Informatica® Managed File Transfer
10.2.3

Custom Task Guide

Informatica Managed File Transfer Custom Task Guide
10.2.3
November 2019

Publication Date: 2020-05-21

Table of Contents

Preface

Use the *Informatica Custom Task Guide* to learn how to create and install custom tasks in Managed File Transfer. Learn about advanced task concepts, such as creating and setting variables, and supporting nested task elements in Managed File Transfer and File Transfer Portal.

Informatica Resources

Informatica provides you with a range of product resources through the Informatica Network and other online portals. Use the resources to get the most from your Informatica products and solutions and to learn from other Informatica users and subject matter experts.

Informatica Network

The Informatica Network is the gateway to many resources, including the Informatica Knowledge Base and Informatica Global Customer Support. To enter the Informatica Network, visit https://network.informatica.com.

As an Informatica Network member, you have the following options:

- Search the Knowledge Base for product resources.
- View product availability information.
- Create and review your support cases.
- Find your local Informatica User Group Network and collaborate with your peers.

Informatica Knowledge Base

Use the Informatica Knowledge Base to find product resources such as how-to articles, best practices, video tutorials, and answers to frequently asked questions.

To search the Knowledge Base, visit https://search.informatica.com. If you have questions, comments, or ideas about the Knowledge Base, contact the Informatica Knowledge Base team at KB_Feedback@informatica.com.

Informatica Documentation

Use the Informatica Documentation Portal to explore an extensive library of documentation for current and recent product releases. To explore the Documentation Portal, visit https://docs.informatica.com.

If you have questions, comments, or ideas about the product documentation, contact the Informatica Documentation team at infa_documentation@informatica.com.

Informatica Product Availability Matrices

Product Availability Matrices (PAMs) indicate the versions of the operating systems, databases, and types of data sources and targets that a product release supports. You can browse the Informatica PAMs at https://network.informatica.com/community/informatica-network/product-availability-matrices.

Informatica Velocity

Informatica Velocity is a collection of tips and best practices developed by Informatica Professional Services and based on real-world experiences from hundreds of data management projects. Informatica Velocity represents the collective knowledge of Informatica consultants who work with organizations around the world to plan, develop, deploy, and maintain successful data management solutions.

You can find Informatica Velocity resources at http://velocity.informatica.com. If you have questions, comments, or ideas about Informatica Velocity, contact Informatica Professional Services at ips@informatica.com.

Informatica Marketplace

The Informatica Marketplace is a forum where you can find solutions that extend and enhance your Informatica implementations. Leverage any of the hundreds of solutions from Informatica developers and partners on the Marketplace to improve your productivity and speed up time to implementation on your projects. You can find the Informatica Marketplace at https://marketplace.informatica.com.

Informatica Global Customer Support

You can contact a Global Support Center by telephone or through the Informatica Network.

To find your local Informatica Global Customer Support telephone number, visit the Informatica website at the following link:
https://www.informatica.com/services-and-training/customer-success-services/contact-us.html.

To find online support resources on the Informatica Network, visit https://network.informatica.com and select the eSupport option.

CHAPTER 1

Introduction Overview

Informatica Managed File Transfer includes over 60 built-in tasks to satisfy most of the business processes needed by organizations for moving, securing, and translating data. However, in some cases your company may need a custom task to provide additional functionality that is not provided in the base package. For example, a custom task could be built to integrate with an internal application, such as an ERP system, through a proprietary interface.

Custom tasks are fully integrated into Managed File Transfer, so they offer several benefits (versus calling external programs) for providing additional functionality in your Projects:

- Custom tasks can be chosen quickly (just as easily as the base tasks) within the Project Designer.
- You can create input and output parameters (with custom labels) for the task, which the user can easily fill out within the Project Designer.
- Custom tasks will be easy to identify and work with in the Project outline.
- Job Managers in Managed File Transfer will be able to more effectively hold or cancel jobs which contain custom tasks.
- Log messages from the custom task will be included into the Project's Job log.

This guide provides Java programmers with the knowledge they need in order to build, package and install custom tasks in Managed File Transfer. Follow these instructions carefully.

