ORACLE®

Oracle® Documaker

Connector

Developer Guide

12.7.1

Part number: F76382-01

January 2023

ORACLE®

Copyright © 2020, 2021 Oracle and/or its affiliates. All rights reserved.

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software--Restricted Rights (June 1987). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Oracle, JD Edwards, and PeopleSoft are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

CONTENTS

Preface 4
- Audience 4
- Documentation Accessibility 4
- Related Documents 5
- Conventions 5

Introduction 6
- Overview 7
- Moving Documents 9
- Documaker Connector Components 10
- The Development Philosophy 12

Developing Source Components 13
- Overview 14
- Source Component Details 15
- An Example Source 17
- The Source Implementation 18

Developing Destination Components 22
- Overview 23
- Destination Component Details 24
- An Example Destination 25
- The Destination Implementation 26

Developing Periodic Processes 28
- Overview 29
- An Example Periodic Process 30

Developing Phase Listeners 31
- Overview 32
- Phase Listener Component Details 33
- An Example Phase Listener 34
- The Phase Listener Implementation 34
- Standard Source Configuration Properties 37
- Standard Configuration Properties 38
- BatchLoaderSource Implementation 40
- BatchLoaderSystem Implementation 43
- BatchLoaderDocumentData Implementation 53
- FileECMDocument Implementation 56
- FTPDestination Implementation 58
- FTPDestinationSystem Implementation 59
- MockPhaseListener Implementation 76

Preface

This manual contains information a developer can use to create custom applications for transferring documents using Oracle Documaker Connector.

AUDIENCE

This document is intended for developers who are creating new source or destination components for use with Documaker Connector. Experience as a Java developer is necessary, as well as programmer domain-knowledge in the APIs for the document source or destination product to be used.

DOCUMENTATION ACCESSIBILITY

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Oracle Customer Support

If you have any questions about the installation or use of our products, please call +1.800.223.1711 or visit the My Oracle Support website:

http://www.oracle.com/us/support/index.html.

Go to My Oracle Support to find answers in the Oracle Support Knowledge Base, submit, update or review your Service Requests, engage the My Oracle Support Community, download software updates, and tap into Oracle proactive support tools and best practices.

Hearing impaired customers in the U.S. who need to speak with an Oracle Support representative may use a telecommunications relay service (TRS); information about TRS is available at http://www.fcc.gov/cgb/consumerfacts/trs.html, and a list of phone numbers is available at http://www.fcc.gov/cgb/dro/trsphonebk.html. International hearing impaired customers should use the TRS at 1.605.224.1837.

Contact

USA:+1.800.223.1711

Canada: 1.800.668.8921 or +1.905.890.6690

Latin America: 877.767.2253

For other regions including Latin America, Europe, Middle East, Africa, and Asia Pacific regions: Visit- http://www.oracle.com/us/support/contact/index.html.

Follow us

https://blogs.oracle.com/insurance

https://www.facebook.com/oraclefs

https://twitter.com/oraclefs

https://www.linkedin.com/groups?gid=2271161

Related Documents

For more information, refer to the following Oracle resources:

- Documaker Connector Installation Guide
- Documaker Connector Administration Guide
- Java programming resources
- API documentation for your source and destination systems

Conventions

The following text conventions are used in this document:

Convention	Description
bold	Indicates information you enter.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands, URLs, code in examples, and text that appears on the screen.

Chapter 1

Introduction

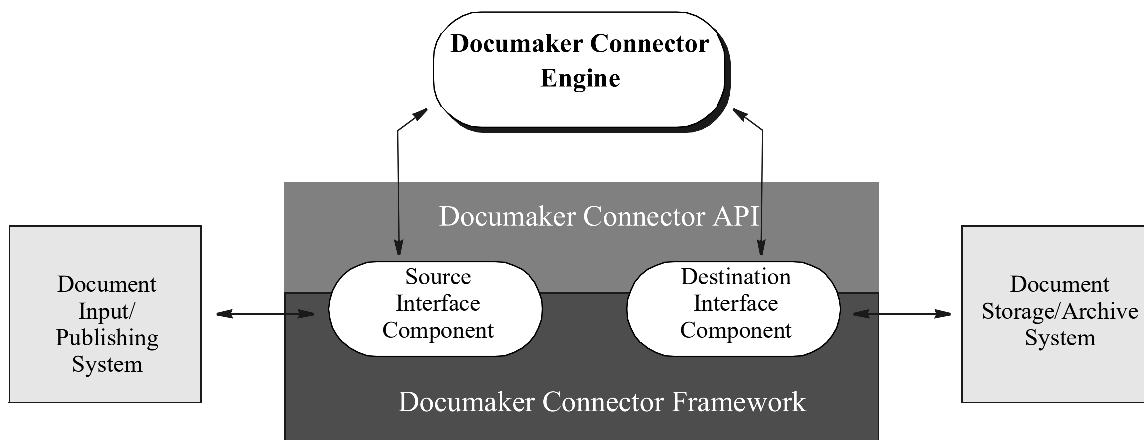
This chapter provides an overview of Documaker Connector and the components that comprise it. This chapter includes these topics:

- *Overview* on page 3
- *Moving Documents* on page 5
- *Documaker Connector Components* on page 6
- *The Development Philosophy* on page 8

OVERVIEW

Moving documents between applications which do not have a common interface has traditionally been a manual and tedious job. You can create batch and/or cron jobs to automate this process, but these tend to be very specific and not reusable.

Documaker Connector provides a pipeline to move documents between applications. The basic model of the Documaker Connector is that there is a *source* of incoming or generated documents and a *destination* where those documents are to be sent or stored. It consists of these parts:



The source and destination interfaces are customizable and this chapter explains how to create new implementations of these components.

For example, the source might be a document automation solution, such as Oracle Documaker, which may run continuously or on demand. Such an application can, for example, push output documents into a file system directory or into database storage with the documents in large-objects (LOB).

The destination could be an electronic content management system such as Oracle WebCenter Content Core Capabilities, previously known as Oracle Universal Content Management (UCM), or another service such as a queue system or web service that leads to an archive or distribution, such as a printing or email.

Documaker Connector eases the fail-safe transfer of the documents from the source to the destination, providing status reporting and restart and recovery if necessary. The interfaces to the source and destination systems are encapsulated in replaceable components which adhere to defined interfaces to the core Documaker Connector engine.

In addition to the stand-alone Documaker Connector application, this same technology is integrated with Documaker Enterprise Edition and Document Factory. Destination components developed to work with Documaker Connector also function as destinations for Document Factory's Archiver.

It is important to note that sources and destinations are not designed to work with each other. Instead, they are each designed to work with Documaker Connector interfaces that are independent of any source or destination. This way, any source should work with any destination, and vice versa.

By implementing a new source or destination component which adheres to these defined interfaces, you can create a new pipeline without domain knowledge of the other systems involved (those for which you *are not* writing the interface components).

Moving Documents

Moving documents from a source application to a destination application via the Documaker Connector API consists of these steps:

1. The client application, such as Documaker Connector or the Archiver, acquires a Source instance from the Documaker ConnectorFramework, optionally specifying a Source and/or Destination identifier.

   ```
   Source aSrc = ConnectorFramework.getInstance().acquireSource();
   ```

2. The client application calls importDocuments on the Source instance.

   ```
   int docCount = aSrc.importDocuments();
   ```

3. The Source instance acquires a list of DocumentData objects, usually by querying the source application.

   For each document acquired in step 3, repeat the next three steps:

4. The Source instance sends a DocumentData object to its associated Destination instance.
5. The Destination instance imports the DocumentData object into the destination application and sets the DocumentData's result code and description based on the import result.
6. Once the destination process is complete for that DocumentData object, control returns to the Source instance where it processes the result of the import attempt and possibly updates the source application or some other system.
7. Finally, control is returned to the client application which may...
   - Return to step 2
   - Release the Source instance and continue processing

     ```
     ConnectorFramework.getInstance().returnSource(aSrc);
     ```

   - Release the Source instance and go back to step 1 and start over

The Documaker Connector client application executes these steps in one of these modes of operation:

Mode	In this mode, this process is
Server	Repeated by each source component instance for as long as the application is running.
Singleton	Executed until each source component replies with an empty document list. Then the application exits.

Note For more information about the configuration and execution of Documaker Connector, see the Documaker Connector Installation Guide.

DOCUMAKER CONNECTOR COMPONENTS

The Documaker Connector is comprised of a number of components that work together to provide its functionality. You can customize all of these components, except the Connector Framework component, to...

- Acquire documents from a new source
- Import documents into a new destination
- Provide additional document metadata to the destination

External to the import process are the periodic process components. These generally provide functionality to maintain Documaker Connector's environment. This includes tasks like cleaning up temporary files, updating services, and so on.

The Configuration Component

Class	oracle.documaker.ecmconnector.connectorapi.data.ConfigurationData

The configuration component is a list of name/value pairs that contain the configuration information for each of the other Documaker Connector components. This component is generally not customized but it can be customized if necessary. You can simply add name/value pairs to handle most situations without the need for specialization.

The Documaker Connector Framework

Class	oracle.documaker.ecmconnector.connectorapi.frameworks.ConnectorFramework

This is the main interface client applications use to interact with Documaker Connector. Along with managing the lifetimes of almost all of the other components, this singleton provides access to the source-to-destination channel, which consists of of a source, a destination, and phase listener components.

Destination Components

Class	oracle.documaker.ecmconnector.connectorapi.Destination

The interface between Documaker Connector and the destination system is implemented via a specialization of the destination component. Based on properties provided by the configuration component, the specific implementations establish connections to the destination system, correctly package documents and metadata, import those packages, and safely close destination system connections.

Document Data Components

Class	oracle.documaker.ecmconnector.connectorapi.data.DocumentData

Each document that passes through Documaker Connector has a set of associated metadata that can describe everything from the author to policy information to security to legal dispensation. All of this information is collected in the Document Data component that accompanies the document instance from the source to the destination components.

Periodic Processes

Class	oracle.documaker.ecmconnector.connectorapi.process.PeriodicProcess

Depending on the generation and importation of documents (the usage of the Documaker Connector), it may happen that there are additional functional steps that need to be taken outside the standard process. The periodic process component provides the framework for the implementation of this functionality as well as its integration into Documaker Connector.

