileDate.setUTCMinutes( dateIn.minutes );
fileDate.setUTCSeconds( dateIn.seconds );
fileDate.setUTCMilliseconds( dateIn.milliseconds);

// Other supported properties
// dateIn.biasIsPositive    - Boolean, false for US, true for India
// dateIn.biasHours         - Integer, 5 for New York
// dateIn.biasMinutes       - Integer, 30 for Newfoundland
// dateIn.timezoneName      - String "EST"
// dateIn.lastDayOfMonth    - Integer in range 28-31
// dateIn.isLeapYear        - Boolean, true for 2012
```

Access Control List (ACL) Entry parameters are also passed as COM objects:

```
// Get ACL entry
// Supported properties
// aclEntry.principal - String, user or group name
// aclEntry.privilege - String, privilege name (not validated)
// aclEntry.isGranted - Boolean, true=granted, false=denied
var aclEntry =  entityData.getEntityDataACLEntryValue("Author");
if ( aclEntry.privilege == "ReadWrite" && aclEntry.isGranted )
{
    var author = aclEntry.principal;
    aclEntry.privilege = "Read";
    ...
}
```

If the input parameters are array types, the input data must be converted from SafeArray format into a JScript array. The script template defines a conversion function SafeArrayToJScriptArray() to perform this conversion. This is used as follows:

```
// Read array parameters
// - The passed value is a SafeArray, which is opaque to JScript
// - You must use SafeArrayToJScriptArray() to convert to a compatible array type
var readOnlyUsers = SafeArrayToJScriptArray(
 entityData.getEntityDataStringArrayValue("ReadOnlyUsers"));

// Use the array parameter
for (var idx in readOnlyUsers)
{
  var userName = readOnlyUsers[idx];
  ...
}
```

Date array input parameters require both array conversion and date conversion.

Gathering metadata within a script

The IEntityData API class provides a type-safe property bag that is used to gather metadata properties within a script. You create an instance for each metadata source emitted by your task or collector, except for task status and file metadata. You then add named properties to the property bag.

```
// Create an instance of IEntityData to hold custom metadata
var customMetadata = new ActiveXObject("ConfigurableConnectorDLL.EntityData");

// Add property values to it
customMetadata.setEntityDataStringValue("FileName", filename);
customMetadata.setEntityDataStringValue("FileExtension", extension);
customMetadata.setEntityDataIntValue("FileSize", size > 1024 ? size : 1024);

// A JScript Date type is required to return date/time properties
var fileDate = new Date();
fileDate.setUTCFullYear(2012);
fileDate.setUTCMonth(4);       // 0 based, 0 = January, 4 = May
fileDate.setUTCDate(31);       // 1 based
customMetadata.setEntityDataDateTimeValue("FileDate", fileDate);

// A JScript Object with principal, privilege, and isGranted properties
// is required to return ACLEntry properties
var aclEntry = new Object();
aclEntry.principal = owner;
aclEntry.privilege = "All";
aclEntry.isGranted = true;
customMetadata.setEntityDataACLEntryValue( "FileOwner", aclEntry );

// A JScript Array type is required to return array properties
var readOnlyUsers = new Array();
readOnlyUsers.push("User1");
readOnlyUsers.push("User2");

customMetadata.setEntityDataStringArrayValue("ReadOnlyUsers", readOnlyUsers);
```

The property bag typically contains the property values for user defined metadata sources or for custom metadata sources registered by the Script Connector. However, at this stage, the object is not associated with a specific metadata source and the properties you add are not validated (except to test for unique names). It is your responsibility to ensure the correctness of the property names and the data types of the values.

Returning metadata from a task script

Metadata is returned from a task to the task route service when the task exits. You add metadata to an entity by calling methods on ICOMCaptureUtilForScripts.

```
// Associate customMetadata with a specific metadata source
captureUtilApi.addCustomMetadata(entityId, "Custom.Metadata1", customMetadata);

// Update the file metadata
captureUtilApi.addFileMetadata(entityId, fileName);

// Update the entity with a status of success
captureUtilApi.addTaskStatusMetadata(entityId, true, @LogTrace2, "");
```

You add metadata to an existing entity by reusing an entity ID passed to the script within the task inputs. To add a new entity to the current entity set, you simply

provide a new unique entity ID and associate metadata with that entity ID. For example, a task could take an input file, render it to another file format, and add metadata for the new file.

```
var pdfFilename = filename + ".pdf";
if (RenderToPDF(filename, pdfFilename))
{
 var pdfEntityId = ("" + pdfFilename).toLowerCase();

  // Add file metadata, custom metadata and task status metadata to the pdf file entity
  captureUtilApi.addFileMetadata(pdfEntityId, pdfFilename);
  captureUtilApi.addCustomMetadata(pdfEntityId, "Custom.Metadata1", customMetadata);
  captureUtilApi.addTaskStatusMetadata(pdfEntityId, true, @LogTrace2, "");
}
```

Important: If you call addCustomMetadata(*EntityId*, *MetadataSourceId*, *Metadata*) for an existing entity, and the entity already has metadata associated with the same *MetadataSourceId* value, the existing metadata is overwritten, not updated. To avoid potential problems, use task-specific metadata sources. If you need to update existing metadata, all metadata properties that you need to preserve should be mapped to task input parameters and copied into the output metadata within the script.

Submitting metadata from a collector script

Unlike tasks, collectors do not wait until completion to submit metadata. Collector submission is managed by the API class ICOMCollectorSubmitProxyForScripts. When an item is found, an entity ID is assigned, and the metadata associated with the entity (or group of related entities) is compiled.

When the entity metadata is complete, it is submitted to the task route service by calling the send() method. The buffer must be cleared by calling clearTaskOutputs() before continuing to search for the next item.

```
// Create an instance of ICOMCollectorSubmitProxyForScripts to submit collected entities
var submitStub = captureUtilApi.createCollectorSubmitStubProxy();

try
{
  // Associate customMetadata with a specific metadata source
  submitStub.addCustomMetadata(entityId, "Custom.Metadata1", customMetadata);

  // Update the file metadata
  submitStub.addFileMetadata(entityId, fileName);

  // Update the entity with a status of success
  submitStub.addTaskStatusMetadata(entityId, true, @LogTrace2, "");

  // Submit an entity to the task route service
  submitStub.send();
}
finally
{
  // Clean up after submitting an entity
  submitStub.clearTaskOutputs();
}
```

Important: Collector scripts that were developed for IBM Content Collector 2.2 are not compatible with IBM Content Collector 3.0. See "Upgrading Script Connector scripts from earlier IBM Content Collector versions" on page 18 for details.

Logging

To write to the log, you call the ICOMCaptureUtilForScripts.logEvent() method, specifying a log level and a message.

```
captureUtilApi.logEvent(@LogTrace, "Processing entity with id " + entityId);
```

Each template script defines constants for the supported log levels:

```
// Log Level Constants
@set @LogFatal = 1;
@set @LogError = 2;
@set @LogWarning = 3;
@set @LogInfo = 4;
@set @LogTrace = 5;
@set @LogTrace2 = 6;
```

When you configure the Script Connector, you specify a log level and a log location. The log level you pass in the logEvent() method is compared with the configured log level, and only messages of equal or greater severity (lower log level) are logged.

Error handling

If an operation in a script causes an exception to be thrown, and the exception is not caught and handled within the script, the exception will be logged by the Script Connector. However, the information logged is unlikely to be precise enough to pinpoint the problem. For production use, scripts should take responsibility for handling errors.

Best practices for error handling:

- The execute() method should include a try...catch block, and the catch block should log the error.
- For tasks created using the `CustomTaskx.js` template files, the ProcessEntity() method should include a try...catch block. Within the catch block:
  - Log the error, clearly identifying the entity ID of the item that failed.
  - Call the ICOMCaptureUtilForScripts.entityError() method to update the error performance counters and write an event to the Windows Event log.
  - Set the task status in the entity metadata to indicate failure, including the exception message. Because the exception is caught, scripts can continue processing the next entity, or can invalidate all entities in the current entity set.

Updating performance counters

Most connectors use the IBM Content Collector performance counters to record how many entities are accessed, submitted, processed, and skipped. Production quality Script Connector should also update these performance counters using methods on ICOMCaptureUtilForScripts.

For more information on the methods used to update performance counters, consult the ICOMCaptureUtilForScripts reference in the Script API documentation. See “Script Connector API documentation” on page 47 for details on how to access the API documentation.

Using existing COM objects in script code

If your task performs anything more complex than simple data conversion, it will probably require the services of other COM classes to complete. Windows Scripting provides some very useful COM classes for use in scripts, including WScript.Shell for interacting with the Windows shell (for running programs and manipulating environment variables and registry entries) and Scripting.FileSystemObject for interacting with the file system.

These classes are so useful that they are automatically registered by the script template .wsc files:

```
<!-- Create Scripting.FileSystem object for managing files -->
<object id='FileSystem' progid='Scripting.FileSystemObject' />

<!-- Create WScript.Shell object for registry access etc. -->
<object id='Shell' progid='WScript.Shell' />
```

Registering these classes in this way means that static methods can be called directly using the ID specified in the .wsc file. This code snippet shows how a custom collector can use Scripting.FileSystemObject to iterate through the files in a folder:

```
function PerformCollection(captureUtilApi, taskInput)
{
  var filename, entityId, customMetadata;

  // Get the folder to collect from the configuration
  var folderName = taskInput.getConfigurationData();

  // Create a submit stub for returning the files that we find
  var submitStub = captureUtilApi.createCollectorSubmitStubProxy();

  // Get the specified folder object from the registered Scripting.FileSystemObject
  var folderObj = FileSystem.getFolder(folderName);

  for (var files = new Enumerator(folderObj.files); !files.atEnd(); files.moveNext())
  {
    if (captureUtilApi.getIsStopped())
      break;

    filename = "" + files.item();
    entityId = ("" + filename).toLowerCase();

    try
    {
      customMetadata = CreateCustomMetadata(filename);

      submitStub.addFileMetadata(entityId, filename);
      submitStub.addCustomMetadata(entityId, "Custom.Metadata1", customMetadata);
      submitStub.addTaskStatusMetadata(entityId, true, @LogTrace2, "");
      submitStub.send();
    }
    finally
    {
      submitStub.clearTaskOutputs();
    }
}
```

See the “File processing connector” on page 50 for a full implementation.

Instances of other root COM objects are instantiated by creating a new ActiveXObject, specifying the ProgId or the guid of the COM class:

```
var xmlDoc = new ActiveXObject("Msxml2.DOMDocument.6.0");
```

There are many other COM classes that you can use in your scripts. Microsoft MSXML classes are frequently used in Script Connector implementations, for reading and writing XML. Other APIs used include the Lotus Domino COM API, Novell GroupWise COM, Microsoft Office automation, FileNet IDM for Content Services and Image Services, IBM FileNet P8 3.5 COM APIs and .Net classes that expose COM Callable Wrappers.

If you are implementing multiple tasks, you might have significant duplicate code in your task scripts. Each task might define identical JScript classes to represent the same custom data objects. If you have significant duplicate script code, consider creating separate scripts that encapsulate the common functionality within JScript classes and use .wsc files to register these classes as COM objects. After registration, you can create and use instances of your classes:

```
var myClass = new ActiveXObject("MyConnector.MyCommonClass.1");
myClass.doSomething();
...
```

For some very specific requirements, it might be necessary to develop custom COM components using other programming languages (for example C++ or C#). In this case, the Script Connector simply provides a bridging technology to create an instance of your component and to call methods on it during task route execution. This approach means that you are free to implement your COM server as an out-of-process server that is packaged as an executable (.exe) that runs in its own memory space as a separate process, instead of an in-process COM server in the Script Connector's memory space. This separation can provide a more resilient and better performing system that gives you complete control of caching and object lifetimes. However, if you are contemplating this approach, you should also consider the alternative of developing a custom connector using the connector SDK.

Upgrading Script Connector scripts from earlier IBM Content Collector versions

Script Connector task and registration scripts developed for IBM Content Collector 2.2 and all scripts developed for IBM Content Collector 3.0 can be used by IBM Content Collector 3.0 FP2 without modification. However, collector scripts developed for IBM Content Collector 2.2 must be modified to change the way that entities are submitted to the task route service.

In IBM Content Collector 2.2, entities were submitted by a collector script using methods on the main API class ICOMCaptureUtilForScripts:

```
try
{
  // Associate customMetadata with a specific metadata source
  captureUtilApi.addCustomMetadata(entityId, "Custom.Metadata1", customMetadata);

  // Update the file metadata
  captureUtilApi.addFileMetadata(entityId, fileName);

  // Update the entity with a status of success
  captureUtilApi.addTaskStatusMetadata(entityId, true, @LogTrace2, "");

  // Submit an entity to the task route service
  captureUtilApi.sendTaskOutputs();
}
finally
{
  // Clean up after submitting an entity
  captureUtilApi.clearTaskOutputs();
}
```

In IBM Content Collector 3.0, Script Connector collector submission is managed by a separate API class ICOMCollectorSubmitProxyForScripts. The equivalent script for a version 3.0 collector implementation is:

```
// Create an instance of ICOMCollectorSubmitProxyForScripts to submit collected entities
var submitStub = captureUtilApi.createCollectorSubmitStubProxy();

try
{
  // Associate customMetadata with a specific metadata source
  submitStub.addCustomMetadata(entityId, "Custom.Metadata1", customMetadata);

  // Update the file metadata
  submitStub.addFileMetadata(entityId, fileName);

  // Update the entity with a status of success
```

```
  submitStub.addTaskStatusMetadata(entityId, true, @LogTrace2, "");

  // Submit an entity to the task route service
  submitStub.send();
}
finally
{
  // Clean up after submitting an entity
  submitStub.clearTaskOutputs();
}
```

When upgrading from IBM Content Collector 2.2 to 3.0, collector scripts must be updated to use ICOMCollectorSubmitProxyForScripts.

Testing script code

Consider how you will test your script as you develop it. Testing is much simpler if you are able to test the key functionality of your script without running multiple IBM Content Collector services.

This is relatively easy to accomplish:

- Enforce a clear separation between the methods in your script code that implement your core functionality and the methods that interface with the task route service (including methods provided by the script template and the methods you write that use Script Connector API classes). The aim is to create methods that can be called directly from a test script without running the task routing service.
- Expose the methods you want to expose to your test script by adding methods to the public interface definition in the .wsc file (see "Modifying .wsc files for COM registration" on page 27).
- Create a test script that creates an instance of your script as an ActiveXObject. Your test script can then call the public methods you exposed in the .wsc file, passing known test parameters.
- In general, you should minimize the number of methods in your script that call Script Connector API methods. For example, rather than peppering calls to ICOMCaptureUtilForScripts.logEvent() throughout your script code, write your own logging method that is aware of the calling context (test or run time). Your logging method simply calls the logEvent() method at run time, but can either ignore the logging messages or write them to a log file when called from a test script.

  For more complex test requirements, you might prefer to use JScript and .wsc files to create and register test classes that expose methods with the same names and parameters as the Script Connector API methods: instances of these test classes can then be passed to your test scripts in place of the actual Script Connector API classes.
- However, assuming that you are testing on a system with IBM Content Collector and the Script Connector installed, you should use IEntityData in your test scripts, both to pass parameters and to return custom metadata. This minimizes the chance of errors caused by duplicate property IDs and unexpected datatypes.

Writing connector scripts in Python

The provided Python utilities and template simplify the process of creating task and collector scripts. Python tasks and collectors can be implemented using base classes in the **`ibm.ctms.template`** namespace. These base classes encapsulate the boilerplate script code that the JScript script template provides for equivalent tasks and collectors that are written in JScript.

Windows does not ship with Python installed, so you must download and install a suitable Python distribution for Windows that supports 32-bit COM objects before you can write connector scripts in Python (see “Prerequisites” on page 5).

Creating tasks

Follow these steps to create new tasks from the script template.

To create a new task:

1. Import the **ibm.ctms.template** and **ibm.ctms.util** packages. Note that template namespaces are resolved using relative paths from the `ctms` folder.
2. Create a new class that inherits from **ibm.ctms.template.TaskBase**.
3. Specify the COM registration parameters for the class using standard pywin32 COM registration functionality. Registering Python task and collector scripts in this way means that a .wsc file is not required for COM registration, and multiple tasks and a collector can be packaged within a single Python file.
4. Define the constructor and destructor, setting a task-specific log entry prefix.
5. If each entity in the task input can be processed independently, you can implement the task functionality simply by overriding the ProcessEntity() method in the base class. If a failure in processing any one entity causes all submitted entities to fail, or if there are dependencies between entities, override the ProcessTaskInput() method and process the entities as a set.

The following Python script demonstrates the key steps. See the sample “Python Unix date conversion” on page 52 for a complete, working example.

```
# 1. Import template containing base classes and connector utilities
import ibm.ctms.template
import ibm.ctms.util

# 2. Class inherited from ibm.ctms.template.TaskBase
class CustomTask( ibm.ctms.template.TaskBase ) :

    # 3. COM registration properties
    _public_methods_ = [ "execute" ]
    _reg_progid_ = "IBM_CTMS.CustomTaskPy"
    _reg_desc_ = "Custom Task in Python"
    _reg_clsid_ = "{643E6E84-59DA-4E13-A4A4-2DD0329FCAFF}"

    # 4. Constructor/destructor
    def __init__( self ) :
        # Set the logging prefix – include a trailing space
        super( CustomTask , self ).__init__( "Custom.Task1: " )

    def __del__( self ) : #Destructor
        super( CustomTask , self ).__del__()

    # 5. ProcessEntity method implementation
    # \brief Task implementation, overriding virtual method in the template.
    # \param config        Task configuration (ibm.ctms.template.TaskConfig object)
    # \param entityId      Entity id (String)
    # \param entityData  Input parameter values for entityId (IEntityData instance)
    def ProcessEntity( self, config, entityId, entityData ) :

        # Log task start – config contains task and task route names and IDs
        self.LogEvent( ibm.ctms.util.LogLevels.Trace,
                       "Task " + config.Name + " in task route " + config.TaskRouteName +
                       " is processing entity " + entityId )

        try :
            # Read passed input parameters defined in the Script Connector task configuration
            paramValue = entityData.getEntityDataStringValue( "Parameter.ID1" )
            ...

            # Perform the task
            # config.ConfigString contains the task configuration
            ...

            # Create the output metadata and set the properties
            newMetadata = win32com.client.Dispatch( "ConfigurableConnectorDLL.EntityData" )
            newMetadata.setEntityDataStringValue( "Custom.Metadata1", propertyValue )
            ...

            # Add the output metadata to the entity
            self.CaptureUtilApi.addCustomMetadata( entityId, "Custom.Metadata1.Property1", newMetadata )
```

```
        # Add file metadata (if the task creates a file associated with the entity ID )
        self.CaptureUtilApi.addFileMetadata( entityId, fileName )

        # Set task status to success — leave this step to last in case of an error
        self.CaptureUtilApi.addTaskStatusMetadata( entityId, True, ibm.ctms.util.LogLevels.Trace2, "" )

    except Exception as e :
        # Get the exception details
        message = "Error: " + GetErrorMessage(e)

        #Log the error
        self.LogEvent( ibm.ctms.util.LogLevels.Error,
                       "Task failed for entity: " + entityId + " : " + message)

        # Set task status to failure
        self.CaptureUtilApi.addTaskStatusMetadata( entityId, False, ibm.ctms.util.LogLevels.Error, message )
```

Reading task input parameters

Task input parameters can be retrieved from an IEntityData object that is passed as part of the task input. To access these parameters, you can use the following scripts.

Reading non-array task input parameters in Python is straightforward:

```
# Read passed input parameters
fileName = entityData.getEntityDataStringValue("FileName")
fileExtension = entityData.getEntityDataStringValue("FileExtension")
fileSize = entityData.getEntityDataInt64Value("FileSize")

# Determine if owner property exists in the input metadata and conditionally read data
fileOwner = ""
if entityData.haveParameter("FileOwner") :
    fileOwner = entityData.getEntityDataStringValue("FileOwner")
```

Python, like JScript, must apply special handling to DateTimeVal input parameters, because the input data is a COM object rather than a native Python data type. Python scripts can use language features for manipulating dates and times (in the **datetime** package). Subclass the **datetime.tzinfo** class to use the time zone properties that are passed in the input object.

```
# Get Date input
# Input parameter date object properties are UTC
dateIn = entityData.getEntityDataDateTimeValue("FileDate")

# Convert to Python datetime.datetime
fileDate = datetime.datetime( dateIn.year, dateIn.month, dateIn.day,
                              dateIn.hour, dateIn.minutes, dateIn.seconds,
                              dateIn.milliseconds * 1000 )
```

Access Control List (ACL) Entry parameters are also passed as COM objects:

```
# Get ACL entry
aclEntry =  entityData.getEntityDataACLEntryValue("Author")
if ( aclEntry.privilege == "ReadWrite" and aclEntry.isGranted ) :
    author = aclEntry.principal
    aclEntry.privilege = "Read"
    ...
```

Reading array input parameter in Python is simpler than in JScript, because Python can work with COM SafeArrays directly (JScript requires the function SafeArrayToJScriptArray() to convert a SafeArray to a JScript array):

```
# Read array parameters
# The passed value is a SafeArray
readOnlyUsers = entityData.getEntityDataStringArrayValue("ReadOnlyUsers")

# Use the array parameter
for userName in readOnlyUsers :
    ...
```

Gathering metadata within a script

In Python scripts, you create an IEntityData instance to gather output metadata and set metadata properties in much the same way as in the equivalent JScript code.