Requirements

The following requirements are needed in order to build a custom task:

1. You must have a solid understanding on how tasks and Projects work in Managed File Transfer.
2. You will need access to a Managed File Transfer installation where you can install and test the custom tasks created.
3. You must be a Java programmer with a good understanding of Java 8.0 and XML.
4. You will need an IDE, for example, Eclipse or NetBeans, to create, compile and package a Java project.

   Warning: If a custom task is written incorrectly, it may cause serious problems or performance issues in Managed File Transfer. It is critical that custom tasks are only developed by experienced Java programmers. Custom tasks should be tested thoroughly to ensure that they work properly and do not adversely affect the operation of Managed File Transfer.

   Warning: Your organization will be responsible for any problems (including data loss and downtime) caused by custom tasks. If you are not comfortable with these risks, please do not attempt to build your own custom tasks.

Support

Please note that custom tasks are not covered under standard support by Informatica.

If you need assistance with building custom tasks, please contact your account representative at Informatica. Our team would be happy to talk with you about your requirements and provide you with an estimate on the hours/costs for building a custom task.

CHAPTER 2

Building a Custom Task Overview

In this tutorial, we will walk you through the process of how to create and install a custom task within Managed File Transfer.

The custom task in this tutorial will copy a file from one directory to another.

Step 1. Create and Finalize the Project XML to Define the Task

The first step to create a custom task is to finalize how the task is defined within the project XML. For this simple task, we will use the following XML fragment to define our copy task:

```
<example:copy sourceFile="/srcdir/myfile.txt" destinationDirectory="/destdir" />
```

When you add a custom task to a Project, the XML fragment will be created and added to the project XML.

All tasks have a unique tag name, which is defined by the task's author. For our example task, we have chosen "copy" to be the tag name for our task. All custom tasks must also have a namespace. The namespace can be any string, but ideally should represent the entity that created the custom task. In this example, we have chosen "example" as the namespace for our task.

The following two attributes are also defined in the XML:

- sourceFile – This is the path to the file that you want to copy.
- destinationDirectory – The destination directory to which the source file should be copied.

Step 2. Code the Copy Task

Perform the following tasks to code the copy task:

1. Setup your IDE
2. Create the CopyTask class
3. Add a getTagName method
4. Add the getVersion method
5. Add the getVendor method
6. Create Getter and Setter methods

7. Add user validation
8. Implement the Execute method
9. Add the Cleanup method
10. Implement logging

Configure your IDE

Startup your IDE, such as Eclipse or NetBeans, and create a new Java project. Add the following JAR files from the Managed File Transfer installation to your project class path:

- gamft-<*version number*>.jar
- linoma-commons.jar

The above JAR files can be located in the directory named `lib` under theManaged File Transfer installation directory.

Create the CopyTask Class

Create a new Java class named com.example.CopyTask. In order for the CopyTask to be a valid task, it must be extend the class com.linoma.ga.projects.CustomTask. All tasks must also have a public constructor with TaskContainer as the argument.

```
package com.example;
import com.linoma.ga.projects.CustomTask;
import com.linoma.ga.projects.TaskContainer;
public class CopyTask extends CustomTask {
public CopyTask(TaskContainer parent) {
super(parent);
}
}
```

Add getTagName Method

The next step is to add the abstract method, getTagName. All tasks must indicate the tag name they use in the XML by returning it from this method. In our case we return “example:copy”.

Please note that the red text denotes the new source added to the example.

```
package com.example;
import com.linoma.ga.projects.CustomTask;
import com.linoma.ga.projects.TaskContainer;
public class CopyTask extends CustomTask {
private static final String TAG_NAME = "example:copy";
public CopyTask(TaskContainer parent) {
super(parent);
}
@Override
public String getTagName() {
return TAG_NAME;
}
}
```

Add the getDisplayString Method

Now, implement the abstract method getDisplayString(). All tasks must return a short string representing this task, which is used by the Project Designer in Managed File Transfer. We will return either the tag name (by default) or user defined label that all tasks support.