Phase Listeners

Class	oracle.documaker.ecmconnector.connectorapi.PhaseListener

During the importation of a document or list of documents, a number of milestones (phases) occur. Examples of these phases are before the acquisition of the document list or after the destination imports a single document, and so on. The phase listener component gives the developer and system administrator the opportunity to insert additional functionality at each of these points.

Source Components

Class	oracle.documaker.ecmconnector.connectorapi.Source

Source components define the interaction between the Documaker Connector channel and the document generation (source) system. These specializations are responsible for acquiring a list of documents (along with metadata) to import, for sending each of these to the destination component, and for processing the results of the destination's import attempt.

THE DEVELOPMENT PHILOSOPHY

These Documaker Connector components were implemented using the concept of *inversion of control*:

- Source
- Destination
- Phase Listener
- Periodic Process

Their basic functional paths are set, although specializations can provide custom behavior at each step in their processing. Because of this rigor, implementations of these components generally do not need any internal knowledge of any others. The notable exception being periodic processes which are designed to work with particular source or destination implementations.

Chapter 2

Developing Source Components

This chapter contains information you can use to develop source components for use with Oracle Documaker Connector. This chapter includes these topics:

- *Overview* on page 14
- *Source Component Details* on page 15
- *An Example Source* on page 17
- *The Source Implementation* on page 18

OVERVIEW

As described earlier, the source components define the interaction between the Documaker Connector channel and the document generation (source) system. This interaction follows series of steps that perform these tasks:

- Acquire a list of documents
- Individually send those documents to the destination component
- Process the results of each import attempt

While there are several pre-packaged source components, you may want to create a custom version to use with an unsupported document generation facility. When you create a new specialization of the Source class, consider these questions:

- Where are the documents stored and how will their contents be accessed by the source implementation?
- Where is the metadata (the information about, but not part of each document) associated with each document stored?
- What is required to maintain the environment in which each instance will be run?
- How will the results from the destination component be processed?

SOURCE COMPONENT DETAILS

To answer the source development questions, a detailed knowledge of the source component's functional steps is required (or at least highly desired). In the introduction, we saw where the main functions of the source component occur:

- At the beginning of the import request with the acquisition of the document list
- After each destination import attempt with the processing of the document's results

This list describes each activity that takes place during an import request, such as when a client application calls the Source.importDocuments function.

1. Each of the registered PREACQUIREDOCUMENTS phase listeners is executed followed by the Source.preAcquireDocumentList(List<DocumentData> documentList) function. Any custom functionality that needs to execute before the documents are acquired should either be implemented as a phase listener for this step or via the override method.
2. The Source.acquireDocumentList(List<DocumentData> documentList) function is called. This is normally where the source specialization interacts with the document generation system (database, file system, and so on) to acquire the list of documents and their metadata.
3. Each of the registered POSTACQUIREDOCUMENTS phase listeners is executed followed by the Source.postAcquireDocumentList(List<DocumentData> documentList) function.
4. Each of the registered PRESUBMITALLDOCUMENTS phase listeners is executed followed by the Source.preSubmitAllDocuments(List<DocumentData> documentList) function.

   The destination's PREIMPORTALLDOCUMENT phase is executed at this point.
5. A document is sent to the destination for import via the Destination.importSingleDocument(DocumentData documentData) function.
6. Each of the registered PREPROCESSRESULTS phase listeners is executed followed by the Source.preProcessResults(DocumentData documentData) function.
7. The Source.processResults(DocumentData documentData) function is called to process the results of the current document's import attempt.
8. Each of the registered POSTPROCESSRESULTS phase listeners is executed followed by the Source.postProcessResults(DocumentData documentData) function.

   Steps 5-8, are repeated for each document on the list acquired in step 2. Once all of the documents on the list have been through steps 5-8, the destination's PREIMPORTALLDOCUMENT phase is executed.

9. Each of the registered POSTSUBMITALLDOCUMENTS phase listeners is executed followed by the Source.postSubmitAllDocuments(List<DocumentData> documentList) function.
10. The number of documents in the acquisition list is returned as the function exits.

You can introduce custom functionality to the source component in these ways:

- Override one of the methods called from importDocuments
- Configure a phase listener which will execute at the appropriate point in the sequence

AN EXAMPLE SOURCE

Here is a brief description of the requirements for a source implementation that processes batch files generated by the Oracle WebCenter Content BatchLoader application (BatchLoader source component). The only objective is to be able to make the documents (files) from a batch file available to Documaker Connector.

The Batch Loader source reads a single plain text file which contains a sequence of multiline file reference entries. This file is called a *batch file*. Each file reference includes a path to the file and a variable length list of name/value pairs which provide metadata about each document. A sample from the WebCenter Content documentation is this:

```
#This is a comment...
Action=insert
dDocName=Sample1
dDocType=Report
dDocTitle=Title of first document to be checked in
dDocAuthor=sysadmin
dSecurityGroup=Public
primaryFile=sample1.doc
dInDate=5/14/04
<<EOD>>
```

The only supported action for the sample component is *insert*. Each entry must end with this end-of-data marker:

```
<<EOD>>
```

A pound sign (#) indicates a comment, which is ignored when the file is processed. The primaryFile field gives the path to the file to be stored. The other lines are properties for the document record.

The Batch Loader source can instead be configured with a *batch queue file* which is a plain text file that contains the paths to multiple above-described *batch files* listed one per line. This batch queue file can be appended to while Documaker Connector is running to add batch files to the queue. This allows the Batch Loader source to be usable with a Documaker Connector running as a service or daemon in *Server* mode.

Another process can add new lines to the end of the batch queue file. The Batch Loader source will read and remove the first line of the batch queue file and use it as a path to a batch file to process. When all the files in that batch file have been processed, the Batch Loader returns to the batch queue file for the next line. When the file is empty, the source returns an empty document list to Documaker Connector. If Documaker Connector is running in Server mode, the source will eventually be called again after a polling interval expires. Otherwise, the source instance is terminated.

The BatchLoader source implementation includes these classes:

Class Name	Extends Class	Description
BatchLoaderSource	Source	Uses the BatchLoaderSystem class to read a list of incoming files and translates that list into an acceptable source output list for the engine. Also, processes a list which is returned with result statuses.

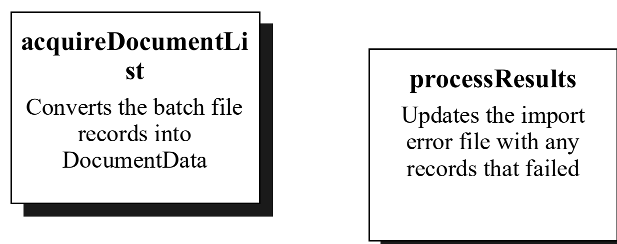
Class Name	Extends Class	Description
BatchLoaderSystem	(none)	Internal singleton class which handles maintaining state and reading the configuration and incoming batch input/ definition files.
BatchLoaderDocumentData	DocumentData	An extension of DocumentData that remembers the document's original batch file name.
BatchLoaderProcess	PeriodicProcess	Implements the periodic cleanup of processed document files.
BatchLoaderSourceExceptio n	SourceException	Specializes the exception thrown on errors by some BatchLoader source classes.

The Source Implementation

The Source class acquires lists of documents to be processed by the destination component. It is the conduit between the document generation application and the configured Destination class. As such, it has a number of methods that need to be specialized to properly interact with the source application:

- The class constructor accepts a ConfigurationData object and a String identifier
- The acquireDocumentList method is called to query the Source for any documents waiting to be imported
- The processResults method is called after each set of documents have been imported
- The repair method is called periodically after an error has occurred in the Source instance to give it an opportunity to recover

The BatchLoaderSource implementation is shown here:



To see how this version of source was completed, let’s look at each method in more detail.

For the BatchLoader source implementation, you need to read from batch files. You also need to remember where in the batch you are between requests for document lists. A batch file could have records for a million documents. You cannot process all of them at once. If a batch queue file is being used to submit multiple batch files, you will need to span batch files if necessary and you will need to know which files have been processed.

For this implementation, you will create a singleton to manage these requirements and have the Source specialization interact with this class instead of the batch and other files directly. This simplifies the configuration of the BatchLoaderSource class. It reads one property (batchloader.source.max.records) and configures the singleton's instance.

The BatchLoaderSource's constructor

```
public BatchLoaderSource(ConfigurationData configData,
                         String sourceId) throws SourceException {
   super(configData, sourceId);
   maxBatchSize = Integer.parseInt(configData.getProperty(
               "batchloader.source.max.records", "1"));
    if (maxBatchSize < 1)
        maxBatchSize = 1;

    if (batchSystem == null) {
        batchSystem = BatchLoaderSystem.getInstance();
        batchSystem.configure(configData);
    }
}
```

The BatchLoaderSystem's configure method is more complicated as it:

- Reads and interprets more of the configuration data and stores it internally
- Checks for a configured batch file and if none, checks for a configured batch queue file
- Validates the path to the directory containing the various error files
- Opens the initial batch file, either the one configured or removes the first entry from the batch queue file and opens that one

As mentioned before, acquireDocumentList converts the batch records that contain each document's details into DocumentData objects that the Destination instance can process. The source of these records in this example is the BatchLoaderSystem class, which is in charge of managing the batch files.