```
# IEntityData creation
customMetadata = win32com.client.Dispatch( "ConfigurableConnectorDLL.EntityData" )

# Simple property
stringVal = "A day in the life"
customMetadata.setEntityDataStringValue( "Metadata.StringVal", stringVal )
```

Use a Python datetime.datetime instance to return datetime properties. When you work with third-party libraries, you can also return COM dates and variants containing dates.

```
# DateTime properties- Use datetime.datetime (UTC assumed)
dateVal = datetime.datetime( 2012, 10, 22, 14, 58, 21, 0 )
customMetadata.setEntityDataDateTimeValue( "Metadata.DateTimeVal", dateVal )
```

The native Python types that represent ACL entries (including values in an ACL entry array) must be wrapped in an IDispatch wrapper.

```
# ACL entry properties – Wrap an ACLEntry instance
aclVal = ibm.ctms.util.ACLEntry( "dvader", "superuser", true )
customMetadata.setEntityDataACLEntryValue( "Metadata.ACLEntryVal",
    win32com.server.util.wrap( aclVal ) )
```

The Script Connector is flexible in the way it handles array parameters, accepting lists, tuples, and array.array types as input.

```
# Array properties – Can use Python lists, tuples or numeric arrays
intList = [1, 1, 2, 3, 5, 8, 13, 21, 34]
customMetadata.setEntityDataInt64ArrayValue( "Metadata.Int64ArrayVal", intList )
```

Returning metadata from a task script

Metadata is returned from a task by calling methods on the instance of ICOMCaptureUtilForScripts that is initially passed as a parameter to the execute method. The Python template base class exposes this object to script code as the CaptureUtilApi instance member.

```
# Associate customMetadata with a specific metadata source
self.CaptureUtilApi.addCustomMetadata( entityId, "Custom.Metadata1", customMetadata )

# Update the file metadata
self.CaptureUtilApi.addFileMetadata( entityId, fileName )

# Set task status to success
self.CaptureUtilApi.addTaskStatusMetadata( entityId, True, ibm.ctms.util.LogLevels.Trace2, "" )
```

Creating collectors

The steps for creating a new collector from the script template are very similar to creating a task.

To create a new collector:

1. Import the **ibm.ctms.template** and **ibm.ctms.util** packages.
2. Create a new class that inherits from **ibm.ctms.template.CollectorBase**.
3. Specify the COM registration parameters for the class.
4. Define the constructor and destructor, setting a collector-specific log entry prefix.
5. Implement the PerformCollection() method.

The following Python script demonstrates the key steps:

```
# 1. Import template containing base classes and connector utils
import ibm.ctms.template
import ibm.ctms.util

# Used by example
import os, os.path
import base64

# 2. Class inherited from ibm.ctms.template.CollectorBase
class CustomCollector( ibm.ctms.template.CollectorBase ) :

    # 3. COM registration properties
    _public_methods_ = [ "execute" ]
    _reg_progid_ = "IBM_CTMS.CustomCollectorPy"
    _reg_desc_ = "Custom Collector in Python"
    _reg_clsid_ = "{1B5006C9-ACFF-4CB7-ABA1-1DA8DFB0CC8F}"

    # 4. Constructor/destructor
    def __init__( self ) :
        # Set the logging prefix - include a trailing space
        super( CustomCollector , self ).__init__( "Custom.Collector: " )

    def __del__( self ) : #Destructor
        super( CustomCollector , self ).__del__()

    # 5. PerformCollection method implementation
    # \brief Collector implementation, overriding virtual method in the template.
    # \param config       Collector configuration (ibm.ctms.template.TaskConfig object)
    def PerformCollection( self, config ) :

        # config contains collector and task route names and ids
        # config.ConfigString contains the collector configuration
        folderName = config.ConfigString

        # Perform the collection
        if os.path.exists( folderName ) :
            for root, dirs, files in os.walk( folderName ) :
                for f in files :
                   # Check for end of processing window
                   if self.CaptureUtilApi.getIsStopped() :
                        self.LogEvent( ibm.ctms.util.LogLevels.Trace,
                                       "Collector " + config.Name + " is stopping" )
                        return

                    # Process the file
                    fullpath = os.path.join( root, f )
                    if os.path.splitext( fullpath )[1]==".txt" :
                        # Located a text file in the target folder
                        # Submit the file to the task route
                        self.SubmitFile( fullPath )
```

Submitting metadata from a collector script

Entities are submitted using an instance of ICOMCollectorSubmitProxyForScripts, exposed by the collector base class as the SubmitStub instance member.

Continuing the pervious example, typical usage might be:

```
def SubmitFile( self, filePath ) :

    # Make sure we have a value in case of an error
    entityId = "Not yet assigned"

    try :
        # Create an opaque entity ID that maps 1:1 with file path
        # Using filename based entity ID prevents re-submission while file is being processed
        entityId = base64.b64encode( filePath.lower() )

        # Create collector specific output metadata and set the properties
        newMetadata = win32com.client.Dispatch( "ConfigurableConnectorDLL.EntityData" )
        ...

        # Add the output metadata to the entity
        self.SubmitStub.addCustomMetadata( entityId, "Custom.Metadata1", newMetadata )

        # Add file metadata (if the collector finds or creates a file associated with the entity ID)
        self.SubmitStub.addFileMetadata( entityId, filePath )

        # Set task status to success
```

```
        self.SubmitStub.addTaskStatusMetadata( entityId, True, ibm.ctms.util.LogLevels.Trace2, "" )

        # Submit the entity to the Task route service
        self.SubmitStub.send()

        #Log the submission
        self.LogEvent( ibm.ctms.util.LogLevels.Trace,
                       "Send succeeded for file " + filePath + " as " + entityId )

    except Exception as e :
        # Get the exception details
        message = "Error: " + GetErrorMessage(e)

        #Log the error
        self.LogEvent( ibm.ctms.util.LogLevels.Error,
                       "Send failed for file " + filePath + " as " + entityId + " : " + message )

        # Clear the metadata that failed submission
        # Entities are not normally submitted if there is an error
        self.SubmitStub.clearTaskOutputs()
```

Assigning entity IDs

All entity IDs are strings that are used without modification by the task route service to uniquely identify an entity. To prevent possible data corruption if an entity is currently being processed, the task route service throws an error if the same entity is resubmitted. This means that your entity ID should accurately reflect the identity semantics of the item it represents, be it a file, mail message, a document in a document repository, or a unique version of a document.

Therefore, an entity should always be assigned the same entity ID, and that entity ID should be distinct from the entity IDs assigned to all other entities. The difficulty lies in defining what an entity is and what defines uniqueness and identity.

Note the way that entity IDs are generated in the code snippet that is shown in the topic about submitting metadata from a collector script. The code sample uses entity IDs that are based on the full file path, thus defining an entity as a file with a specific name at a specific location on the file system, factoring in that Windows file names are not case sensitive. Thus, two files with the same name in different locations are considered to be different entities, as are any two distinct files with the same binary content. If you wanted to define an entity by its binary content, you could generate a hash key from the file contents. If the entity can be processed safely by multiple connectors and tasks at the same time, you can also generate a guid as the entity ID. However, you must ensure that entity IDs are unique.

When writing connector scripts, you can give meaning to the entity ID by using a file name as an entity ID or by generating an entity ID for an email message by using a combination of the server name, the mailbox identifier, and the message identifier. However, other tasks will not be able to parse your entity ID to obtain this information. A better alternative is to provide this information as metadata, so that you can use this information to create decision point rules and can map the information to document properties during archiving. In the code snippet, the file name is encoded to preserve the identity semantics without conveying obvious meaning.

Logging

The template base classes define a LogEvent() method that filters redundant logging calls. The first parameter, the log level, can be specified as an integer, but using the ibm.ctms.util.LogLevels enumeration improves readability.

```
# Log the current entity ID
self.LogEvent( ibm.ctms.util.LogLevels.Trace, "Processing entity " + entityId )
```

Error handling

The Python template classes catch unhandled exceptions that are thrown within your script code, log an error, and might wrap the error in a COMException to provide additional context to the task route service. The template also provides a method, GetErrorMessage(), that obtains a stack trace for Python error messages to provide detailed context for the error. If the exception results from a call to a method on any of the Script Connector API types, calling getErrorInfo() returns an object that contains any additional error information.

As the provided Python code snippets show, you should take control of exception handling within your scripts. Exception handling procedures differ between task and collector scripts.

In the exception handler of a task script, you should log the error, clearly identifying the entity ID of the item that failed, and set the task status in the entity metadata to indicate failure, providing the error message. If a failure in processing one entity means that all entities should fail, call self.CaptureUtilApi.clearTaskOutputs() to clear the metadata added by the task prior to the failure. Then set the task status of all entities to error (you need to keep track of the entity IDs).

In a collector script, you also log the error but rather than submitting an entity with an initial status of error, simply skip the item. Following an error, you can choose to continue collecting other items or to stop collection. Your collector might also manage some form of blacklist to avoid reprocessing problem items that are likely to fail repeatedly.

Updating performance counters and Windows event logs

To enable monitoring of your collector using the IBM Content Collector System Dashboard and other system tools, your collectors can call the following methods on ICOMCaptureUtilForScripts (self.CaptureUtilApi). Calling these methods writes entries to the IBM Content Collector task route event logs (the event logs are named CTMS - *task_route_name*) and update the Windows performance counters that record how many locations the collector searches, how many entities the collector accesses, and how many cause errors.

Method	Parameters	Usage	Event ID	Performance counter
locationStarted	*Task Route Name*, *Collector Name*, *Location*	Collectors, when starting collection from a specific location (such as a file folder).	144	
locationFinished	*Task Route Name*, *Collector Name*, *Location*, *Error Message*	Collectors, after stopping collection from a specific location. If an error occurred, include the error message, otherwise specify `None`.	145, 147, 149	CTMS Collector, Searched Locations/sec
entityAccessed	*Task Route Name*, *Collector Name*	Collectors, when processing a candidate entity.		CTMS Collector, Accessed Entities/sec
entitySkipped	*Task Route Name*, *Collector Name*, *Entity ID*, *Location*, *Reason*	Collectors, when skipping a candidate entity, specifying the reason for skipping the entity (for example, previously processed or captured, access denied).	119	

Method	Parameters	Usage	Event ID	Performance counter
entityError	*Task Route Name*, *Collector Name*, *Entity ID*, *Location*, *Error Message*, *Additional Information*	When processing a specific entity caused an error.	120	CTMS Collector, Entity Errors/sec

You can use the following method in tasks that update target systems to record the number of documents created in a target repository.

Method	Parameters	Usage	Performance counter
documentCreated	*Task Route Name*, *Task Name*	After adding a new document to a target document repository.	CTMS Target, Documents Created/sec

Note that the Script Connector and the task route service also write to the event log, and they update further performance counters as a side effect of calling other methods.

Using existing COM objects in script code

The Python language and Python Standard Library provide all the functionality that you need to implement many script connector tasks. Other tasks can be implemented using third-party libraries, for example, to access web servers or to read Microsoft Office document properties.

Python is also very well suited to working with other COM types because it supports both late (IDispatch) and static binding (using `makepy.py` to generate a wrapper from a type library), and it allows low-level access to COM and Windows API internals if necessary.

The .wsc files that are used to register JScript task and collector scripts typically register WScript.Shell and Scripting.FileSystemObject, making them available to script code (as Shell and FileSystem, respectively). In Python scripts, you typically use equivalent functionality provided by the Python Standard Library and other modules. However, because these are standard COM objects, you can still use them from Python if necessary (for example when porting JScript tasks to Python).