Please note that the red text denotes the new source added to the example.

```
package com.example;
import com.linoma.ga.projects.CustomTask;
import com.linoma.ga.projects.TaskContainer;
public class CopyTask extends CustomTask {
private static final String TAG_NAME = "example:copy";
public CopyTask(TaskContainer parent) {
super(parent);
label = TAG_NAME;
}
@Override
public String getTagName() {
return TAG_NAME;
}
@Override
public String getDisplayString() {
return label;
}
}
```

Add the getVersion Method

The getVersion method indicates the version of the task. The task author may need to version the task if any changes made to the task are no longer compatible with the previous version of the task. Multiple versions of a task can co-exist in Managed File Transfer. The code below shows the new method that you need to add to your class:

```
@Override
public String getVersion() {
return "1.0";
}
```

Add the getVendor Method

The getVendor method returns the vendor name (or the author name) of the task. Listed below is the code snippet with this new method:

```
@Override
public String getVendor() {
return "Example, Inc.";
}
```

Create the Getter and Setter Methods

Create getter and setter methods for each attribute we have defined in the Copy Task's XML. The setter and getter methods must follow the following naming conventions so the Managed File Transfer Compiler/ Runtime can work with the objects.

The setter methods must meet the following criteria:

- Scope must be public
- Return type must be void
- Must take one argument of type java.lang.String
- Must not be static

- The name of the method must begin with “set”, followed by the attribute name (with the first letter of the attribute name converted to upper case), then followed by the word “attribute”.

The getter method must meet the following criteria:

- Scope must be public
- Return type must be of type java.lang.String
- Must not have any arguments
- Must not be static
- The name of the method must begin with “get”, followed by the attribute name (with the first letter of the attribute name converted to upper case), then followed by the word “attribute”.

Refer to the code shown below in how the sourceFile and destinationDirectory attributes are added:

Please note that the red text denotes the new source added to the example.

```
package com.example;
import com.linoma.ga.projects.CustomTask;
import com.linoma.ga.projects.TaskContainer;
public class CopyTask extends CustomTask {
private static final String TAG_NAME = "example:copy";
private String sourceFileAttribute = null;
private String destinationDirectoryAttribute = null;
public CopyTask(TaskContainer parent) {
super(parent);
label = TAG_NAME;
}
@Override
public String getTagName() {
return TAG_NAME;
}
@Override
public String getDisplayString() {
return label;
}
@Override
public String getVersion() {
return "1.0";
}
@Override
public String getVendor() {
return "Example, Inc.";
}
public String getSourceFileAttribute() {
return sourceFileAttribute;
}
public void setSourceFileAttribute(String sourceFileAttribute) {
this.sourceFileAttribute = sourceFileAttribute;
}
public String getDestinationDirectoryAttribute() {
return destinationDirectoryAttribute;
}
public void setDestinationDirectoryAttribute(
String destinationDirectoryAttribute) {
this.destinationDirectoryAttribute = destinationDirectoryAttribute;
}
}
```

Add User Validation

Validation of user input is also the responsibility of the task author. For example, in our copy task we want to make sure the user has supplied a non-null and non-blank value for both source file and destination directory attributes.

There are two different validation methods a task author must implement.

Implement the validateAttributes Method

The validateAttributes method is already defined in the com.linoma.dpa.Task class which performs validation on common attributes shared by all tasks, such as label, onError, logLevel, and so on. Concrete tasks might have to override this method to perform validation of any task specific attributes. This method is called by the Project Designer.

Implement the validate Method

This method is called by the Project Compiler in Managed File Transfer. All projects go through compilation phase before they execute. The compilation of a project will create an object for each component in the project and call it's validate() method. If any component fails to validate, a compilation error will be reported.