The BatchLoaderSource's acquireDocumentList method

```
public void acquireDocumentList(List<DocumentData> documentList)
throws SourceException {
    Vector<Properties> fileRecords =
                     batchSystem.acquireBatchFileList(maxBatchSize);
    String batchId = UUID.randomUUID().toString();

    for (Properties fileRecord : fileRecords) {
        String primaryFile = fileRecord.getProperty("primaryFile");
        String batchFileName =

fileRecord.getProperty(BatchLoaderSystem.BATCHFILENAME);

        if (primaryFile == null || primaryFile.isEmpty() == true) {
            batchSystem.writeBadRecordToFile(fileRecord,
                 "\"primaryFile\" property is missing", null, false);
            continue;
        }

        ECMDocument ecmDoc = new FileECMDocument(primaryFile);
        DocumentData metaData =
```

```
                new BatchLoaderDocumentData(batchId, ecmDoc,
batchFileName);
        Enumeration keys = fileRecord.keys();

        try {
            while (keys.hasMoreElements()) {
                String key = (String)keys.nextElement();

                if ((key.equalsIgnoreCase("Action") == true) ||
                    (key.equals(BatchLoaderSystem.BATCHFILENAME) ==
true))
                        continue;

                if (key.toLowerCase().indexOf("date") >= 0)
                    metaData.put(new DataIdentifier(key, true),
                            dateFormat.format(batchDateFormat.parse(
                                    fileRecord.getProperty(key))));
                else
                    metaData.put(new DataIdentifier(key),
                                    fileRecord.getProperty(key));
            }

            documentList.add(metaData);
        } catch (Exception genErr) {
            batchSystem.writeBadRecordToFile(fileRecord,
                                "Invalid date format", null, false);
        }
    }
}
```

There are a number of items referenced in this method. The DocumentData class contains the following:

- Each document's information
- A reference to the document's content wrapper class (the ECMDocument implementation)
- After the import is complete, the result code and description

Since the destination component may processes date metadata differently from other types, the DataIdentifier class contains the name of each piece of metadata and whether it is a date or not.

Note The system assumes that if a property has the text *date* in its name, it is a date.

Looking through the code, you can see the tasks most source specializations need to complete.

- Get the list of document data from the source system or application (typically in that system's format)
- Determine the batch's identifier (this often comes from the source system)
- For each item from the source system...
    - Create an ECMDocument instance referencing the document's contents
    - Create a DocumentData instance to hold the document's metadata
    - Copy the document data into the DocumentData instance

- Add the DocumentData instance to the DocumentData list

Once the Destination has processed each document, the list is returned to the source implementation as a parameter to the processResults method call. This method lets each implementation update the source system with the results of the import attempts.

The BatchLoaderSource either places each imported record in a list to be deleted or writes the record to an error file (with any error message) based on the record's result code.

The BatchLoaderSource's processResults method

```
public void processResults(List<DocumentData> documentList) throws
SourceException {
    for (DocumentData docData : documentList) {
        if (docData.getResultCode() == DocumentData.IMPORTED)
            batchSystem.addDeletableFile(
                         (String)docData.getProperty("primaryFile"));
        else
            batchSystem.writeImportErrorRecordToFile(docData);
    }
}
```

Here are the additional methods you can override by a specialization. These methods are not used in the BatchLoader source example:

Method	Description
repair	Called periodically to provide opportunity for the Source instance to correct the issues that led to it being placed in the invalid source pool;
cleanUp	Called by the Documaker ConnectorFramework when the system is being closed. This provides an opportunity for the Source instance to clean up any system resources it has acquired and/or perform any required termination functions. For example, you could use this to reestablish a connection with the source system or database that was taken off-line.

Chapter 3

Developing Destination Components

This chapter contains information you can use to develop destination components for use with Oracle Documaker Connector. This chapter includes these topics:

- *Overview* on page 23
- *Destination Component Details* on page 24
- *An Example Destination* on page 25
- *Configuring the FTP destination* on page 25
- *The Destination Implementation* on page 26

OVERVIEW

Destination components define the interaction between the Documaker Connector channel and the document retention (destination) system. Typically the destination is a traditional content management system, but it can be most anything.

Destination implementations maintain the connection to the external system and convert the document and metadata into the appropriate format. Documaker Connection includes a number of already-created destination components and you can easily create a custom destination if necessary.

The questions you need to answer when creating a new destination implementation mirror those for the source:

- Where will the documents be stored and how will their contents be transmitted?
- Which metadata items associated with each document are required and which are optional?
- What is required to maintain the environment in which each instance will be run?
- How will the results from the import attempt be determined?

Destination Component Details

Just as you need a detailed knowledge of the source component to develop effective custom implementations, you need a detailed knowledge of the destination component to develop specializations that answer the destination questions. This list describes each activity that takes place during an importSingleDocument request:

1. Each of the registered PREIMPORTALLDOCUMENT phase listeners is executed followed by the Destination.preImportAllDocument(List<DocumentData> documentList) function. This functionality is called as part of the source component processing.
2. Each of the registered PREIMPORTDOCUMENT phase listeners is executed followed by the Destination.preImportDocument(DocumentData documentData) function.
3. All of the registered IMPORTDOCUMENT phase listeners is executed followed by the Destination.importDocument(DocumentData documentData) function.
4. Each of the registered POSTIMPORTDOCUMENT phase listeners is executed followed by the Destination.postImportDocument(DocumentData documentData) function.

   Steps 2-4 are repeated for each document in the list acquired by the source component.
5. Each of the registered POSTIMPORTALLDOCUMENT phase listeners is executed followed by the Destination.postImportAllDocument(List<DocumentData> documentList) function. This functionality is called as part of the source component processing.

Just as with the source component, you can introduce custom functionality to the destination component in one of these ways:

- Override one of the methods called
- Configure a phase listener which will execute at the appropriate point in the sequence

AN EXAMPLE DESTINATION

The FTP destination opens a session to the configured destination site and uploads each document into a subdirectory of the FTP base directory named after the document's batch name. The destination is configured with the properties file and contains these properties:

- `destination.name`
- `destination.ftp.server`
- `destination.ftp.username`
- `destination.ftp.password`
- `destination.ftp.port`
- `destination.ftp.base.directory`
- `destination.ftp.side.base`
- `destination.ftp.template.path`

The destination implementation for FTP includes these classes:

Class name	Description
FTPDestinationSyste m	Internal singleton class which has all the real workhorse code in it.
FTPDestination	Interfaces destination calls from the engine to the internal FTPDestinationSystem object to process each document.

Configuring the FTP destination

The FTP destination is used to write out print streams to a specified accessible FTP location. The FTP destination can be configured to generate an index file for each batch processed, containing the indexing information needed for importing the print stream into an indexed content management systems. The source of the index file format is called a template and the source for the index file data are the values from the columns associated with the batch within the Assembly Line schema.

Use these properties to define the FTP destination settings.

Property	Description
Group Name = Configuration	
destination.name	oracle.documaker.ecmconnector.ftpdestination.FTPDestination is the name of the class used for the destination.Do not change the name.
destination.ftp.server	Name of the ftp server, listed by server name only. Do not include ftp:// prefix.
destination.ftp.port	Port of the ftp server.

Property	Description
destination.ftp.usernam e	User name for FTP access.
destination.ftp.password	Password for FTP access.
destination.ftp.base.dire ctory	Location for output files. Files are placed in individual directories within this location,one per batch by default. Default value of Files being placed in individual directories within the base directory is{BCHS.BCH_ID}
destination.ftp.side.base	Provides a secondary location if you wish to store index files separate from the FTP base directory where the print files are stored. If not specified, the destination.ftp.base.directory will be used
	Note: Any variables referenced in the indexfiletemplate.xml (or referenced template) must be defined as properties in the Mappings.

Property	Description
Group Name = Mappings	
destination.ftp.name.pa ttern	Provides the file naming structure for the printed output on written to the FTP server. For example {PUBID}.pdf would generate an output file with the PUB ID value with a .pdf extension. Default value is {PUBS.PUB_ID}.{PUBS.PUBPRTTYPE}
destination.ftp.side.na me.pattern	Provides the file naming structure for the index file. If blank, the index file will not be generated. Default value is {PUBS.PUB_ID}_index.xml
destination.ftp.subdirec tory.pattern	Provides a location/naming convention if you wish to have further subdirectories within the base directory. Use “.” to store files in the base directory. Default value is {BCHS.BCH_ID}
destination.ftp.template	Provides the content of the index file template. Use this option instead of the destination.ftp.template.path in the Configuration to list out the content of the template. Default value is {BCHS.BCH_ID},{PUBS.PUB_ID},{JOBS.JOB_ID} . Use either one of the options.
Variable names to use in the indexfiletemplate data	Tablename.column name from the dmkr_asline schema populate the variables listed in the indexfiletemplate.xml referenced by the destination.ftp.template.path Configuration option.

The Destination Implementation

Destination specializations process import requests from Source instances. They accomplish this by overriding a number of methods in the Destination class.

- The class constructor accepts a ConfigurationData object and a String identifier
- The importDocument method which imports a single document. This is called by the importDocuments method for each item in the DocumentData list

- The repair method is called periodically after an error has occurred in the Destination instance to give it an opportunity to recover

The FTPDestination has little functionality of its own. Most of what it does is to proxy requests through to the FTPDestinationSystem singleton. Because of this, you should understand how FTPDestinationSystem is used to process import requests.

The method called from the FTPDestination's importDocument function is *uploadDocumentContents*. This function performs these tasks:

- Checks the connection to the FTP server
- Moves to the batch directory (after creating it if necessary)
- Uploads the file using an input stream

This code shows how the FTPDestinationSystem gets access to the document contents.

```
// Store file using provided file name
ECMInputStream ecmInput = contents.acquireStream();
InputStream in = ecmInput.getInputStream();

ftpClient.storeFile(ecmInput.getFileName(), in);
```

To free the resources you have acquired, execute the following code in the method's finally block:

```
try { contents.release(); }
catch(Exception genErr) { /* ignore */ }
```

Where *contents* is a ECMDocument reference from the current DocumentData instance. From this, you get the ECMInputStream reference (ecmInput) that provides the java.io.InputStream reference to the document contents and the file name to be associated with them. These are all that are necessary for the FTP interface to store the document. Other external destination systems may require more from the list of document data.

Chapter 4

Developing Periodic Processes

This chapter contains information you can use to create periodic processes for use with Oracle Documaker Connector. This chapter includes these topics:

- *Overview* on page 29
- *An Example Periodic Process* on page 30

OVERVIEW

There are times when your implementation of Documaker Connector requires functionality that is associated with the import process, but is not executed as part of it. In this situation, you can define a periodic process specialization to satisfy the requirement.

Periodic processes are analogous to Java's *runnable implementations*. They do, in fact, indirectly implement this interface. To specialize the periodic process component, override the *process* method with the functionality you want to implement each time the component is executed.