```
wsShell = win32com.client.Dispatch( "WScript.Shell" )
fileSystem = win32com.client.Dispatch( "Scripting.FileSystemObject" )
```

Preparing for deployment

Modifying .wsc files for COM registration

Your scripts must be registered as COM objects before the Script Connector can call them.

To register all scripts that are written in JScript or VBScript and connector registration scripts that are written in Python, you create a .wsc file and register the .wsc file using `Regsvr32.exe`. A .wsc file is an XML file that specifies the information needed to register a script as a COM type, including the COM class properties, public interface, and the script location. Python task and collector scripts are registered using the built-in support for COM registration in pywin32.

A typical .wsc file used by the Script Connector will include the following elements and attributes:

```
<?xml version='1.0'?>
<component>
...
    <!-- COM registration added to Windows registry -->
    <registration
        description="Unix Date Conversion Task"
        version="1.00"
        classid="{632EBF06-D586-4E6E-97BB-C2375B211A82}"
        progid="IBM_CTMS.UnixDateConversionTask"
    />
...
    <public>
        <!-- Script method that is called when task is invoked by the Task Routing Service -->
        <method name='execute' internalName='Execute' dispid='1' >
            <PARAMETER name='comTaskInputs' />
            <PARAMETER name='captureUtilApi' />
        </method>
    </public>
...
    <!-- Language and location of script code -->
    <script language='JScript' src='UnixDateConversionTask.js' />
</component>
```

Update the following attributes:

Table 9. Attributes to be updated

Element	Attribute	Required update
registration	description	Provide a user-friendly display name for the COM class that provides access to your script. This name is primarily used by developer tools and class browsers that show installed COM classes. Note that this name is distinct from task and collector IDs and display names shown in Configuration Manager.

Table 9. Attributes to be updated (continued)

Element	Attribute	Required update
registration	classid	Replace with a unique guid specific to your task or collector. **Important:** Do not update the classid in the .wsc file for the connector configuration script (`ConnectorConfiguration.wsc`). The Script Connector always accesses the configuration script using the hard coded guid {4FB50180-2060-41A4-AD98-66AA88807088}. See "Generating new guids for tasks and collectors" for more information.
registration	progid	The ProgId used to register the COM class in HKEY_CLASSES_ROOT in the Windows Registry. This name can be used to create an instance of the script class programmatically. The format of a ProgId is: *Program.Component.Version* A ProgId has a maximum length of 39 characters, cannot start with a digit, and cannot contain punctuation, spaces, or underscores (except for periods between components).
method	internalName	Required if the name of the method in the script code does not match the method name in the name attribute. No action is required if you are using the script templates as the method stubs have already been created.
script	language	No action required if the script is written in JScript, otherwise change to match the scripting language used (VBScript, Python).
script	src	Update to specify the location of the script code. Can include an absolute or relative file path.

Generating new guids for tasks and collectors

Each task and collector registered on the system must have a unique guid. Generating a new guid for each task and collector script makes it possible to use scripts from third parties without naming conflicts and protects against unexpected results from the inadvertent use of the wrong script.

The `ScriptConnector\Tools` folder contains a simple script that can be used to generate new guids for JScript scripts. If you use Python, you can generate new guids with the Python interpreter.

To generate new guids:

- If you use JScript, open a Windows command prompt. Navigate to the Tools folder and run the following command:

  ```
  cscript GuidGenerator.js //NoLogo
  ```

  Or, to redirect the output to a file:

  ```
  cscript GuidGenerator.js //NoLogo >> guids.txt
  ```

- If you use Python, enter the following commands into the Python interpreter:

```
>>> import pythoncom
>>> print pythoncom.CreateGuid()
```

The generated guid is used to:

- Replace the default guid specified in the classid attribute in the registration element of the task or collector script's .wsc file to enable COM registration. See "Modifying .wsc files for COM registration" on page 27 for details.
- Replace the default guid specified in the DescribeMethods() method in the connector registration script (`ConnectorConfiguration.js`). See "Creating a registration script" for details.

Creating a registration script

During IBM Content Collector installation, the connectors you choose to install are automatically registered by the installer.

The connector registration process:

- Performs Windows service registration, so the connector executable can be run as a Windows service.
- Registers with IBM Content Collector the set of tasks that the connector can perform. For each task, the task list specifies a task ID, the task type (either collector, task, or end), and the metadata types emitted by the task.

The Script Connector is not registered by the installer, because the tasks it will perform and the metadata emitted by these tasks are entirely dependent on your specific implementation. However, connectors can also be registered (and unregistered) manually, by running the connector executable from the command line, passing appropriate command line arguments. The Script Connector is registered from the command line. The tasks and metadata sources to be registered are specified using a custom registration script.

For registration, the Script Connector uses the same scripting technology that it uses at run time. During the registration process, the Script Connector creates a COM object using a specific guid, then calls specific methods on that object to obtain the information required for registration. This guid corresponds to a customized registration script that you provide, describing your specific implementation.

Creating a registration script using JScript

`ConnectorConfiguration.js` provides a template for a customized JScript registration script, which you then complete and register using `ConnectorConfiguration.wsc`.

You must make the following changes to `ConnectorConfiguration.js` so that the Script Connector can register your implementation:

1. Optional: Update the display name used when registering the Script Connector as a Windows service:

   ```
   // Display name for the Windows service. You can change this.
   var WindowsDisplayName = "IBM Script Connector";
   ```

2. Optional: In DescribeMethods(), specify any custom metadata sources that should be registered when your connector is registered.

   The metadata sources you register will be considered *system* metadata sources and cannot be edited by users, but are otherwise functionally identical to *user*

metadata sources defined in the Configuration Manager. Custom metadata sources should only be created when your collector or tasks or both generate a consistent set of metadata properties and where your implementation is always the source of that metadata.

Your collector and tasks are not restricted to generating custom metadata; they can also generate other types of metadata including file metadata, task status metadata, user defined metadata, and other types of system metadata that are implemented using `SimpleMetadata.dll` (except for email metadata, most are).

```
function DescribeMethods(captureUtilApi)
{
    // Optional: Describe custom metadata created by our connector
    DescribeCustomMetadata(captureUtilApi);

    // Describe tasks and collectors implemented by our connector
    ...
}

// Define custom metadata sources for registration
function DescribeCustomMetadata(captureUtilApi)
{
    // Create a new custom metadata source
    var customMetadataSource = new MetadataSource("Custom.Metadata1");

    // Construct property descriptors for the metadata source
    var customMetadataProperties = new Array();

    customMetadataProperties.push(
        new MetadataProperty("Custom.Metadata1.Property1", TypeSystem2.StringVal));

    customMetadataProperties.push(
        new MetadataProperty("Custom.Metadata1.Property2", TypeSystem2.StringVal));

    // Register our custom metadata source
    captureUtilApi.addMetadataDescription(customMetadataSource, customMetadataProperties);
}
```

The convenience types, MetadataSource and MetadataProperty, are predefined in `ConnectorConfiguration.js` in the Script Connector template.

This example uses text keys (Custom.Metadata1) that are unlikely to cause naming collisions but are not in a friendly format for display to the user. In this case, the display names shown to the user should be specified in the localization file. However, you might prefer to define your metadata sources using names that can be displayed directly without a localization entry (such as *Custom Metadata*), especially if you need to support only a single language.

3. Specify the tasks (0, 1, or more) and collectors (0 or 1) that should be registered when your connector is registered. To register a task or collector, you need to specify:
   - A unique ID for each task (the collector ID is fixed)
   - A unique guid generated specifically for each task and collector (the same guid must be specified in the corresponding .wsc file)
   - The metadata sources generated by the task or collector
   - The type of task (collector, task, end)

   For example, to register a task and collector:

```
function DescribeMethods(captureUtilApi)
{
    // Describe custom metadata created by our connector
    ...