Please note that the red text denotes the new source added to the example.

```
package com.example;
import java.util.ArrayList;
import java.util.List;
import com.linoma.commons.StringUtilities;
import com.linoma.dpa.Message;
import com.linoma.ga.projects.CustomTask;
import com.linoma.ga.projects.TaskContainer;
import com.linoma.dpa.ValidationException;
public class CopyTask extends CustomTask {
private static final String TAG_NAME = "example:copy";
private static final String ATTR_SOURCE_FILE = "sourceFile";
private static final String ATTR_DESTINATION_DIRECTORY = "destinationDirectory";
private String sourceFileAttribute = null;
private String destinationDirectoryAttribute = null;
public CopyTask(TaskContainer parent) {
super(parent);
label = TAG_NAME;
}
@Override
public String getTagName() {
return TAG_NAME;
}
@Override
public String getDisplayString() {
return label;
}
@Override
public String getVersion() {
return "1.0";
}
@Override
public String getVendor() {
return "Example, Inc.";
}
public String getSourceFileAttribute() {
return sourceFileAttribute;
}
public void setSourceFileAttribute(String sourceFileAttribute) {
this.sourceFileAttribute = sourceFileAttribute;
}
public String getDestinationDirectoryAttribute() {
return destinationDirectoryAttribute;
}
public void setDestinationDirectoryAttribute(
String destinationDirectoryAttribute) {
this.destinationDirectoryAttribute = destinationDirectoryAttribute;
}
@Override
public void validateAttributes() throws ValidationException {
// A list to hold any validation errors
List<Message> errors = new ArrayList<Message>();
// Call the validateAttributes of the super class
try {
```

```
super.validateAttributes();
}
catch (ValidationException exp) {
// If the attributes on the super class fail to validate, add the
// errors to the list.
errors.addAll(exp.getMessages());
}
// Perform our task-specific validation.
if (StringUtilities.isEmpty(sourceFileAttribute)) {
errors.add(new Message("Source File is required"));
}
if (StringUtilities.isEmpty(destinationDirectoryAttribute)) {
errors.add(new Message("Destination Directory is required"));
}
if (!errors.isEmpty()) {
throw new ValidationException(errors);
}}
@Override
public void validate() throws ValidationException {
// Simply call the validateAttributes to validate all the attributes.
validateAttributes();
}
}
```

Implement the Execute Method

With validation complete it is time to write the core piece of code, which is the execute method to copy the file. Keep in mind that projects and tasks in Managed File Transfer support variables and expressions. If you decide to support variables and expressions in your task (in one or more attributes), you must write code to evaluate any expression specified for any attributes. In this example, we will support expressions for both source file and destination directory attributes. With the execute method implemented, the code looks like this:

```
package com.example;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.ArrayList;
import java.util.List;
import com.linoma.commons.StringUtilities;
import com.linoma.dpa.ExecutionException;
import com.linoma.dpa.Message;
import com.linoma.ga.projects.CustomTask;
import com.linoma.ga.projects.TaskContainer;
import com.linoma.dpa.ValidationException;
public class CopyTask extends CustomTask {
private static final String TAG_NAME = "example:copy";
private static final String ATTR_SOURCE_FILE = "sourceFile";
private static final String ATTR_DESTINATION_DIRECTORY = "destinationDirectory";
private String sourceFileAttribute = null;
private String destinationDirectoryAttribute = null;
private File sourceFile = null;
private File destinationDirectory = null;
public CopyTask(TaskContainer parent) {
super(parent);
label = TAG_NAME;
}
@Override
public String getTagName() {
return TAG_NAME;
}
@Override
public String getDisplayString() {
```

```
return label;
}
@Override
public String getVersion() {
return "1.0";
}
@Override
public String getVendor() {
return "Example, Inc.";
}
public String getSourceFileAttribute() {
return sourceFileAttribute;
}
public void setSourceFileAttribute(String sourceFileAttribute) {
this.sourceFileAttribute = sourceFileAttribute;
}
public String getDestinationDirectoryAttribute() {
return destinationDirectoryAttribute;
}
public void setDestinationDirectoryAttribute(
String destinationDirectoryAttribute) {
this.destinationDirectoryAttribute = destinationDirectoryAttribute;
}
@Override
public void validateAttributes() throws ValidationException {
// A list to hold any validation errors
List<Message> errors = new ArrayList<Message>();
// Call the validateAttributes of the super class
try {
super.validateAttributes();
}
catch (ValidationException exp) {
// If the attributes on the super class fail to validate, add the
// errors to the list.
errors.addAll(exp.getMessages());
}
// Perform our task-specific validation.
if (StringUtilities.isEmpty(sourceFileAttribute)) {
errors.add(new Message("Source File is required"));
}
if (StringUtilities.isEmpty(destinationDirectoryAttribute)) {
errors.add(new Message("Destination Directory is required"));
}
if (!errors.isEmpty()) {
throw new ValidationException(errors);
}
}
@Override
public void validate() throws ValidationException {
// Simply call the validateAttributes to validate all the attributes.
validateAttributes();
}
public void execute() throws ExecutionException {
internalValidate();
try {
copy();
}
catch (IOException exp) {
throw new ExecutionException(exp.getMessage(), exp);
}
}
private void internalValidate() throws ExecutionException {
String temp = project.expandVariables(sourceFileAttribute);
if (StringUtilities.isEmpty(temp)) {
throw new ExecutionException("Source File is required");
}
sourceFile = new File(temp);
temp = project.expandVariables(destinationDirectoryAttribute);
if (StringUtilities.isEmpty(temp)) {
throw new ExecutionException("Destination Directory is required");
}
```

```
destinationDirectory = new File(temp);
}
private long copy() throws IOException {
InputStream in = null;
OutputStream out = null;
try {
final int bufferSize = 4096;
in = new BufferedInputStream(new FileInputStream(sourceFile),
bufferSize);
out = new BufferedOutputStream(new FileOutputStream(new File(
destinationDirectory, sourceFile.getName())), bufferSize);
byte[] buffer = new byte[bufferSize];
int bytesRead = 0;
long bytesCopied = 0L;
while ((bytesRead = in.read(buffer)) != -1) {
out.write(buffer, 0, bytesRead);
bytesCopied += bytesRead;
}
return bytesCopied;
}
finally {
if (in != null) {
try {
in.close();
}
catch (IOException exp) {
exp.printStackTrace();
}
}
if (out != null) {
try {
out.close();
}
catch (IOException exp) {
exp.printStackTrace();
}
}
}
}
}
```