Then, in the component's configuration, specify the number of and the wait time between iterations, as described in the Documaker Connector Installation Guide.

AN EXAMPLE PERIODIC PROCESS

The BatchLoaderSource implementation created earlier has an additional requirement, to delete the files that were successfully imported. The list of these files is managed by the batch system and you can access this list via its acquireDeletableFiles method call.

The following code gets the current list (all documents added since the last execution) and attempts to delete each one if it exists. If the file does not exist, the periodic process logs an error.

```
public void process() {
    Vector<String> filePaths = batchSystem.acquireDeletableFiles();

    for (String filePath : filePaths) {
        File deletableFile = new File(filePath);

        if (deletableFile.exists() == true)
            deletableFile.delete();
        else
            cleanUpErrors.println(
                 "Deletable file [" + filePath + "] does not exist.");
    }
}
```

Chapter 5

Developing Phase Listeners

This chapter contains information you can use to create phase listener components for use with Oracle Documaker Connector. This chapter includes these topics:

- *Overview* on page 32
- *Phase Listener Component Details* on page 33
- *An Example Phase Listener* on page 34
- *The Phase Listener Implementation* on page 34

OVERVIEW

Phase listener components provide a configurable way to add functionality to a Documaker Connector channel regardless of the source and/or destination instances. When requesting the Documaker Connector channel from the ConnectorFramework, the client application can provide a list of identifiers for the desired phase listener components as well. These will be executed at their specified points in the import process.

As with the source and destination components, there are some questions that need to be answered when developing a new Phase Listener:

- Is the functionality applicable to more than one channel configuration? Does it make more sense as a new Source or Destination instead?
- At what point or points during the import process should the Phase Listener execute? Will it need the entire list of documents or just a single one?

Phase Listener Component Details

There are two primary methods any specialization of the phase listener component must implement/override:

- getActivePhaseIdentifiers()
- One or more of the execute() methods, based on the phases in which the listener is active

This table lists and describes the phases of the import process.

Name	execute() version	Context	Description
PREACQUIREDOCUMENTS	document list	Source	Executed before the Source component's acquisition of the list of documents for import.
POSTACQUIREDOCUMENTS	document list	Source	Executed after the Source component's acquisition of the list of documents for import.
PRESUBMITALLDOCUMENTS	document list	Source	Executed before sending the list of documents to the Destination for processing.
PREIMPORTALLDOCUMENTS	document list	Destination	Executed before receiving the list of documents from the Source for processing.
PREIMPORTDOCUMENT	single document	Destination	Executed before importing a single document from the list.
POSTIMPORTDOCUMENT	single document	Destination	Executed after importing a single document from the list.
PREPROCESSRESULTS	single document	Source	Executed after importing a single document from the list but before the results of that import attempt are processed.
POSTPROCESSRESULTS	single document	Source	Executed after the results from a single document's import attempt are processed.
POSTIMPORTALLDOCUMENTS	document list	Destination	Executed after the entire list of documents have been imported.
POSTSUBMITALLDOCUMENTS	document list	Source	Executed after sending the entire list of documents to the Destination for import.

The active phases for the listener determines if the execute method will have access to the entire list of documents or only individual documents.

An Example Phase Listener

The Documaker Connector library contains a number of *mock* components for testing and sample implementations. The MockPhaseListener is a simple specialization of the Phase Listener component that may be used as a starting point for new implementations. It is configured with a list of phases that specify when it is active and its execute methods simply display the name of the phase for when they are called.

The Phase Listener implementation contains these classes:

Class name	Description
MockPhaseListener	The specialization of the PhaseListener base class that implements the getActivePhaseIdentifiers() method and overrides each of the execute(…) methods.

The Phase Listener Implementation

Like the other components, phase listener specializations get and process their configurations via their constructors. Generally, this is also where the list of active phases is created, so calls to getActivePhaseIdentifiers() only need to return the pre-constructed list.

The MockPhaseListener checks for and reads the phaselistener.mock.phase.list property, tokenizes it based on the comma (,) character, and adds the phase identifiers to its internal array.

```
    public MockPhaseListener(ConfigurationData configurationData,
                             String phaseId) throws PhaseException {
        super(configurationData, phaseId);

        activePhases = new ArrayList<Phases>();

        String phaseListString =
            configurationData.getPropertyWithModifier(PHASE_LIST,
phaseId);

        if (phaseListString != null) {
            String[] phaseList = phaseListString.split(",");

            for (String phaseStr : phaseList) {
                Phases phaseInst = Phases.valueOf(phaseStr.trim());

                if (phaseInst != null) {
                    activePhases.add(phaseInst);
                }
                else
                logger.debug("Invalid phase id string - " + phaseStr);
            }
        } else
          throw new PhaseException("A list of phases must be provided
for proper configuration of the MockPhaseListener.",
                                            ExceptionCodes.CNT0500000002,
                                          new Object[] { "A list of phases
must be provided for proper configuration of the MockPhaseListener."
});
    }
```

The execute(…) methods for specializing a phase listener component are normally not all overridden unless the active phases require this. When more than one phase causes the execute(…) method to fire, the implementation can use the provided phase identifier to specify which functionality should be executed. The MockPhaseListener's execute methods just display the name of the phase for which they are being fired.

```
    public void execute(List<DocumentData> documentList,
                        Phases phaseId) throws PhaseException {
      logger.debug("Executing phase (list) - " + phaseId.toString());
    }

    public void execute(DocumentData documentData,
                        Phases phaseId) throws PhaseException {
        logger.debug("Executing phase (single) - " +
phaseId.toString());
    }
```

There is a bit of leeway in the code you can add to the execute(..) methods. For example, the PDFBurster phase listener (included with Documaker Connector), removes PDF documents found in the list and replaces them with a series of new documents based on the Documaker forms found in the original. After the documents are imported, the original document is returned to the list and updated with the results of all the *PDFlet* import attempts.

Appendix A

Configuration Properties

This appendix documents the various configuration properties used by Documaker Connector. These include the following topics:

- *Standard Source Configuration Properties* on page 37
- *Standard Configuration Properties* on page 38

STANDARD SOURCE CONFIGURATION PROPERTIES

The following list of properties is available to any source implementation. Any additional properties needed by the implementation should be documented in the implementation's guide.

Name	Description	Default
source.name	The fully qualified name of the source implementation	-
source.administration.name	The fully qualified name of the SourceAdministration implementation	-
source.count	The number of instances of the source implementation to create	1
source.max.records	The maximum number of documents to return when the getMetaData method is called	1
source.administration.cleanupwait	The number of seconds between source system cleanup calls.	10
source.import.delete.imported.files	Delete the imported files from the file system.	True
source.import.delete.imported.files.count	The number of files to be deleted during each cleanUp call.	50
source.persistence.path	The directory path to contain any result data that cannot be updated in the source system.	-

STANDARD CONFIGURATION PROPERTIES

The following list of properties is available to any destination implementation. any additional properties needed by the implementation should be documented in the implementation's guide.

Name	Description	Default
destination.name	The fully qualified name of the destination implementation	-
destination.administration.name	The fully qualified name of the DestinationAdministration implementation	-
destination.active.wait	The number of seconds to wait for the destination system to return as active	10

Appendix B

Sample Implementations

This appendix provides the following sample implementations:

- *BatchLoaderSource Implementation* on page 40
- *BatchLoaderSystem Implementation* on page 43
- *BatchLoaderDocumentData Implementation* on page 53
- *FileECMDocument Implementation* on page 56
- *FTPDestinationSystem Implementation* on page 59
- *MockPhaseListener Implementation* on page 76

BATCHLOADERSOURCE IMPLEMENTATION

```
package oracle.documaker.ecmconnector.batchloadersource;

import java.text.SimpleDateFormat;

import java.util.Enumeration;
import java.util.List;

import java.util.Properties;
import java.util.UUID;
import java.util.Vector;

import org.apache.log4j.Logger;

import oracle.documaker.ecmconnector.connectorapi.Source;
import
oracle.documaker.ecmconnector.connectorapi.data.ConfigurationData;
import oracle.documaker.ecmconnector.connectorapi.data.DocumentData;
import
oracle.documaker.ecmconnector.connectorapi.data.DataIdentifier;
import oracle.documaker.ecmconnector.connectorapi.data.ECMDocument;
import
oracle.documaker.ecmconnector.connectorapi.data.FileECMDocument;
import
oracle.documaker.ecmconnector.connectorapi.exceptions.SourceExceptio
n;

public class BatchLoaderSource extends Source {
    private static Logger logger =
        Logger.getLogger(BatchLoaderSource.class.getName());
    private static Integer maxBatchSize;
    private SimpleDateFormat dateFormat =
        new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
    private SimpleDateFormat batchDateFormat =
        new SimpleDateFormat("MM/dd/yy");
    private BatchLoaderSystem batchSystem;

    public BatchLoaderSource(ConfigurationData configData,
                             String sourceId) throws SourceException {
        super(configData, sourceId);
        if (maxBatchSize == null)
            maxBatchSize =

Integer.parseInt(configData.getProperty("batchloader.source.max.reco
rds",
                                                                   "1"));
        if (maxBatchSize < 1)
            maxBatchSize = 1;
        logger.debug("Maximum batch size is " + maxBatchSize);

        if (batchSystem == null) {
            batchSystem = BatchLoaderSystem.getInstance();
            batchSystem.configure(configData);
        }
    }

   public void acquireDocumentList(List<DocumentData> documentList)
throws SourceException {
        Vector<Properties> fileRecords =
            batchSystem.acquireBatchFileList(maxBatchSize);
        String batchId = UUID.randomUUID().toString();

        logger.debug("BatchId=" + batchId);
```

```
        for (Properties fileRecord : fileRecords) {
          String primaryFile = fileRecord.getProperty("primaryFile");
            String batchFileName =
            fileRecord.getProperty(BatchLoaderSystem.BATCHFILENAME);

           if (primaryFile == null || primaryFile.isEmpty() == true) {
                batchSystem.writeBadRecordToFile(fileRecord,
                 "\"primaryFile\" property is missing", null, false);
                continue;
            }

            ECMDocument ecmDoc = new FileECMDocument(primaryFile);
            DocumentData metaData =
                new BatchLoaderDocumentData(batchId, ecmDoc,
                                                batchFileName);
            Enumeration keys = fileRecord.keys();

            try {
                while (keys.hasMoreElements()) {
                    String key = (String)keys.nextElement();

                    if (key.equalsIgnoreCase("Action") == true) {
                       logger.debug("Skipping \"Action\" name/value
pair");
                        continue;
                    }

                    if (key.equals(BatchLoaderSystem.BATCHFILENAME)
== true) {
                        logger.debug("Skipping batch file name name/
value pair");
                        continue;
                    }

                    if (key.toLowerCase().indexOf("date") >= 0)
                        metaData.put(new DataIdentifier(key, true),

dateFormat.format(batchDateFormat.parse(fileRecord.getProperty(key))
));
                    else
                        metaData.put(new DataIdentifier(key),
                                 fileRecord.getProperty(key));
                }

                documentList.add(metaData);
            } catch (Exception genErr) {
                batchSystem.writeBadRecordToFile(fileRecord,
                                   "Invalid date format", null,
                                           false);
            }
        }
    }

   public void processResults(List<DocumentData> documentList) throws
SourceException {
        for (DocumentData docData : documentList) {
            if (docData.getResultCode() == DocumentData.IMPORTED)

batchSystem.addDeletableFile((String)docData.getProperty("primaryFil
e"));
            else
                batchSystem.writeImportErrorRecordToFile(docData);
        }
    }
```

```
    public boolean repair() {
        return false;
    }

    public void cleanUp() {
    }
}
```