    // Describe tasks and collectors implemented by our connector
    // Having one method call per task makes it easier to mix and match tasks in one connector
    DescribeCustomTask1(captureUtilApi);
    DescribeCustomCollector(captureUtilApi);
}

// Register a custom task
function DescribeCustomTask1(captureUtilApi)
{
    // Task properties: ID, Guid
    var taskId = "Custom.Task1";
    var taskGuid = "{F5103CD7-DBAC-49B7-902F-B0AED50CA9F9}";   // unique to this task
```

```
    // Metadata emitted by the task.
    var taskMetadata = new Array();
    taskMetadata.push(new MetadataOutput("ctms.taskstatus.metadata")); // Task status
    taskMetadata.push(new MetadataOutput("Custom.Metadata1"));         // Our custom type

    // Register our task
    // ICOMCaptureUtilForScripts.addMethodDescription() expects the following parameters
    //
    // taskId:    Task or collector ID used internally (Registry, DataStore)
    // guid:      COM class GUID specified in the .wsc file
    // metadata:  Metadata sources emitted
    // type:      Task type (task, collector, end)
    captureUtilApi.addMethodDescription(taskId, taskGuid, taskMetadata, TaskTypes.task);
}

// Register a custom collector
function DescribeCustomCollector(captureUtilApi)
{
    // Collector properties: ID (fixed), Guid
    var collectorId = "ibm.ctms.connector.configurable.collector";
    var collectorGuid = "{BCF4DE64-90D9-4DAD-933A-D7493CF4BF62}";   // unique to this collector

    // Metadata emitted by the collector.
    var collectorMetadata = new Array();
    collectorMetadata.push(new MetadataOutput("ctms.taskstatus.metadata"));  // Task status
    collectorMetadata.push(new MetadataOutput("ctms.file.metadata"));        // File metadata
    collectorMetadata.push(new MetadataOutput("Custom.Metadata1"));          // Our custom type

    // Register our collector
    // ICOMCaptureUtilForScripts.addMethodDescription() expects the following parameters
    //
    // taskId:    Task or collector ID used internally (Registry, DataStore)
    // guid:      COM class GUID specified in the .wsc file
    // metadata:  Metadata sources emitted
    // type:      Task type (task, collector, end)
    captureUtilApi.addMethodDescription(collectorId, collectorGuid, collectorMetadata,  TaskTypes.collector);
}
```

The convenience type, MetadataOutput, is predefined in `ConnectorConfiguration.js` in the Script Connector template.

Creating a registration script in Python

Unlike Python task and collector scripts, Python connector registration scripts must be registered as COM objects by using .wsc files. For convenience, the Python registration script can be packaged in the same .py file as the task and collector classes.

For registration, four module level functions must be implemented:

- CreateObject()
- GetWindowsDisplayName()
- GetConnectorName()
- DescribeMethods()

The CreateObject(), GetWindowsDisplayName(), and GetConnectorName() functions are boilerplate code. Use the code as is. However, you can change the Windows service display name.

```
# Required imports for registration
import ibm.ctms.util
import win32com.server.util

# Create an expando object that implements IDispatchEx
# Used at run time to pass information to scripts
# Do not change this
def CreateObject( ignr ) :
    newObject = ibm.ctms.util.Object()
    return win32com.server.util.wrap( newObject, usePolicy=win32com.server.policy.DynamicPolicy )

# Get the connector's Windows service display name for registration
# You can change this
def GetWindowsDisplayName( ignr ) :
    return "IBM Content Collector Script Connector"

# Get the connector name for registration
# Used in the datastore, Windows registry, .adf and localization files
# Do not change this
def GetConnectorName( ignr ) :
    return "ibm.ctms.connector.configurable.Connector"
```

If Python functions are exposed within the public interface of a COM object that is registered using a .wsc file, they must include an additional parameter that is required but unused. This is an artifact of the .wsc COM registration process. The equivalent JScript code does not require this redundant parameter, because the JScript language supports variable numbers of parameters.

The DescribeMethods() function must be implemented by your connector registration script to provide registration information for your connector's tasks, collectors, and custom metadata sources.

```
# Describe tasks, collectors and custom metadata sources for registration
# captureUtilApi ( ICOMCaptureUtilForScripts ) provides API for interacting with the Script Connector
def DescribeMethods( captureUtilApi, ignr ) :

    # Describe custom metadata to be registered by our connector
    DescribeCustomMetadata( captureUtilApi )

    # Describe tasks and collectors implemented by our connector
    # Multiple tasks can be registered but only 1 collector
    # Separate method calls makes it easier to manage tasks
    DescribeCustomTask( captureUtilApi )
    DescribeCustomCollector( captureUtilApi )
```

Registering custom metadata sources is optional, because your connector can also use file system metadata, task status metadata, and any existing user defined metadata sources, but it is useful to guarantee the availability of specific metadata sources with known metadata properties. These metadata sources are created as system metadata sources when your connector is registered and cannot be edited by users.

The **ibm.ctms.util** namespace contains two convenience classes, MetadataSource and MetadataProperty, to provide information about metadata sources (an identifier, and the name and location of the method used to create an instance) and metadata properties (an identifier and the data type). These classes are native Python types, therefore they must be wrapped in an IDispatch wrapper using the win32com.server.util.wrap method to expose their properties to the Script Connector. The **ibm.ctms.util** namespace also provides an enumeration, TypeSystem2TypeIds, that defines all valid data types.

The metadata source and property information is passed back to the Script Connector by calling ICOMCaptureUtilForScripts.addMetadataDescription().

```
# Describe custom metadata to be registered by our connector
def DescribeCustomMetadata( captureUtilApi ) :

    # Metadata source descriptor
    customMetadataSource = win32com.server.util.wrap( ibm.ctms.util.MetadataSource( "Custom.Metadata1" ) )

    # Array of property descriptors for the properties in the metadata source
    customMetadataProperties = [
        win32com.server.util.wrap(
            ibm.ctms.util.MetadataProperty( "Custom.Metadata1.Property1", ibm.ctms.util.TypeSystem2TypeIds.StringVal ) ) ,
        win32com.server.util.wrap(
            ibm.ctms.util.MetadataProperty( "Custom.Metadata1.Property2", ibm.ctms.util.TypeSystem2TypeIds.StringVal ) )
    ]

    # Register our custom metadata types
    captureUtilApi.addMetadataDescription( customMetadataSource, customMetadataProperties )
```

Your connector registration script must provide registration information for at least one task or collector. You can register multiple tasks, but only one collector can be registered (because the collector ID is fixed). To register a task or collector, you call ICOMCaptureUtilForScripts.addMethodDescription(). This method expects the following parameters:

- A unique ID for each task (the collector ID is fixed).
- The COM class ID specified in the task or connector class.

- The metadata sources generated by the task or collector, including task status metadata, file metadata, other system metadata types, and user defined metadata. Use an array of wrapped ibm.ctms.util.MetadataOutput instances to specify metadata source types.
- The type of task (collector, task, end). Set the corresponding value in the ibm.ctms.util.TaskTypes enumeration.

```
# Describe a task implemented by our connector
def DescribeCustomTask( captureUtilApi ) :

    # Task properties: ID, Guid
    taskId = "Custom.Task"
    taskGuid = CustomTask._reg_clsid_

    # Array of metadata source types emitted by the task.
    taskMetadata = [
        win32com.server.util.wrap( ibm.ctms.util.MetadataOutput( "ctms.taskstatus.metadata" ) ) ,
        win32com.server.util.wrap( ibm.ctms.util.MetadataOutput( "Custom.Metadata1" ) )
    ]

    # Register our task
    captureUtilApi.addMethodDescription( taskId, taskGuid, taskMetadata,
                                         ibm.ctms.util.TaskTypes.task )

# Describe the collector implemented by our connector
def DescribeCustomCollector( captureUtilApi ) :

    # Collector properties: ID (fixed), Guid
    collectorId = "ibm.ctms.connector.configurable.collector"
    collectorGuid = CustomCollector._reg_clsid_

    # Array of metadata source types emitted by the collector.
    collectorMetadata = [
      win32com.server.util.wrap( ibm.ctms.util.MetadataOutput( "ctms.taskstatus.metadata" ) ) ,
      win32com.server.util.wrap( ibm.ctms.util.MetadataOutput( "ctms.file.metadata" ) ) ,
      win32com.server.util.wrap( ibm.ctms.util.MetadataOutput( "Custom.Metadata1" ) )
    ]

    # Register our collector
    captureUtilApi.addMethodDescription( collectorId, collectorGuid, collectorMetadata,
                                         ibm.ctms.util.TaskTypes.collector )
```

Enabling task and collector configuration in Configuration Manager

If you have implemented custom tasks or collectors, Configuration Manager needs some additional information to associate your task or collector with the user interface components that you use to configure it. This information is specified in the associated .adf file that you added to the `adf` folder when you set up your working folder from the template.

The .adf files for tasks must be updated to ensure that each task has an ID that does not conflict with task IDs in any other .adf file. This step is not necessary for custom collectors because you can only have one defined.

To update the .adf file associated with a custom task:

1. Open the file in a text editor.
2. Use search and replace to replace the value Custom.Task1 with the task ID of your custom task (as registered in your connector configuration script). The .adf file contains comments that describe how to do this.

Customizing configuration screens

The labels and tooltips displayed in the Script Connector user interface elements in Configuration Manager are defined in the file `ConfigurableConnector.Resource.en-US.xml`. This file is added to the localization folder when you set up your working folder from the template.

To change the labels and tooltips, update the text contained within CDATA sections in the label and info elements. The localization file contains XML comments that document where each label appears in Configuration Manager.

If you implement multiple tasks or custom metadata sources, or both, you can provide more readable display names to the user. `ConfigurableConnector.Resource.en-US.xml` contains commented-out sections within the Tasks and Metadata elements that show the changes that are necessary to provide task and metadata display names. The task, metadata source, and metadata property IDs that you supply must match those specified in the connector registration script (see "Creating a registration script" on page 29).

To localize for other languages, create a copy of `ConfigurableConnector.Resource.en-US.xml` in the localization folder, replacing en-US with the appropriate language abbreviation (see other Configuration Manager language files for valid suffixes). Then, translate the text in the label and info elements.

If you are implementing multiple tasks and metadata sources, you can provide separate localization files for each task. This means that there is a 1:1 correspondence between .js, .py, .wsc, .adf, and localization files. The Unix date conversion ("Unix date conversion" on page 49) and Web service call ("Web service call" on page 51) samples use this approach.

Deploying your connector

Following the correct deployment sequence is essential to prevent problems. The steps detailed here can easily be automated using one or more batch files.

For all commands specified in this section, it is assumed that the current directory is the `ctms` directory under the root installation folder for IBM Content Collector. Running these commands requires Administrator access.

In a scale-out environment, repeat the deployment procedure on each node.

Important: On a 64-bit operating system, you must use the 32-bit version of `RegSvr32.exe` for successful registration of the .wsc files. Using the full file path ensures that you use the right version of `RegSvr32.exe`. Do not use right-click registration for the same reason. The RegSvr32 file path specified in this section is the default for a 64-bit Windows system. Modify it accordingly.

Prerequisite steps

Before you can deploy and register the new implementation, you have to close Configuration Manager. For a new installation, you must make sure that the configuration library is registered. For modifying an existing Script Connector implementation, further steps are required.

1. Close Configuration Manager.
2. Stop the IBM Content Collector Configuration Access service, either from `Services.msc` or using the following command:
   ```
   sc stop ibm.ctms.ui
   ```
3. For a new installation, make sure the configuration library is registered.
   ```
   C:\Windows\SysWOW64\RegSvr32.exe /s ConfigurableConnectorDLL.dll
   ```

Unregistering the existing implementation

If you are modifying an existing Script Connector implementation, you must remove the existing Script Connector service and COM registration before replacing or deleting any of the deployed files.

The correct unregistration procedure removes all Script Connector entries from the Windows registry. Obsolete entries in the Windows registry are difficult to clean up and can lead to unexpected behavior and errors that are difficult to troubleshoot.

To unregister an existing installation:

1. Unregister the Script Connector.
   ```
   ConfigurableConnector.exe -u
   ```
2. Unregister task, collector, and configuration COM objects. To unregister JScript scripts, use `RegSvr32.exe`. In the following example, .wsc file names are for a collector and two tasks using the template filenames. Modify appropriately based on the .wsc filenames for the tasks and collector (if any) that are currently deployed.

© Copyright IBM Corp. 2008, 2012