The execute method is setup to call the internallValidate() method. The purpose of this method is to evaluate any expressions specified for the source file and destination directory attributes. The method then ensures that the evaluated values are still valid for the task; i.e. the values are not null and not empty. If they are valid, the internalValidate method initializes the sourceFile and destinationDirectory instance variables. It then calls the other private method, copy() to actually copy the file.

Add the cleanUp Method

The cleanUp() method must be implemented by all tasks to release any resources. This method is called when the last task in the project has finished executing. Not all tasks need to have code in this method. Only those tasks that hold on to some resources for longer than the task execution should implement this method. In our case, we do not have any resources to release, so we provide a blank implementation as shown below:

```
@Override
public void cleanUp() {
// We have nothing to cleanup.
}
```

At this point, your code should compile without errors.

Implement Logging

Every execution of a Managed File Transfer Project produces a job log that contains important events. Most of the events come from the tasks within the project. The task author makes appropriate decisions on what

should be logged depending on the log level that the task is running under. Logging is done by firing log events. A log listener is registered with every task in the project. The log listener takes on the responsibility of logging the messages to the job log file. Listed below is the code snippet with some log messages from the execute() method:

Please note that the red text denotes the new source added to the example.

```
public void execute() throws ExecutionException {
if (isNormalLogLevel()) {
fireInfo(this, "Copy Task started");
}
internalValidate();
try {
if (isVerboseLogLevel()) {
fireInfo(this, "Copying file " + sourceFile + " t directory "
+ destinationDirectory);
}
long bytesCopied = copy();
if (isVerboseLogLevel()) {
fireInfo(this, bytesCopied + " byte(s) copied");
}
}
catch (IOException exp) {
throw new ExecutionException(exp.getMessage(), exp);
}
if (isNormalLogLevel()) {
fireInfo(this, "Copy Task finished");
}
}
```

Pausing and Canceling Tasks

Managed File Transfer Jobs, or actively running Projects, can be canceled or paused and later resumed. In order to support cancel, pause and resume, the task author must periodically check for any pending cancel or pause requests. As a task author, you need to call the checkForHoldCancel() method periodically from a long running section of code. In our example, we will check for cancel/pause requests in the while loop of the copy method as shown below:

Please note that the red text denotes the new source added to the example.

```
private long copy() throws IOException {
InputStream in = null;
OutputStream out = null;
try {
final int bufferSize = 4096;
in = new BufferedInputStream(new FileInputStream(sourceFile),
bufferSize);
out = new BufferedOutputStream(new FileOutputStream(new File(
destinationDirectory, sourceFile.getName())), bufferSize);
byte[] buffer = new byte[bufferSize];
int bytesRead = 0;
long bytesCopied = 0L;
while ((bytesRead = in.read(buffer)) != -1) {
checkForHoldAndCancel();
out.write(buffer, 0, bytesRead);
bytesCopied += bytesRead;
}
return bytesCopied;
}
finally {
if (in != null) {
try {
in.close();
}
catch (IOException exp) {
exp.printStackTrace();
}
}
```

```
if (out != null) {
try {
out.close();
}
catch (IOException exp) {
exp.printStackTrace();
}
}
}
}
```

The call to the checkForHoldAndCancel will do the work by either cancelling the Job or pausing the Job (thread) if the user requested so. Cancelation of a Job is achieved by throwing an exception, ExecutionCancelledException, which is a RuntimeException. The Project Runtime catches this exception and marks the job as canceled in the audit logs. Pausing is achieved by calling the Thread.wait() on the thread that is executing the Job. The thread will wait until someone requests to resume the Job.

Step 3. Create the BeanInfo XML

The BeanInfo file is a simple XML file that provides additional information needed by the task at project design time, by the Project Designer. The project designer generates a screen for your task based on the information from the bean info. You can define the following information in the bean info:

1. Labels for various attributes of the task. For example, you can define a nice label, “Source File”, for the “sourceFile” attribute.
2. Descriptions or help text for various attributes of the task.
3. The field type for attributes (e.g. text field, text area, file, directory etc.)
4. Define whether an attribute is required or not so the project designer can decorate the field as required using the standard convention (typically a red asterisk after the field label)
5. Any fixed options that an attribute supports, which will be listed in a drop-down field.
6. Indicating the default values for attributes, if any.

   For detailed field and element information that can be used in the bean, see Chapter 4, “Field Types and Elements” on page 23.

   The BeanInfo XML must exist in the same package as the corresponding Task (or any other project component). The name of the file must be same as the class name of the project component, followed by BeanInfo.xml. For our CopyTask, the BeanInfo must be defined in com.example.CopyTaskBeanInfo.xml.

   Listed below is the bean info for our Copy Task:

```
<?xml version="1.0" encoding="UTF-8" ?>
<projectComponent name="CopyTask">
<label>Example Copy Task</label>
<description>A simple task for copying a file from one directory to another. </
description>
<tabPanel name="main">
<tab name="basic">
<attribute name="sourceFile">
<label>Source File</label>
<description>Specify the file to copy. </description>
<fieldType>localOrNetworkFile</fieldType>
<required>true</required>
<cols>60</cols>
</attribute>
<attribute name="destinationDirectory">
<label>Destination Directory</label>
```

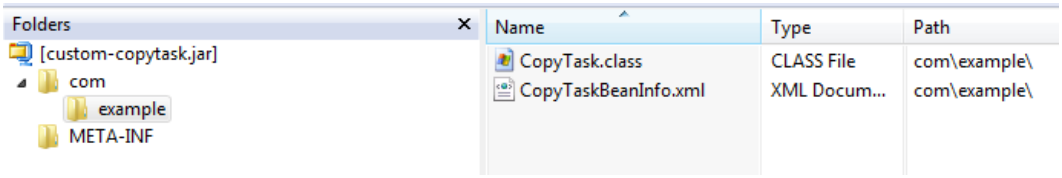
```
<description>Specify the directory to which the source file should be copied. </
description>
<fieldType>localOrNetworkDirectory</fieldType>
<required>true</required>
<cols>60</cols>
</attribute>
</tab>
<tab name="control" />
<tab name="onError" />
</tabPanel>
</projectComponent>
```

The BeanInfo must start with a root element of <projectComponent>. The name of the project component can be anything, but is recommended to keep the same as the class name. Define label and description for the project component using the <label> and <description> tags.