BatchLoaderSystem Implementation

BatchLoaderSource Implementation section with the following code:

```
package oracle.documaker.ecmconnector.batchloadersource;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.io.RandomAccessFile;

import java.nio.channels.FileLock;

import java.util.Collections;
import java.util.Enumeration;
import java.util.HashSet;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Properties;
import java.util.Set;
import java.util.Vector;

import oracle.documaker.ecmconnector.connectorapi.data.DocumentData;
import
oracle.documaker.ecmconnector.connectorapi.data.ConfigurationData;
import
oracle.documaker.ecmconnector.connectorapi.data.DataIdentifier;
import
oracle.documaker.ecmconnector.connectorapi.exceptions.ExceptionCodes
;
import
oracle.documaker.ecmconnector.connectorapi.exceptions.SourceExceptio
n;

import org.apache.log4j.Logger;

class BatchLoaderSystem {
    private static Logger logger =
        Logger.getLogger(BatchLoaderSystem.class.getName());
    private boolean configured;
    private String identifier;
    private String pathSep;
    private File currentBatchFile;
    private File batchQueueFile;
    private boolean queuedFiles;
    private String errorDirectory;
    private BufferedReader batchFileReader;
    private PrintWriter sourceRecordErrors;
    private PrintWriter importRecordErrors;
    private Set deletableFiles =
        Collections.synchronizedSet(new HashSet<String>());
    private Boolean deleteImportedFiles;
    private Integer deleteFileCount;
    private static BatchLoaderSystem batchSystem;
    private static Hashtable<String, BatchLoaderSystem>
alternateSystems;


    private BatchLoaderSystem() {
        // Hidden to prevent instances
        logger.debug("Executing");
```

```
        configured = false;
    }

    static synchronized BatchLoaderSystem getInstance() {
        logger.debug("Executing");

        if (batchSystem == null) {
            logger.debug("Creating singleton instance");
            batchSystem = new BatchLoaderSystem();
        }

        return batchSystem;
    }

    static synchronized BatchLoaderSystem getInstance(String
identifier) throws BatchLoaderSourceException {
        logger.debug("Executing");

        if (identifier == null)
            return getInstance();

        if (alternateSystems == null)
            alternateSystems = new Hashtable<String,
BatchLoaderSystem>();

        BatchLoaderSystem retVal = alternateSystems.get(identifier);

        if (retVal == null) {
            retVal = new BatchLoaderSystem();
            retVal.identifier = identifier;
            alternateSystems.put(identifier, retVal);
        }

        return retVal;
    }

    synchronized void configure(ConfigurationData configData) throws
SourceException {
        logger.debug("Executing");

        boolean configError = false;

        pathSep = System.getProperty("file.separator");

        // Read the cleanup data
        if (deleteImportedFiles == null)
            deleteImportedFiles =

Boolean.parseBoolean(configData.getPropertyWithModifier("source.impo
rt.delete.imported.files",
                                                         identifier,
                                                          "false"));
       logger.debug("Delete imported files - " + deleteImportedFiles);

        if (deleteFileCount == null)
            deleteFileCount =

Integer.parseInt(configData.getPropertyWithModifier("source.import.d
elete.imported.files.count",
                                                         identifier,
                                                              "0"));
        logger.debug("Number of files per deletion - " +
deleteFileCount);
```

```
        // Get the batch file or batch queue file
        if (currentBatchFile == null && batchQueueFile == null) {
            String path =

configData.getPropertyWithModifier("source.batchloader.batchfile",
                                                        identifier);

            if (path != null && path.isEmpty() == false) {
                currentBatchFile = new File(path);

                if (currentBatchFile.exists() == false) {
                    logger.error("Configured batch file [" + path +
                                 "] does not exist. Please check
configuration.");
                    configError = true;
                } else if (currentBatchFile.isFile() == false) {
                    logger.error("Configured batch file [" + path +
                               "] does not reference a file. Please
check configuration.");
                    configError = true;
                } else {
                 logger.debug("The configured batch file is " + path);
                    queuedFiles = false;
                }
            } else {
                path =
configData.getPropertyWithModifier("source.batchloader.batchqueuefil
e",
                                   identifier);

                if (path != null && path.isEmpty() == false) {
                    batchQueueFile = new File(path);

                    if (batchQueueFile.exists() == false) {
                       logger.error("Configured batch queue file ["
+ path +
                                     "] does not exist. Please check
configuration.");
                        configError = true;
                    } else if (batchQueueFile.isFile() == false) {
                       logger.error("Configured batch queue file ["
+ path +
                                      "] does not reference a file.
Please check configuration.");
                        configError = true;
                    } else {
                      logger.debug("The configured batch queue file
is " +
                                     path);
                        queuedFiles = true;
                    }
                } else {
                    logger.error("No batch file or batch queue file
property provided. Please check configuration.");
                    configError = true;
                }
            }

            if (errorDirectory == null) {
                errorDirectory =

configData.getPropertyWithModifier("source.batchloader.errordirector
y",
                                                                 identifier);
```

```
                if (errorDirectory == null ||
                    errorDirectory.isEmpty() == true) {
                    logger.error("An error directory needs to be
defined. Please check configuration.");
                    configError = true;
                } else {
                    File errorDir = new File(errorDirectory);

                    if (errorDir.exists() == false)
                        errorDir.mkdir();

                    if (errorDir.isDirectory() == false) {
                        logger.error("The error directory path [" +
                                     errorDirectory +
                                "] does not point to a directory.");
                        configError = true;
                } else if (errorDirectory.endsWith(pathSep) == false)
                        errorDirectory += pathSep;
                }
            }

            // Validate operational mode
            // Todo - figure this out
        }

        if (configError == true)
            throw new SourceException("Failed to configure the Batch
Loader Source. Please check log for details.",
                                ExceptionCodes.CNT0501500002, null);

        configured = true;
        openBatchFile();
    }

    synchronized Vector<Properties> acquireBatchFileList(int count)
throws SourceException {
        if (configured == false)
           throw new SourceException("The BatchLoaderSystem instance
has not been configured.",
                                ExceptionCodes.CNT0501500002, null);

        Vector<Properties> batchFileList = new Vector<Properties>();

        if (batchFileReader == null) {
            openBatchFile();

            if (batchFileReader == null)
                return batchFileList;
        }

        while (batchFileList.size() < count) {
            Properties batchData = readBatchRecord();

            // Check for end of file
            if (batchData == null || batchData.size() == 0) {
                openBatchFile();

                if (batchFileReader == null)
                    return batchFileList;
                else
                    continue;
            }
```

```
            String actionType = batchData.getProperty(ACTIONLABEL);

            if (actionType == null)
              writeBadRecordToFile(batchData, "No action specified",
null,
                                   false);
         else if (actionType.equalsIgnoreCase(ACTIONINSERT) == false)
                writeBadRecordToFile(batchData,
                                 "Unsupported action specified - " +
                                   actionType, null, false);
            else
                batchFileList.add(batchData);
        }

        return batchFileList;
    }

    synchronized void addDeletableFile(String filePath) {
        if (deleteImportedFiles == true)
            deletableFiles.add(filePath);
    }

    synchronized Vector<String> acquireDeletableFiles() {
        Vector<String> returnVal = new Vector<String>();

        if (deleteImportedFiles == true) {
            Iterator<String> fileIter = deletableFiles.iterator();

            while (returnVal.size() < deleteFileCount &&
fileIter.hasNext())
                returnVal.add(fileIter.next());

            for (String fileP : returnVal) {
                logger.debug("returning - " + fileP);

                deletableFiles.remove(fileP);
            }
        }

        return returnVal;
    }

    private void openBatchFile() throws SourceException {
        // Open configured batch file
        if (queuedFiles == false) {
            if (batchFileReader != null)
            throw new SourceException("Trying to reprocess configured
batch file. This is probably due to running in server mode without a
batch queue. Please check configuration.",
                                ExceptionCodes.CNT0501500004, null);
            else {
                try {
                    batchFileReader =
                            new BufferedReader(new
FileReader(currentBatchFile));
                    return;
                } catch (Exception genErr) {
                throw new SourceException("Failed to open configured
batch file [" +

currentBatchFile.getAbsolutePath() +
                                        "] - " + genErr.getMessage(),
                                              genErr,
                                       ExceptionCodes.CNT0501500005,
```

```
                                              new Object[] {
currentBatchFile.getAbsolutePath(),
                                                genErr.getMessage()