```
C:\Windows\SysWOW64\RegSvr32.exe /s -u CustomTask1.wsc
C:\Windows\SysWOW64\RegSvr32.exe /s -u CustomTask2.wsc
C:\Windows\SysWOW64\RegSvr32.exe /s -u CustomCollector.wsc
C:\Windows\SysWOW64\RegSvr32.exe /s -u ConnectorConfiguration.wsc
```

To unregister Python scripts, use the -unregister command line argument. The connector configuration script, however, must be unregistered using `RegSvr32.exe`. For example:

```
python PyConnector.py –unregister
C:\Windows\SysWOW64\RegSvr32.exe -u PyConnectorConfig.wsc
```

Ensure you have a backup of the deployed files. Having the registration script and COM registration details means you can clean up the Windows registry manually in the event of problems.

Deploying the new implementation files

To deploy the new implementation files, you must copy them to the appropriate location in the IBM Content Collector installation directory.

To deploy the files:

1. Copy the .js (or .py) and .wsc files for the configuration script and for the tasks and collectors that you are deploying from your working folder to the `ctms` directory.
2. Copy localization files from your working `localization` folder to the `ctms\localization` folder.
3. Copy .adf files from your working `adf` folder to the `ctms\adf` folder.
4. Remove any obsolete files that were not replaced by the new deployment.

Registering the new implementation

After you deployed the new implementation files, you must register the task, collector, and configuration COM objects, and the Script Connector service and metadata.

To register the new implementation:

1. Register task, collector, and configuration COM objects. To register JScript scripts, use `RegSvr32.exe`. In the following example, the .wsc file names are for a collector and two tasks using the template filenames. Modify appropriately using the .wsc filenames for the collectors and tasks you are actually deploying.

   ```
   C:\Windows\SysWOW64\RegSvr32.exe /s ConnectorConfiguration.wsc
   C:\Windows\SysWOW64\RegSvr32.exe /s CustomCollector.wsc
   C:\Windows\SysWOW64\RegSvr32.exe /s CustomTask1.wsc
   C:\Windows\SysWOW64\RegSvr32.exe /s CustomTask2.wsc
   ```

   For Python scripts, you simply run the script to register the COM types. The connector configuration script, however, must be registered using **RegSvr32.exe**. For example:

   ```
   python PyConnector.py
   C:\Windows\SysWOW64\RegSvr32.exe PyConnectorConfig.wsc
   ```

2. Register the Script Connector service and metadata.

   ```
   ConfigurableConnector.exe -r
   ```

   The Script Connector service will be registered to log on using the Local System account. You can change the service configuration to log on using a specific user account and password.

At this stage, the Script Connector is deployed and is ready for use.

After deployment

After the Script Connector is deployed and is ready for use, you can create the required configuration objects in Configuration Manager.

1. Restart the IBM Content Collector Configuration Access service service, either from `Services.msc` or using the following command:

   ```
   sc start ibm.ctms.ui
   ```

2. Restart Configuration Manager.
3. Configure your connector, add collectors and tasks to task routes, and configure them.
4. Restart the IBM Content Collector Task Routing Engine service.

Configuring your connector

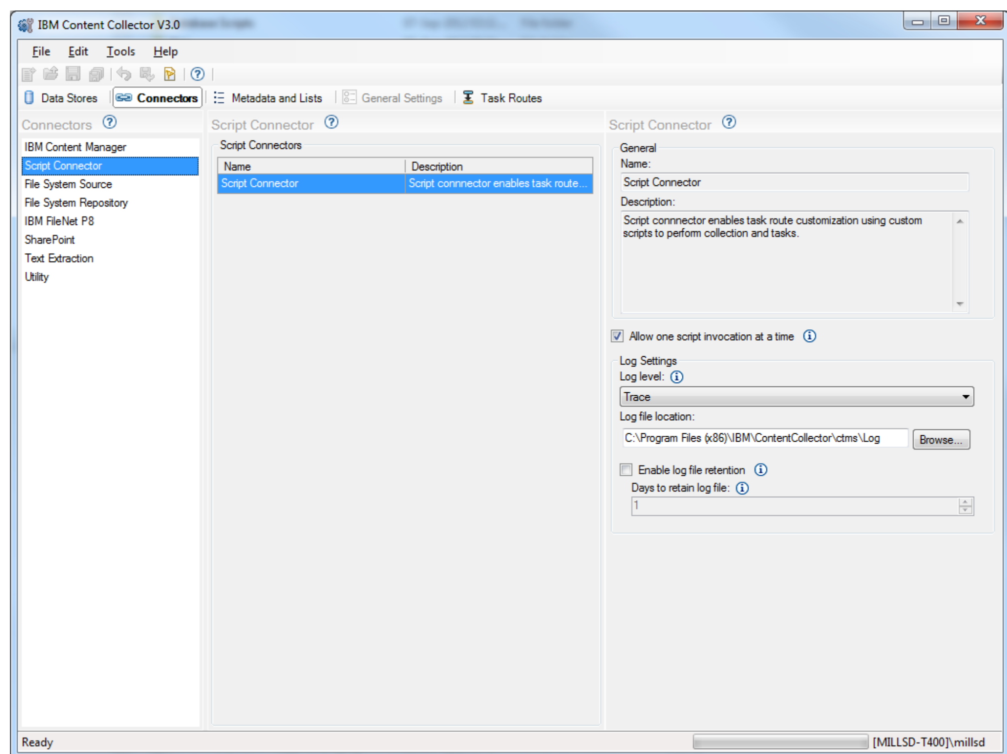
To be able to work with your Script Connector implementation, you must configure the connector and the custom collectors and tasks in Configuration Manager.

Connector configuration

When your Script Connector implementation is installed and registered, it will appear in the **Connectors** configuration window.

The name that appears in the **Connectors** list, the window titles, and the labels are defined by entries in the Script Connector localization file (`ConfigurableConnector.Resources.en-US.xml` on an English language system).

For example, if you install the Sample File Processing Connector sample, the Script Connector configuration window looks like this:



The connector name and description are populated from entries in the localization file during connector registration and are not editable. Log settings (**Log file location**, **Log level**, and **Log file retention**) can be configured by the user.

The option **Allow one script invocation at a time** was added in IBM Content Collector 3.0.0 Fix Pack 2. By default, this option is not selected. Leave it deselected if your script engine (and any third-party COM libraries that your script calls) fully supports multithreaded operation. If you select this option, the Script

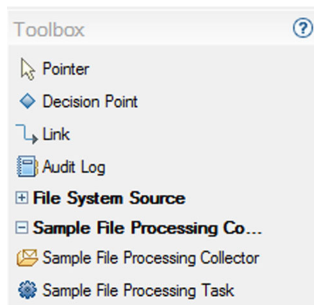
© Copyright IBM Corp. 2008, 2012

Connector ensures that scripts are called sequentially and, if necessary, queues calls. When you use Python scripts, select this option.

The Script Connector toolbox

After registration, the collector and tasks provided by your Script Connector implementation will be available in the Task Route Designer toolbox for adding to a task route.

The connector, collector, and task names will be displayed using the corresponding entries in the localization file, and will default to the registered IDs if the localization entries are not available.



Task configuration

You must configure any Script Connector task that you add to a task route.

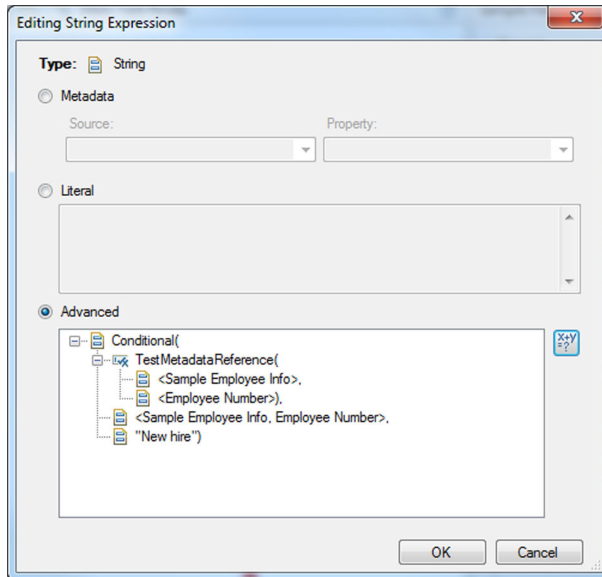
When you drop a Script Connector task onto your task route, the task node is given a default name and description. You can rename the task and update the description.

The **Task configuration** section contains a multiline text box to specify configuration parameters that do not change between calls to the task. The way that this is used is entirely dependent on your implementation and is not validated by Configuration Manager. In this example, the entry for **Configuration string** in the localization file has not been updated, so you must know the expected configuration format.

The **Input metadata mappings** section is used to specify the input parameters that are sent to your task when it is called. The property names and data types specified here must match the names and data types that your task expects. To make this simpler for the user, you can use the localization file to provide a tooltip that documents the expected parameter names and data types.

As well as specifying the property name and value, you must also provide a property mapping expression to provide a value.

One common problem with metadata mappings is the case where a specific metadata property is not available at the point where it is referenced by a task. To avoid errors in your task script caused by missing metadata, select the **Advanced** option when configuring property mappings.



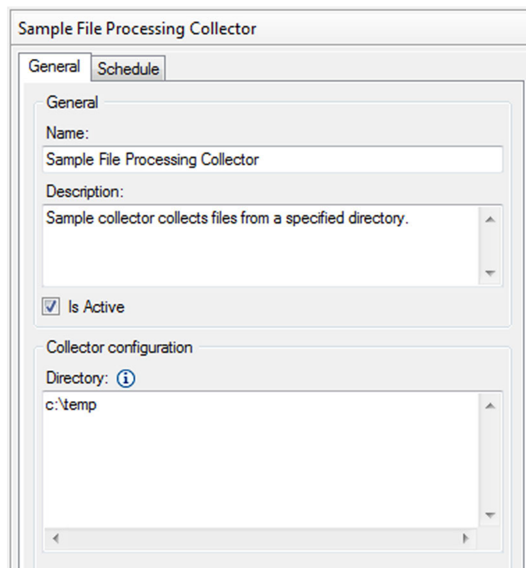
Create an expression that uses the TestMetadataReference() function to test for the existence of the property, and supply a default value if the property has not been set.

You can also test for the existence of a metadata property in your script code.

Collector configuration

You must configure any Script Connector collector that you add to a task route.

When you drop a Script Connector collector onto your task route, the collector node is also given a default name and description based on entries in the localization file. You are free to rename the collector and update the description.



The **Collector configuration** section also contains a multi-line text box for specifying collector configuration. As with tasks, the way that this is used is entirely dependent on your implementation and is not validated by Configuration Manager.

Using the localization file, you can update the label and tooltip to describe the configuration expected by your collector. In this example, the label has been localized to show that the collector expects a directory name.

Before you can use the collector, you also need to configure the collection schedule in the schedule tab. The schedule defaults to **Run always**.

Troubleshooting configuration issues

You can troubleshoot common problems that you might encounter when configuring a Script Connector in Configuration Manager.

The Script Connector does not appear in Configuration Manager

Symptoms

The list of connectors in the Connectors View in Configuration Manager does not include the Script Connector.

Resolving the problem

Ensure that you have deployed and registered at least one Script Connector task or collector.

Restart the IBM Content Collector Configuration Access service and restart Configuration Manager.

A new task or collector does not appear in Configuration Manager

Symptoms

A new Script Connector task or collector is not available in the Configuration Manager toolbox.

Resolving the problem

Ensure that you have deployed and registered your new scripts, ADF file changes, and localization changes.

If you modified the registration script, unregister and register the registration script's .wsc file, and rerun the connector registration.

Restart the IBM Content Collector Configuration Access service and restart Configuration Manager.

A task or collector cannot be configured

Symptoms

When you try to edit a Script Connector task or collector in Configuration Manager, you see a plain configuration panel with this message:

```
No configuration required
```

Resolving the problem

Ensure you have deployed the ADF and localization files. Then, check the ADF file and verify that the task ID in the ADF file matches the task ID in the registration script and in the Windows registry.

The Script Connector configuration window displays cryptic labels

Symptoms

The Script Connector configuration window displays cryptic labels with salmon pink backgrounds.

Resolving the problem

Ensure that you have deployed your localization files to the localization folder.

If the only affected control is a check box with the label `ibm.ctms.connector.configurable.ConnectorConfigUI.allowOneTaskInvocationAtATime`, your localization file is missing an entry for a check box that was added in IBM Content Collector 3.0.0 Fix Pack 2. Copy the missing localization entry from the IBM Content Collector 3.0.0.2 template to the deployed localization file.

Restart Configuration Manager.

Debugging connector scripts

Scripts can often be debugged by using the logEvent function to write log messages to a log file or to the Windows Event log. However, many issues will require the use of a debugger to step through the code as it executes.

This is an example for debugging by using the logEvent function:

```
captureUtilApi.logEvent(@LogTrace, "Unix created date " + createdMsSince1970
  + " converted to "  + createdDate.toUTCString());
```

If your script is written in JScript or VBScript, you can debug your code as follows:

1. Ensure you have a compatible debugger installed. Microsoft Visual Studio 2005 or later is compatible. An alternative is the deprecated Microsoft script debugger, `scd10en.exe`. This is very basic but has a very small footprint (670 KB download).
2. Disable the Script Connector service. This prevents the IBM Content Collector Task Routing Engine service from starting a second instance of the Script Connector as a service when you have already launched the Script Connector from the command line.
3. Ensure the debug attribute on the component processing instruction in your .wsc file is set to true. For example:

   ```
   <?xml version='1.0'?>
   <component>

       <!-- Enable debugging -->
       <?component error='true' debug='true'?>

   </component>
   ```

4. Place a debugger statement in your script code.

   ```
   // This is the method called via the .wsc registration when the task is invoked
   /// \brief Passes each task input object of task input array to ProcessTaskInput
   /// \param comTaskInputs    Task inputs as SafeArray
   /// \param captureUtilApi   Script Connector API COM helper class
   function execute(comTaskInputs, captureUtilApi)
   {
       debugger;

   ...
   ```

5. Use the 32-bit version of `RegSvr32.exe` to re-register the .wsc file. Otherwise you will continue to use the cached registered script, not the modified script containing the debugger statement.
6. Start the Script Connector service in console mode from the command line:

   ```
   ConfigurableConnector.exe -c
   ```

   This launches the Script Connector as a console application using your user credentials.
7. Start the IBM Content Collector Task Routing Engine service. When the IBM Content Collector Task Routing Engine service encounters a call to a collector or task implemented by the Script Connector, it will attempt to establish named-pipe communication with an existing Script Connector process. Because your process is already running, it will handle the request and call the execute method on your task or collector. When execution hits the debugger statement, execution will stop and the debugger will take control.

© Copyright IBM Corp. 2008, 2012

Consider this:

Connectors and the IBM Content Collector Task Routing Engine service run in a multithreaded environment. If you have a collector that submits multiple items during a task debugging session, you can easily become confused as the debugger switches contexts each time thread execution hits one of your break points. For this reason you should disable task routes and collectors other than those you are debugging. To control the debugging process still further, you can disable all collectors, add a new collector to your task route specifically for debugging, which you then configure so that it finds and submits one entity at a time.

The stand-alone script debugger will occasionally fail to attach to the script that is being debugged, usually after starting and stopping the Script Connector several times.

Running the Script Connector from the command line using with the **-c** switch can change connector behavior:

- When the connector is run as a service, the current working directory for the process will be the Windows system directory. However, when the connector is run from the command line with the **-c** switch, the current working directory will be inherited from the parent process, typically the current working directory of `cmd.exe`. This can cause differences in behavior if your script attempts to open files without providing a fully qualified path.
- When the connector is run as a service, the service account is defined in the service configuration. When the connector is run from the command line, the Script Connector will normally run using your user account. This means that the set of resources that are accessible (files, databases, network shares, and so on) might be different. You can work around this by using the **runas** command on the command line to ensure that the Script Connector process runs with the same user account as it would when started as a service.

Additional documentation

Script Connector API documentation

During Script Connector installation, the file `ScriptConnectorAPIDocs.zip` is added to the folder `ctms\ResourceKit\ScriptConnector\Documentation`. This file contains HTML documentation that is generated directly from the C++ source code. You must extract the files before you can use the documentation.

Click the **Start here** shortcut to access the Script Connector API documentation in a two panel view with a navigation pane on the left side and content on the right. Each method is documented and each method description includes a code snippet that shows how to use the method in JScript code.

To help you cross-reference the code samples with the API documentation, the comments and "How to" code snippets in the template scripts clearly indicate the class names of objects that are passed as parameters and created as new ActiveX objects within the code (because JScript is not type safe and the type cannot be inferred from the variable declarations).

© Copyright IBM Corp. 2008, 2012

Windows Script Technologies documentation

Microsoft provides reference and conceptual documentation for all of Microsoft Windows Script Technologies.

Windows Scripting documentation can be obtained from the Microsoft at:

http://www.microsoft.com

Search for *windows script 5.6 documentation*.

When you download the documentation, the downloaded file, `script56.chm`, is a compiled HTML Help file. It documents Windows Scripting Technologies, including JScript, VBScript, Windows Script Host, and the Script Runtime (including WScript.Shell, Scripting.FileSystemObject, and Scripting.Dictionary). Although the documentation is for version 5.6, there have not been any significant changes to the object model in versions 5.7 and 5.8.

Tip: By default, your system might block access to compiled HTML help files. If access is blocked, you might be able to see the index, but not the help file contents (you might see a message `Navigation to the webpage was cancelled`). If your system blocks access, copy `script56.chm` to a location on your local file system, right-click the file, select **Properties**, and click **Unblock** to allow access to the contents.

Samples

TheScript Connector packages includes several samples for custom tasks.

These samples are currently provided:

- "Unix date conversion"
- "File processing connector" on page 50
- "Web service call" on page 51
- "Python Unix date conversion" on page 52

Unix date conversion

This JScript sample demonstrates a simple metadata conversion task. It also demonstrates the use of custom metadata and localization.

The sample is in the `Samples\UnixDateConversion` folder.

Problem:

An external system export process generates XML metadata files that specify the original file creation and modification dates as an integer number representing the number of seconds since January 1st 1970. The FSC Associate Metadata task can read these values as 64-bit integer properties, but IBM Content Collector does not have a built-in mechanism to convert these values to the date representation that is required to provide meaningful creation and modification date metadata in IBM FileNet P8 or IBM Content Manager.

Solution:

The JScript Date class provides a constructor that can perform the required date conversion. This means that the Script Connector can be used to implement a simple task that adds this conversion functionality to IBM Content Collector.

To use the sample:

1. Deploy the UnixDateConversion sample and register the Script Connector. The registration process will add a new system metadata source that contains two date properties. This metadata source is localized and will appear as **Unix date conversion output** to downstream tasks.
2. Create a metadata source for the file system XML. This metadata source will include two integer properties to contain the numeric creation and modification dates read from the XML file.
3. Create a task route that uses the File System Source Connector as the source and any document repository as the target. The target document class should contain two date properties for the creation and modification dates.
4. Configure the **FSC Collector**, specifying the collection sources and schedule.
5. Configure the FSC Associate Metadata task to populate the file system metadata source. This task must assign a value to the numeric file creation and modification dates.
6. Add the Script Connector Unix Date Conversion task to your task route, downstream of the FSC Associate Metadata task. In the **Input property**

© Copyright IBM Corp. 2008, 2012

mappings section, configure two input properties unixCreatedDate and unixModifiedDate of type integer and map these to numeric creation and modification dates in your input metadata.

7. In the target repository connector, map `<Unix date conversion output, Created date>` and `<Unix date conversion output, Modified date>` to appropriate properties in the target document class.

File processing connector

This sample demonstrates how to implement a task that can read input metadata from a file where the file format is not supported by the FSC Associate Metadata task. This task also shows how a task can add entities to an entity set. Furthermore, it demonstrates how to implement a simple collector.

The sample is in the `Samples\CustomMetadataFileHandling` folder.

Problem:

An external process generates files that contain employee records. Employee record files are generated for each data file in the source folder, with the extension .data appended. A typical employee record looks like this:

```
Employee Name: Kirk James T. Employee Number: 5A0780 Date: 03/07/1999
```

The FSC Associate Metadata task cannot parse the employee names, numbers, and hire dates.

Solution:

The JScript Regex class can be used to parse the employee record using a regular expression. Although JScript does not provide a native mechanism for reading files, Scripting.FileSystem from the scripting run time, can be used to read the file contents. Two custom metadata sources are also required, one to contain the employee metadata and one to distinguish between metadata and data files, so that different capture and post processing operations can be performed on the two file types.

To use the sample:

1. Deploy the CustomMetadataFileHandling sample and register the Script Connector. The registration process will add two new system metadata sources:
   - Sample File Type Info that contains a single boolean property Is Metadata File. This property is used to distinguish between data and metadata files.
   - Sample Employee Info that contains employee metadata.
2. Create a new task route using a File System Source Connector task route template.
3. Using the sample collector is optional. If you want to use it, remove the **FSC Collector** and replace it with a **Sample File Processing Collector**. Specify the source folder and collection schedule. The sample collector replicates a small subset of the functionality of the **FSC Collector** and is intended for demonstration purposes only.
4. If you choose not to use the sample collector, configure the **FSC Collector** collection sources and schedule as usual. On the Filter tab, configure a file extension filter to exclude files with an extension of .data.

5. Remove the FSC Associate Metadata task and replace it with the Script Connector Sample File Processing task. In the **Input property mappings** section, configure an input property named filename of type string and map it to `<File, File Path>`.
6. In the target repository connector, map employee metadata to appropriate properties in the target document class. Note that employee metadata is added only to the data file, so you might need to add a condition to your task route that tests the file type using the `<Sample File Type Info, Is Metadata File>` property.

Web service call

This sample demonstrates a basic task that can call an external web service task. It also demonstrates the use of an external COM object (Microsoft.XMLHTTP).

The sample is in the `Samples\WebServiceCall` folder.

For demonstration purposes, this task uses a public web service that accepts anonymous connections (http://www.w3schools.com/webservices/tempconvert.asmx), performing temperature conversion between Celsius and Fahrenheit. Because JScript does not provide direct support for consuming web services, this is an especially naïve and inflexible implementation of a web service client. A more realistic approach would be to package a fully featured web service client as a COM object that is callable from JScript, providing a cleaner interface that isolates the JScript from the implementation details. Alternatively, you could implement a connector in Python, using third-party Python SOAP libraries to call web service methods.

Problem:

A task route needs to invoke an external web service to retrieve additional information, for example a customer record. IBM Content Collector does not currently provide a task or expression that can be used to call an external web service during normal task route processing.

Solution:

The Script Connector can use Microsoft's XML HTTP class to send a SOAP request to an external web service, parse the response and add the data to a custom metadata source.

To use the sample:

1. Deploy the WebServiceCall sample and register the Script Connector. The registration process will add a new system metadata source, WebService.Output that contains two integer properties.
2. Create a new task route using a File System Source Connector task route template. Configure the **FSC Collector**, specifying the collection sources and schedule.
3. Add the Script Connector Web Service task to your task route. In the **Input property mappings** section, configure two input properties celsiusValue and fahrenheitValue of type integer and configure input expressions to provide a value, either based on the input metadata or a literal value.
4. In the target repository task, map `<Web Service output, Celsius converted to fahrenheit>` and `<Web Service output, Fahrenheit converted to celsius>` to

appropriate properties in the target document class. For demonstration purposes, you can use an audit log task instead.

Python Unix date conversion

This sample is functionally equivalent to the JScript Unix date conversion sample described earlier, but is written in Python. In this sample, an integer is converted to a date by using the datetime.utcfromtimestamp() class method in the datetime package.

Windows does not ship with Python installed, so you must download and install a suitable Python distribution for Windows before you can work with this example (see "Prerequisites" on page 5).

This sample demonstrates the usage of the Script Connector Python packages that are installed by the Script Connector installer in IBM Content Collector 3.0.0 Fix Pack 2.

The sample is in the `Samples\PythonUnixDateConversion` folder.

To use the sample:

1. Deploy the sample from the `PythonUnixDateConversion` folder. The process of deploying a Python connector is slightly different than that for deploying a JScript connector.
   a. Close Configuration Manager and stop the IBM Content Collector Configuration Access service.
   b. Copy the files `UnixDateConversion.py` and `UnixDateConversionPy.wsc` to the `ctms` directory.
   c. Copy the localization files from the sample localization folder to the `ctms\localization` folder and copy the .adf files from the sample adf folder to the `ctms\adf` folder.
   d. Register the task and configuration COM objects. Open a command window, and run the following commands from the `ctms` folder:
      ```
      python.exe UnixDateConversion.py
      
      c:\Windows\SysWOW64\regsvr32.exe UnixDateConfiguration.wsc
      
      ConfigurableConnector.exe -r
      ```
   e. Restart the IBM Content Collector Configuration Access service and Configuration Manager.
2. Create a metadata source for the file system XML. This metadata source will include two integer properties to contain the numeric creation and modification dates read from the XML file.
3. Create a task route that uses the File System Source Connector as the source and any document repository as the target. The target document class should contain two date properties for the creation and modification dates.
4. Configure the **FSC Collector**, specifying the collection sources and schedule.
5. Configure the FSC Associate Metadata task to populate the file system metadata source. This task must assign a value to the numeric file creation and modification dates.
6. Add the Script Connector Unix Date Conversion task to your task route, downstream of the FSC Associate Metadata task. In the **Input property mappings** section, configure two input properties unixCreatedDate and

unixModifiedDate of type integer and map these to numeric creation and modification dates in your input metadata.

7. In the target repository connector, map `<Unix date conversion output, Created date>` and `<Unix date conversion output, Modified date>` to appropriate properties in the target document class.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

© Copyright IBM Corp. 2008, 2012

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Deutschland GmbH
Department M358
IBM-Allee 1
71139 Ehningen
Germany

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Microsoft, SharePoint, and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

The Oracle Outside In Technology included herein is subject to a restricted use license and can only be used in conjunction with this application.

Other product and service names might be trademarks of IBM or other companies.

Index

A

adf files 33
API classes
  ICOMCaptureUtilForScripts 11
  ICOMCollectorSubmitProxyForScripts 11
  IComTaskInput 11
  IEntityData 11

C

CaptureUtilApi 22
classid attribute 27
clearTaskOutputs() 15
COM objects 16, 26
COM registration 27, 28
COMException 25
ConfigurableConnector.adf 8
ConfigurableConnector.Resource.en⌂US.xml 8, 34
ConfigurableConnectorDLL.dll 35
configuration library 35
Configuration Manager 33, 39
configuration objects 37
configuration panels
  advanced property mapping 40
  collector 42
  connector 39
  input metadata mapping 40
  task 40
  toolbox 40
connector registration script 28
connector scripts 20
  JScript 11, 12
  overview 11
  Python 11
ConnectorConfiguration.js 8
ConnectorConfiguration.wsc 8
ConnectorTemplate folder 8
CreateObject() 31
CustomCollector.adf 8
CustomCollector.js 8
CustomCollector.wsc 8
customization tasks
  adapting files 10
CustomTask1.adf 8
CustomTask1.js 8
CustomTask1.wsc 8
CustomTask2.adf 8
CustomTask2.js 8
CustomTask2.wsc 8

D

date conversion 49, 52
debugger 45
deployment 36
DescribeMethods() 31
description attribute 27
display names 34

E

entities
  adding metadata 14
entity IDs 24

F

file processing 49
folder contents 5

G

generic user interface 2
GetConnectorName() 31
GetWindowsDisplayName() 31
GuidGenerator.js 5
guids 28

I

IBM Content Collector Script Connector
  getting started 36
IBM Content Collector Task Routing Engine service 45
ibm.ctms.util.LogLevels 24
ICOMCaptureUtilForScripts 11, 14, 16, 18, 22, 25
  LogEvent() 15
ICOMCaptureUtilForScripts.addMethodDescription() 31
ICOMCaptureUtilForScripts.entityError() 16
ICOMCollectorSubmitProxyForScripts 11, 18, 23
IComTaskInput 11
IDispatch 31
IEntityData 11, 14, 22
input parameters
  converting 12, 21
  reading 12, 21
Install Script Connector shortcut 5
installation 5
internalName attribute 27

J

JScript registration script 29

L

labels 34
language attribute 27
limitations 2
localization 34
log levels 15, 24
log messages 15, 24
logEvent() 15

M

metadata
  gathering 14, 22
  returning 14, 22

© Copyright IBM Corp. 2008, 2012

metadata *(continued)*
- submitting 23

method element 27
methods
- clearTaskOutputs() 15
- execute() 16
- getErrorInfo() 25
- GetErrorMessage() 25
- ICOMCaptureUtilForScripts 16, 18
- ICOMCaptureUtilForScripts() 25
- ICOMCollectorSubmitProxyForScripts 18
- logEvent() 15, 24
- ProcessEntity() 16
- send() 15

O

overview 1

P

performance counters 16, 25
prerequisites 5
principles 2
ProcessEntity() 16
progid attribute 27
property bag 14, 22
Python 20
Python registration script 31
Python scripts
- collectors 22
- tasks 20

R

readme.txt file 5
registration
- connector 29, 31
- ConnectorConfiguration.js 29
- ConnectorConfiguration.wsc 29
- JScript 29
- metadata 29, 31
- Python 31
- task 29, 31

registration element 27
RegSvr32.exe 35
removing registry entries 35
restarting services 37

S

samples
- file processing connector 50
- JScript Unix date conversion 49
- Python Unix date conversion 52
- web service call 51

Script Connector APIs
- ICOMCaptureUtilForScripts 47
- ICOMCollectorSubmitProxyForScripts 47
- IComTaskInput 47
- IEntityData 47

Script Connector service 45
script element 27
ScriptConnectorAPIDocs.zip 5
ScriptConnectorConnectorSetup.msi 5
Scripting documentation 48
Scripting.FileSystemObject 16, 26
self.CaptureUtilApi 25
self.CaptureUtilApi.clearTaskOutputs() 25
send() 15
src attribute 27
SubmitStub 23

T

Task Route Designer toolbox 40
task samples 49
template 5, 8
template files
- ConfigurableConnector.adf 8
- ConfigurableConnector.Resource.en_US.xml 8
- ConnectorConfiguration.js 8
- ConnectorConfiguration.wsc 8
- CustomCollector.adf 8
- CustomCollector.js 8
- CustomCollector.wsc 8
- CustomTask1.adf 8
- CustomTask1.js 8
- CustomTask1.wsc 8
- CustomTask2.adf 8
- CustomTask2.js 8
- CustomTask2.wsc 8

test scripts 19
tooltips 34

U

uninstallation 5
Unix date conversion 49
use cases 1

W

web service 49
win32com.server.util.wrap method 31
Windows registry 35
Windows Script Component files 27
Windows Script Technologies 48
Windows scripting
- Scripting.FileSystemObject 16, 26
- WScript.Shell 16, 26

working folder 8
wsc files 27
WScript.Shell 16, 26

Product Number: 5724-V57