The Project Designer organizes various attributes of a project component into tabs. Tabs are defined inside a <tabPanel>. Tabs will have a name, label and description. For our copy task, we defined the sourceFile and destinationDirectory attributes under the Basic tab. We then indicated that we want to show the standard Control and On Error tabs that are shared by all tasks. We have defined the field type for the source file as localOrNetworkFile, which will render a Browse button next to the text field so the user can browse the local and network files.

Step 4. Packaging Your Task

Packaging your task for distribution and/or deployment involves creating a JAR file of the task code and associated resources. Using your IDE, create a JAR file containing the compiled task class and the BeanInfo xml. A correct jar file for our example should have the CopyTask.class and CopyTaskBeanInfo.xml files.



Step 5. Install the Task

After you create the task, install it into Managed File Transfer. Follow the instructions below to install the task:

1. Shut down Managed File Transfer.
2. Copy the new JAR file to the userdata/lib directory.
3. Start Managed File Transfer.
4. Log in with an Admin User with the Product Administrator role.
5. Click on the System > Custom Tasks menu item.
6. Click on Install Custom Task.
7. Specify the implementation class as com.example.CopyTask.
8. Click Next.

9. Specify an optional description.
10. Click Install.

Note: When installing a custom task in a clustered environment, steps 1-3 must be completed on each additional node in the cluster. Your custom task is installed and ready for use.

Step 6. Test the Copy Task

Perform the following steps to test the Copy task.

1. Create a new project.
2. Navigate to the Add Task screen.
3. Expand the Custom Tasks folder and click on the Copy task we just installed.
4. Configure the task by specifying source file and destination directory and then click on the Execute Project button.
5. Note that the task also has other attributes such as Label, Log Level, On Error, etc. These are inherited from the super class.
6. Select verbose for the log level.
7. Save and Compile the project.
8. Run the project.

   The final step is to add the new task to a project and test it out in Managed File Transfer.

   The project should complete successfully and copy the file you have selected. The execution produces the following job log:

```
6/23/15 7:41:02 AM            INFO      Start Date and Time: 6/23/15 7:41:02 AM
6/23/15 7:41:02 AM            INFO      Job Number: 1326461398353
6/23/15 7:41:02 AM            INFO      Project Name: /Example Copy Test
6/23/15 7:41:02 AM            INFO      Submitted By: root
6/23/15 7:41:02 AM            INFO       Informatica MFT 5.0.0 running on Windows 7
(amd64)
6/23/15 7:41:02 AM            INFO      Executing project 'Example Copy Test'
6/23/15 7:41:02 AM            INFO      Project location: C:\informaticamft\userdata
\projects\Example Copy Test.xml
6/23/15 7:41:02 AM            INFO      Executing module 'Main'
6/23/15 7:41:02 AM            INFO      Copy Task started
6/23/15 7:41:02 AM            INFO      Copying file C:\temp\persons 6.txt to
directory C:\Users\temp2
6/23/15 7:41:02 AM            INFO      1060 byte(s) copied
6/23/15 7:41:02 AM            INFO      Copy Task finished
6/23/15 7:41:02 AM            INFO      Finished module 'Main'
6/23/15 7:41:02 AM            INFO      Finished project 'Example Copy Test'
6/23/15 7:41:02 AM            INFO      End Date and Time: 6/23/15 7:41:02 AM
```

CHAPTER 3

Advanced Concepts Overview

In this tutorial, we will discuss advanced task concepts, such as creating and setting variables, and supporting nested task elements.

Supporting Sub-elements in Tasks

Now let us take a look at how one can support nested elements inside a Task. For example, if we want to enhance our Copy Task to support multiple source files using the standard FileSet component. The XML for our task in the project would look something like this:

```
<example:copy destinationDirectory="/destdir">
<fileset dir="/mydir">
<include pattern="*.pdf" />
</fileset>
</example:copy>
```