});
                }
            }
        } else {
            try {
                String nextFilePath = readNextBatchFilePath();

               if (nextFilePath == null || nextFilePath.isEmpty() ==
true) {
                    logger.info("No batch file found in queue.");
                    batchFileReader = null;
                } else {
                    currentBatchFile = new File(nextFilePath);
                    batchFileReader =
                            new BufferedReader(new
FileReader(currentBatchFile));
                }
            } catch (SourceException srcErr) {
                throw srcErr;
            } catch (Exception genErr) {
               throw new SourceException("Failed to open batch queue
file or queue'd batch file - " +
                                        genErr.getMessage(), genErr,
                                       ExceptionCodes.CNT0501500006,
                                  new Object[] { genErr.getMessage()
});
            }
        }
    }

    private String readNextBatchFilePath() throws SourceException {
        if (batchQueueFile == null)
         throw new SourceException("No batch queue file configured.",
                               ExceptionCodes.CNT0501500007, null);

        RandomAccessFile randFile = null;
        FileLock fileLock = null;

        try {
            randFile = new RandomAccessFile(batchQueueFile, "rwd");

            fileLock = randFile.getChannel().lock();

            BufferedReader fileRdr =
               new BufferedReader(new FileReader(randFile.getFD()));
            Vector<String> lines = new Vector<String>();
            String line = fileRdr.readLine();
            String nextFilePath = line;

            while (line != null) {
                lines.add(line);
                line = fileRdr.readLine();
            }

            randFile.seek(0);

            PrintWriter fileWtr =
                new PrintWriter(new FileWriter(randFile.getFD()));

            for (int lcv = 1; lcv < lines.size(); ++lcv) {
                fileWtr.println(lines.get(lcv));
```

```
                fileWtr.flush();
            }

            randFile.setLength(randFile.getFilePointer());

            return nextFilePath;
        } catch (Exception genErr) {
            throw new SourceException("Failed to get path for next
batch file - " +
                                          genErr.getMessage(), genErr,
                                          ExceptionCodes.CNT0501500008,
                                   new Object[] { genErr.getMessage() });
        } finally {
            if (fileLock != null)
                try {
                    fileLock.release();
                } catch (Exception genErr) { /* ignore */
                }
            if (randFile != null)
                try {
                    randFile.close();
                } catch (Exception genErr) { /* ignore */
                }
        }
    }

    private Properties readBatchRecord() throws
BatchLoaderSourceException {
        logger.debug("Executing");

        Properties fileData = new Properties();

        fileData.setProperty(BATCHFILENAME,
currentBatchFile.getName());
        try {
            String line = batchFileReader.readLine();

            while (line != null && line.equals(EODMARKER) == false) {
                if (line.startsWith("#") == false &&
                    line.trim().isEmpty() == false) {
                    int index = line.indexOf("=");

                    if (index < 1 || index == (line.length() - 1)) {
                        // Write current record to source error file
                        writeBadRecordToFile(fileData,
                                            "Line does not contain a
name value pair",
                                             line, true);

                        // Start new record
                        fileData.clear();
                    } else {
                      String name = line.substring(0, index).trim();
                     String value = line.substring(index + 1).trim();

                        if (name.isEmpty() || value.isEmpty()) {
                         // Write current record to source error file
                            writeBadRecordToFile(fileData,
                                              "Line does not contain
a name value pair",
                                                  line, true);

                            // Start new record
                            fileData.clear();
```

```
                        } else
                            fileData.setProperty(name, value);
                    }
                }

                line = batchFileReader.readLine();
            }
        } catch (IOException ioErr) {
            writeBadRecordToFile(fileData, ioErr.getMessage(), null,
false);
            throw new BatchLoaderSourceException("Batch file read
exception - " + ioErr.getMessage(),
                                                  ioErr,
ExceptionCodes.CNT0501500009, new Object[] { ioErr.getMessage() });
        }

        return ((fileData.size() <= 1) ? null : fileData);
    }

    synchronized void writeBadRecordToFile(Properties record,
                                String errorMessage, String line,
                                       boolean getRest) throws
BatchLoaderSourceException {
        try {
            getErrorWriter(SOURCEWRITER,
record.getProperty(BATCHFILENAME));
            sourceRecordErrors.println("### RECORD ERROR - " +
errorMessage);

            Enumeration keys = record.keys();

            while (keys.hasMoreElements()) {
                String key = (String)keys.nextElement();

                if (key.equalsIgnoreCase(BATCHFILENAME) == false)
                    sourceRecordErrors.println(key + "=" +
                                           record.getProperty(key));
            }

            if (line != null)
               sourceRecordErrors.println("### INVALID DATA IN BATCH
FILE - " +
                                           line);

            if (getRest == true) {
                String tempBuf = batchFileReader.readLine();

                while (tempBuf != null) {
                    sourceRecordErrors.println(tempBuf);

                    if (tempBuf.equals(EODMARKER) == true)
                        break;

                 if ((tempBuf = batchFileReader.readLine()) == null) {
                        sourceRecordErrors.println("### Missing end
of data marker ... adding");
                        sourceRecordErrors.println(EODMARKER);
                    }
                }
            }
        } catch (IOException ioErr) {
            throw new BatchLoaderSourceException("Source Error File
write exception - " + ioErr.getMessage(),
```

```
                                                ioErr,
ExceptionCodes.CNT0501500010, new Object[] { ioErr.getMessage() });
        } finally {
            if (sourceRecordErrors != null)
                sourceRecordErrors.close();
        }
    }

    synchronized void writeImportErrorRecordToFile(DocumentData
metaDatum) throws BatchLoaderSourceException {
        try {
            getErrorWriter(IMPORTWRITER,

((BatchLoaderDocumentData)metaDatum).getBatchFileName());
         importRecordErrors.println("### RECORD ERROR - Record Faied
to Import");

            Enumeration keys = metaDatum.keys();

            // Write the failure info
            importRecordErrors.println("### [" +
metaDatum.getResultCode() +
                                        "] - " +
                                   metaDatum.getResultDescription());
          // Add the action back to the record (will always be insert)
            importRecordErrors.println("Action=insert");

            while (keys.hasMoreElements()) {
             DataIdentifier key = (DataIdentifier)keys.nextElement();

                importRecordErrors.println(key.getName() + "=" +
                                           metaDatum.get(key));
            }

            // Add end of data marker
            importRecordErrors.println("<<EOD>>");
            importRecordErrors.println();
        } finally {
            if (importRecordErrors != null)
                importRecordErrors.close();
        }
    }

    synchronized void getErrorWriter(int writerType,
                                     String batchFileName) throws
BatchLoaderSourceException {
        StringBuffer errorFileName = new
StringBuffer().append(errorDirectory);

        if (writerType == SOURCEWRITER)
            errorFileName.append(batchFileName + ".SOURCEERRORS");
        else
            errorFileName.append(batchFileName + ".IMPORTERRORS");

        File errFile = new File(errorFileName.toString());

        if (errFile.length() > (100 * 1024)) {
            int nameCntr = 1;
            File newFile =
               new File(errFile.getAbsolutePath() + "." + nameCntr);

            while (newFile.exists() == true) {
                nameCntr += 1;
```

```
                newFile = new File(errFile.getAbsolutePath() + "." +
nameCntr);
            }

            errFile.renameTo(newFile);
        }

        try {
            if (writerType == SOURCEWRITER)
                sourceRecordErrors =
                        new PrintWriter(new
java.io.FileWriter(errorFileName.toString(),
                                                    true), true);
            else
                importRecordErrors =
                        new PrintWriter(new
java.io.FileWriter(errorFileName.toString(),
                                                    true), true);
        } catch (Exception genErr) {
            if (writerType == SOURCEWRITER) {
                sourceRecordErrors = null;
                throw new BatchLoaderSourceException("Could not open
source error file [" +
                                            errorFileName.toString()
+ "] - " + genErr.getMessage(),
                                                  genErr,
ExceptionCodes.CNT0501500011, new Object[] {
errorFileName.toString(), genErr.getMessage() });
            } else {
                importRecordErrors = null;
                throw new BatchLoaderSourceException("Could not open
import error file [" +
                                            errorFileName.toString()
+ "] - " + genErr.toString(),
                                                  genErr,
ExceptionCodes.CNT0501500012, new Object[] {
errorFileName.toString(), genErr.getMessage() });
            }
        }
    }


    private static final String EODMARKER = "<<EOD>>";
    private static final String ACTIONLABEL = "Action";
    private static final String ACTIONINSERT = "insert";
    public static final String BATCHFILENAME = "__BATCH_FILE_NAME__";
    private static final int SOURCEWRITER = 1;
    private static final int IMPORTWRITER = 2;
}
```

BatchLoaderDocumentData Implementation

In the BatchLoaderSource implementation, you need to maintain additional information about each document that will not be part of the import data (the batch file name that the record came from). To do this, you simply sub-class the DocumentData class and add an accessor for the information.

```
package oracle.documaker.ecmconnector.batchloadersource;

import oracle.documaker.ecmconnector.connectorapi.data.DocumentData;
import oracle.documaker.ecmconnector.connectorapi.data.ECMDocument;

public class BatchLoaderDocumentData extends DocumentData {
    private String batchFileName;

    public BatchLoaderDocumentData(String string, ECMDocument
ecmDocument, String batchFileName) {
        super(string, ecmDocument);
        this.batchFileName = batchFileName;
    }

    public String getBatchFileName() {
        return batchFileName;
    }
}

The BatchLoaderProcess Implementation
package oracle.documaker.ecmconnector.batchloadersource;

import java.io.File;

import java.io.PrintWriter;

import java.util.Vector;

import
oracle.documaker.ecmconnector.connectorapi.data.ConfigurationData;
import
oracle.documaker.ecmconnector.connectorapi.process.PeriodicProcess;
import
oracle.documaker.ecmconnector.connectorapi.exceptions.SourceExceptio
n;

public class BatchLoaderProcess extends PeriodicProcess {
    private PrintWriter cleanUpErrors;

    public BatchLoaderProcess(ConfigurationData configurationData,
                              String identifier) throws SourceException {
        super(configurationData, identifier);
        configure(configurationData);
    }

    public BatchLoaderProcess(ConfigurationData configurationData)
throws SourceException {
        super(configurationData);
        configure(configurationData);
    }

   private void configure(ConfigurationData configurationData) throws
SourceException {
       BatchLoaderSystem.getInstance().configure(configurationData);

        String pathSep = System.getProperty("file.separator");
```

```
        String persistencePath =
           configurationData.getProperty("source.persistence.path",
".");

        try {
            cleanUpErrors =
             new PrintWriter(new java.io.FileWriter(persistencePath +
                                (persistencePath.endsWith(pathSep) ?
                                   "" : pathSep) + "CLEANUP_ERRORS",
                                    true), true);
        } catch (Exception genErr) {
            throw new SourceException("Failed to create writer to
cleanup errors file.",
                                       genErr);
        }
    }

    public void process() {
        Vector<String> filePaths =
            BatchLoaderSystem.getInstance().acquireDeletableFiles();

        for (String filePath : filePaths) {
            File deletableFile = new File(filePath);

            if (deletableFile.exists() == true)
                deletableFile.delete();
            else
               cleanUpErrors.println("Deletable file [" + filePath +
                                       "] does not exist.");
        }
    }
}

< The FileECMDocument implementation is good to go >

< Drop the FTPDestinationAdministration implementation >

< Replace the FTPDestination Implementation code with the following >
package oracle.documaker.ecmconnector.ftpdestination;

import oracle.documaker.ecmconnector.connectorapi.Destination;
import
oracle.documaker.ecmconnector.connectorapi.data.ConfigurationData;
import oracle.documaker.ecmconnector.connectorapi.data.DocumentData;
import
oracle.documaker.ecmconnector.connectorapi.exceptions.DestinationExc
eption;

import org.apache.log4j.Logger;

public class FTPDestination extends Destination {
    private static Logger logger =
        Logger.getLogger(FTPDestination.class.getName());
    private FTPDestinationSystem ftpSystem;

    public FTPDestination(ConfigurationData configurationData,
                  String destinationId) throws DestinationException
{
        super(configurationData, destinationId);

        ftpSystem = FTPDestinationSystem.getInstance(destinationId);
        ftpSystem.configure(configurationData);
    }
```

```
    public void importDocument(DocumentData documentData) throws
DestinationException {
        ftpSystem.uploadDocumentContents(documentData);
    }

    public boolean repair() {
        return ftpSystem.establishConnection();
    }

    public void cleanUp() {
    }
}}
```

FILEECMDOCUMENT IMPLEMENTATION

```
public class FileECMDocument implements ECMDocument {
    private String filePath;
    private String fileName;
    private FileInputStream inputStream;
    private long contentLength;
    private Logger logger =
Logger.getLogger(FileECMDocument.class.getName());

    /**
     * Constructs a new FileECMDocument instance using the given file
path to identify the location of the document's contents.
     *
     * @param filePath The path to the file containing the document's
contents
     */
    public FileECMDocument(String filePath) {
        this.filePath = filePath;
        inputStream = null;

        File sourceFile = new File(filePath);

        contentLength = sourceFile.length();
        fileName = sourceFile.getName();
        logger.debug("fileName = " + fileName);
    }

    public String acquireFilePath() throws ECMDocumentException {
        if (filePath == null)
           throw new ECMDocumentException("No filepath available.");
        return filePath;
    }

   public ECMInputStream acquireStream() throws ECMDocumentException
{
        try {
            if (inputStream == null)
                inputStream = new FileInputStream(filePath);
            else
             throw new ECMDocumentException("Input stream associated
with previous instance.");
        } catch (FileNotFoundException fnfErr) {
            fnfErr.printStackTrace();
            throw new ECMDocumentException("Failed to create input
stream.",
                                              fnfErr);
        }

        return new ECMInputStream(fileName, inputStream);
    }

    public long getContentLength() {
        return contentLength;
    }

    public void release() throws ECMDocumentException {
        try {
            if (inputStream != null) {
                inputStream.close();
                inputStream = null;
            }
        } catch (IOException ioErr) {
```

```
            throw new ECMDocumentException("Failed to close input
stream.",
                                           ioErr);
        }
    }
}
```

FTPDestination Implementation

```
public class FTPDestination implements Destination {
    private static Logger logger =
Logger.getLogger(FTPDestination.class.getName());
    private FTPDestinationSystem ftpSystem;

    public FTPDestination() {
        logger.debug("Executing");
    }

    public void configure(Properties properties) throws
DestinationException {
        logger.debug("Executing");

        if (ftpSystem == null)
            ftpSystem = FTPDestinationSystem.getInstance();
    }

    public void beginTransaction(TransactionTypes transactionTypes)
throws DestinationException {
        logger.debug("Executing");
    }

    public void executeTransaction(TransactionTypes transactionType,
MetaData metaData) throws DestinationException {
        logger.debug("Executing");

        switch (transactionType) {
        case IMPORT:
            ftpSystem.uploadDocumentContents(metaData);
            metaData.setResultCode(MetaData.IMPORTED);
            break;

        default:
            throw new DestinationException("Transaction type not
supported");
        }
    }

    public void endTransaction(TransactionTypes transactionTypes)
throws DestinationException {
        logger.debug("Executing");
    }
}
```