For each sub-element in your task, you need to create a new method that creates a new object that is of type ProjectComponent. In essence, each tag in the project XML should correspond to a class (or an object) that implements the com.linoma.dpa.ProjectComponentInterface. The snippet below demonstrates how a nested file set element can be supported by our task:

```
private List<FileSet> filesetElements = null;
public FileSet createFilesetElement() {
if (filesetElements == null) {
filesetElements = new ArrayList<FileSet>(1);
}
FileSet fs = new LocalFileSet(this);
filesetElements.add(fs);
return fs;
}
public void removeFilesetElement(FileSet fileSet) {
filesetElements.remove(fileSet);
}
public List<FileSet> getFilesetElements() {
return filesetElements;
}
```

Note that the FileSet class must implement the ProjectComponentInterface. The FileSet class must also follow the same guidelines for supporting its own attributes and or sub-elements. For example, you need to have setDirAttribute and getDirAttribute methods defined in the FileSet class. You also need to have createIncludeElement which returns another type of ProjectComponent that handles the include tag and its attributes.

During the project execution (in the task's execute method), you can manipulate all the subelements (e.g. filesets) any way you like.

In order for the sub-element to show up to a user in the Project Designer, the Bean Info XML will need to be updated per the red text shown below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<projectComponent name="CopyTask">
<label>Example Copy Task</label>
<description>A simple task for copying a file from one directory to another. </
description>
<tabPanel name="main">
<tab name="basic">
<attribute name="sourceFile">
<label>Source File</label>
<description>Specify the file to copy. </description>
<fieldType>localOrNetworkFile</fieldType>
<required>true</required>
<cols>60</cols>
</attribute>
<attribute name="destinationDirectory">
<label>Destination Directory</label>
<description>Specify the directory to which the source file should be copied. </
description>
<fieldType>localOrNetworkDirectory</fieldType>
<required>true</required>
<cols>60</cols>
</attribute>
</tab>
<tab name="control" />
<tab name="onError" />
</tabPanel>
<subelement name="fileset">
<label>FileSet</label>
<description>Specify a directory of files to copy</description>
</subelement>
</projectComponent>
```

Creating & Setting Variables

It is common for tasks to create or replace a variable in the project so the variable can later be used in subsequent tasks. For example, if you want to generate an output variable that points to the destination file, you need to set the variable using the following line of code:

```
project.createOrReplaceVariable(“variableName”, destinationFile);
```

It is usually a good idea to have an attribute available in the task for the user to specify the name of the output variable.

CHAPTER 4

Field Types and Elements

The BeanInfo file is where you specify additional information needed by the task. The project designer generates a screen for your task based on the information from the bean info. The following input field types and their elements are supported:

textField

A single line text field.

Supported Elements:

- label
- description
- required
- cols
- defaultValue
- option - This will make the text field an editable drop down. Use a new option element for each option you want in the drop down.

  - value

  - description

encryptedField

A password input field with a button to encrypt the plain text.

Supported Elements:

- label
- description
- required
- cols

radioButton

A radio button that is rendered as a dropdown with Yes/No options.

Supported Elements:

- label
- description
- required
- defaultValue
- booleanValidator

textArea

A multi-line text field.

Supported Elements:

- label
- description
- required
- rows
- cols
- defaultValue

file

A single line input field with an ellipsis button. The ellipsis button will launch a file chooser dialog that will allow the user to select a local file only.

Supported Elements:

- label
- description
- required
- cols

directory

A single line input field with an ellipsis button. The ellipsis button will launch a file chooser dialog that will allow the user to select a local directory only.

Supported Elements:

- label
- description
- required
- cols

fileOrDirectory

A single line input field with an ellipsis button. The ellipsis button will launch a file chooser dialog that will allow the user to select a local file or directory.

Supported Elements:

- label
- description
- required
- cols

localOrNetworkFile

A single line input field with an ellipsis button. The ellipsis button will launch a file chooser dialog that will allow the user to select a local or network (SMB) file.

Supported Elements:

- label
- description
- required
- cols

localOrNetworkDirectory

A single line input field with an ellipsis button. The ellipsis button will launch a file chooser dialog that will allow the user to select a local or network (SMB) directory.

Supported Elements:

- label

- description
- required
- cols

localOrNetworkFileOrDirectory

A single line input field with an ellipsis button. The ellipsis button will launch a file chooser dialog that will allow the user to select a local or network (SMB) file or directory.

Supported Elements:

- label
- description
- required
- cols