FTPDestinationSystem Implementation

```
class FTPDestinationSystem {
    private static Logger logger =
      Logger.getLogger(FTPDestinationSystem.class.getName());
    private boolean configured;
    private String identifier;
    private String ftpServer;
    private String userName;
    private String passWord;
    private int port=0;
    private String baseDirectory;
    private String sideBaseDirectory;
    private String defaultSubdirectoryPattern;
    private String defaultSideSubdirectoryPattern;
    private String defaultTemplatePath;
    private String defaultTemplateString;
    private String defaultFileNamePattern;
    private String defaultSideFileNamePattern;
    private FTPClient ftpClient;
    private Configuration fmConfig;
    private static FTPDestinationSystem ftpDestinationSystem;
    private static Hashtable<String, FTPDestinationSystem>
alternateSystems;

    private FTPDestinationSystem() {
        logger.debug("Executing");
        configured = false;
    }

    static FTPDestinationSystem getInstance() {
        logger.debug("Executing");

        if (ftpDestinationSystem == null) {
            logger.debug("Creating singleton instance");
           ftpDestinationSystem = new FTPDestinationSystem();
        }

        return ftpDestinationSystem;
    }

    static synchronized FTPDestinationSystem
getInstance(String identifier) throws DestinationException {
        logger.debug("Executing");
```

```
        if (identifier == null)
            return getInstance();

        if (alternateSystems == null)
            alternateSystems = new Hashtable<String,
FTPDestinationSystem>();

        FTPDestinationSystem retVal =
alternateSystems.get(identifier);

        if (retVal == null) {
            retVal = new FTPDestinationSystem();
            retVal.identifier = identifier;
            alternateSystems.put(identifier, retVal);
        }

        return retVal;
    }

    synchronized void configure(ConfigurationData configData)
throws DestinationException {
        logger.debug("Executing");

        if (ftpServer == null)
            ftpServer =
               configData.getPropertyWithModifier(FTP_SERVER,
                                                  identifier);
        logger.debug("FTP server address - " + ftpServer);


        if (userName == null)
            userName =

configData.getPropertyWithModifier(FTP_USERNAME,
                                                  identifier);

        logger.debug("FTP user name - " + userName);

        if (passWord == null)
            passWord =

configData.getPropertyWithModifier(FTP_PASSWORD,
                                                  identifier);
```

```
        logger.debug("FTP password acquired");

        if (port ==0)
            port =

Integer.parseInt(configData.getPropertyWithModifier(FTP_PORT,
                                                 identifier));
       logger.debug("FTP port number");
        if (baseDirectory == null)
            baseDirectory =

configData.getPropertyWithModifier(BASE_DIRECTORY,
                                                  identifier);
        //startrs added by Kotes to implement FTP indexng and
templating
        baseDirectory =
                baseDirectory +
(baseDirectory.endsWith(File.separator) ? "" :
File.separator);
        logger.debug("Base directory - " + baseDirectory);

        sideBaseDirectory =
configData.getPropertyWithModifier(SIDE_BASE_DIRECTORY,identi
fier);

        if (sideBaseDirectory == null)
            sideBaseDirectory = baseDirectory;
        else {
            sideBaseDirectory =
                    sideBaseDirectory +
(sideBaseDirectory.endsWith(File.separator) ? "" :
File.separator);
        }
        defaultTemplatePath =
           configData.getPropertyWithModifier(TEMPLATE_FILE,
identifier);
        if (defaultTemplatePath != null) {
          File templateFile = new File(defaultTemplatePath);

            if (templateFile.exists() == false)
               throw new DestinationException("The specified
template file [" +
                                        defaultTemplatePath +
                                         "] does not exist.",
```

```
ExceptionCodes.CNT0500800008,
                                               new Object[] {
defaultTemplatePath });
             if (logger.isDebugEnabled() == true)
                logger.debug("Template path is " +
defaultTemplatePath);

        } else {
            defaultTemplateString =

configData.getPropertyWithModifier(TEMPLATE_CONTENTS,
                                                  identifier);

            if (logger.isDebugEnabled() == true)
                logger.debug("Template pattern - " +
defaultTemplateString);
        }

        defaultFileNamePattern =
             configData.getPropertyWithModifier(FILE_PATTERN,
                                                  identifier);
        if (logger.isDebugEnabled() == true)
            logger.debug("Filename pattern - " +
defaultFileNamePattern);


        defaultSideFileNamePattern =

configData.getPropertyWithModifier(SIDE_FILE_PATTERN,
                                                  identifier);

        if (logger.isDebugEnabled() == false)
            logger.debug("Side filename pattern - " +
                         defaultSideFileNamePattern);

        defaultSubdirectoryPattern =

configData.getPropertyWithModifier(SUB_DIRECTORY_PATTERN,
                                                  identifier);
        if (logger.isDebugEnabled() == true)
            logger.debug("Subdirectory pattern - " +
defaultSubdirectoryPattern);
```

```
        defaultSideSubdirectoryPattern =

configData.getPropertyWithModifier(SIDE_SUB_DIRECTORY_PATTERN
,
                                                      identifier);
        if (logger.isDebugEnabled() == true)
            logger.debug("Side subdirectory pattern - " +
                         defaultSideSubdirectoryPattern);

        configured = true;

        establishConnection();


        try {
            fmConfig = new Configuration();

            fmConfig.setDirectoryForTemplateLoading(new
File("."));
            fmConfig.setObjectWrapper(new
DefaultObjectWrapper());
         } catch (IOException ioErr) {
         throw new DestinationException("Failed to initialize
template framework.",

ExceptionCodes.CNT0500800009,
                                          null);
        }
    }
    /**
     * Establish conection to FTP server with the given details
       */
    public boolean establishConnection() {
        if (configured == false)
            return false;

        try {
            if (checkServer() == true)
                ftpClient.disconnect();

            ftpClient = new FTPClient();
            ftpClient.connect(ftpServer,port);
            if
(FTPReply.isPositiveCompletion(ftpClient.getReplyCode()) ==
false) {
```

```
                ftpClient.disconnect();
                ftpClient = null;
            } else if (ftpClient.login(userName, passWord) ==
false) {
                ftpClient.logout();
                ftpClient = null;
            } else {
                int retVal = ftpClient.cwd(baseDirectory);

               if (FTPReply.isPositiveCompletion(retVal) ==
false) {
                  ftpClient.mkd(baseDirectory);
                  ftpClient.cwd(baseDirectory);
               }
            }

            return true;
        } catch (IOException ioErr) {
            try {
                ftpClient.disconnect();
                ftpClient = null;
            } catch (Exception genErr) {
                // Ignore
            }
            return false;
        }
    }

    synchronized boolean checkServer() {
        logger.debug("Executing");
       return (ftpClient != null && ftpClient.isConnected());
    }

    synchronized void close() {
        try {
            if (checkServer() == true)
                ftpClient.disconnect();
        } catch (IOException ioErr) {
            // Do nothing
        }
    }
```

```
    /**
     * Upload documents in FTP server
     * Main Files
     * Side Files
     * With the given Template formats
       */
    synchronized void uploadDocumentContents(DocumentData
metaData) throws DestinationException {
        logger.debug("Executing");
        String fileName="";
        String fileDirPath="";
                if (configured == false){
                        throw new DestinationException("FTP
destination system has not been configured.",

ExceptionCodes.CNT0500900001, null);
         }


         if (ftpClient == null || ftpClient.isConnected() ==
false){
            if(!establishConnection())
           throw new DestinationException("No FTP destination
system connection available.",

ExceptionCodes.CNT0500900002, null);
                    }
         fileDirPath =
            baseDirectory +
(baseDirectory.endsWith(File.separator) ? "" : File.separator)
+
            determineActualPath(SUB_DIRECTORY_PATTERN,
defaultSubdirectoryPattern, metaData);
        ECMDocument contents = metaData.getECMDocument();
              try {
                  ECMInputStream ecmInput =
contents.acquireStream();


            fileName = createMainFilePath(metaData);
          if(fileName==null){
                fileName=ecmInput.getFileName();
            }
            InputStream in =ecmInput.getInputStream();
          //  InputStream in =new
FileInputStream(contents.acquireFilePath());   //To work in
Local Environment
```

```
            // Calling FTP Operation to store MAIN DOC
            if(logger.isDebugEnabled() == true)
                          logger.debug("THe SUB DIRECTORY PATH
IS @@@@@@@@@@1"+fileDirPath);


           ftpOperations(fileDirPath, fileName,in);
           metaData.put(new DataIdentifier(GENERATED_NAME),
                               fileDirPath);
            String sideFileName = createSideFilePath(metaData);

            if (sideFileName != null && sideFileName.length() >
0) {
                    String sideFileDirPath =
                     sideBaseDirectory +
(sideBaseDirectory.endsWith(File.separator) ? "" :
File.separator) +

determineActualPath(SIDE_SUB_DIRECTORY_PATTERN,
defaultSideSubdirectoryPattern, metaData);
              //Call for Side file operations

              if(logger.isDebugEnabled() == true)
                          logger.debug("THe SIDE SUB DIRECTORY
PATH IS @@@@@@@@"+sideFileDirPath);
             generateXMLDocument(sideFileDirPath,sideFileName,
metaData);
              }
               else if (logger.isDebugEnabled() == true)
               logger.debug("No side-car file to be written.");

          } catch (ECMDocumentException docErr) {
              throw new DestinationException("Failed to acquire
document contents - " +
                                          docErr.getMessage(),

ExceptionCodes.CNT0500900003,
                                            new Object[] {
docErr.getMessage() });
          }
          catch (IOException ioErr) {
              try {

                 ftpClient.disconnect();
             } catch (IOException innerIoErr) { /* ignore */
```

```
          }
             ftpClient = null;
          throw new DestinationException("Error in destination
system communication - " +
                                          ioErr.getMessage(),

ExceptionCodes.CNT0500900004,
                                          new Object[] {
ioErr.getMessage() });
        }

finally {
            try {
                contents.release();
            } catch (Exception genErr) { /* ignore */

            }
        }
    }

    /**
     * Upload documents in FTP server
     * in Given directory
     * given file name
     * and input stream
       */
    private void ftpOperations(String fileDirPath, String
filename, InputStream in) throws DestinationException,
                                                 IOException {

                boolean dirExists = true;
        String fileSep = System.getProperty("file.separator");
         establishConnection();
        if (ftpClient == null || ftpClient.isConnected() ==
false){
           if(!establishConnection())
          throw new DestinationException("No FTP destination
system connection available.",
                                 ExceptionCodes.CNT0500900002,
null);
           }

             if(fileSep!=null && !fileSep.equalsIgnoreCase("/
"))
                    fileSep="\\";
```

```
                 String fileDir1=fileDirPath.replace(fileSep,
"/");
                 ftpClient.cwd("..");
          String[] directories = fileDir1.split("/");
          for (String dir : directories ) {
              if (!dir.isEmpty() ) {
              if (dirExists) {
                dirExists =
ftpClient.changeWorkingDirectory(dir);
              }
              if (!dirExists) {
                if (!ftpClient.makeDirectory(dir)) {
                    throw new DestinationException("Unable to
created remote directory '" + dir,
ExceptionCodes.CNT0500900004,  new Object[]
{ftpClient.getReplyString()});
                }
                if (!ftpClient.changeWorkingDirectory(dir)) {
                    throw new DestinationException("Unable to
change into newly created remote directory '" + dir,
ExceptionCodes.CNT0500900004,  new Object[]
{ftpClient.getReplyString()});
                }
              }
            }
          }
                  ftpClient.setFileType(FTP.BINARY_FILE_TYPE);
                  ftpClient.enterLocalPassiveMode();
                  ftpClient.storeFile(filename, in);
                  for(int i=0; i<=directories.length; i++){
                      ftpClient.changeToParentDirectory();
                      }

//         ftpClient.cwd("..");
//            int retVal = ftpClient.cwd(fileDirPath);
//        if (logger.isDebugEnabled() == true)
//            logger.debug("The Ret values is " + retVal);
//      if (FTPReply.isPositiveCompletion(retVal) == false) {
//             if (logger.isDebugEnabled() == true)
//                logger.debug("The Ret values is @@@@@@@@" +
retVal);
//             if (logger.isDebugEnabled() == true)
//                logger.debug("THe Current working Directory
is" + ftpClient.printWorkingDirectory());
//            ftpClient.mkd(fileDirPath);
```

```
//             ftpClient.cwd(fileDirPath);
//          }
//          // Change mode to binary
//
//
//          ftpClient.setFileType(FTP.BINARY_FILE_TYPE);
//          ftpClient.enterLocalPassiveMode();
//          ftpClient.storeFile(filename, in);
//         ftpClient.cwd("..");

    }

    private String createMainFilePath(DocumentData
documentData) throws DestinationException {

        String mainFileName = generateFileName(documentData,
FILE_PATTERN,
                                defaultFileNamePattern);

        return mainFileName;
    }

    private String createSideFilePath(DocumentData
documentData) throws DestinationException {
       String  sidefile=generateFileName(documentData,
SIDE_FILE_PATTERN,
                                defaultSideFileNamePattern);
       return sidefile;
    }

    private String generateFileName(DocumentData documentData,
                                    String patternKey,
                                    String defaultPattern)
throws DestinationException {
        HashMap dataMap = generateDataMap(documentData);
        String fileNamePattern =
(String)documentData.getProperty(patternKey);

        if (fileNamePattern == null)
            fileNamePattern = defaultPattern;

       if (fileNamePattern == null || fileNamePattern.length()
== 0) {
            if (logger.isDebugEnabled() == true)
```

```
                logger.debug("No filename pattern specified.
No filename generated.");
            return null;
        }

        try {
            Template temp =
                new Template(fileNamePattern, new
StringReader(fileNamePattern),
                             fmConfig);
            StringWriter strOut = new StringWriter();

            temp.process(dataMap, strOut);
            return strOut.toString();
        } catch (Exception genErr) {
           throw new DestinationException("Failed to generate
file name due to template error - " + genErr.getMessage(),

ExceptionCodes.CNT0500800010,
                                           new Object[] {
genErr.getMessage() });
        }
    }

   private HashMap generateDataMap(DocumentData documentData)
throws DestinationException {
        HashMap dataMap = new HashMap();

    String s=documentData.getBatchId();
       if (s!=null && s.equalsIgnoreCase(""))
           s="ftpodee";

        dataMap.put("BATCHID", s);
        dataMap.put("substr", new SubstringMethod());

        for (DataIdentifier key : documentData.keySet()) {
            addMapItem(key.getName(),
documentData.get(key).toString(),
                          dataMap);
        }

        try {
          ECMDocument ecmDoc = documentData.getECMDocument();
           String filePath = ecmDoc.acquireFilePath();
```

```
            ecmDoc.release();

            int extIndex = filePath.lastIndexOf(".");
           int fnIndex = filePath.lastIndexOf(File.separator);

            if (extIndex > 0 && extIndex != (fnIndex + 1))
                dataMap.put("INPUT_FILE_EXT",
filePath.substring(extIndex + 1));
            else
                dataMap.put("INPUT_FILE_EXT", "");

            if (fnIndex >= 0) {
                if (extIndex > (fnIndex + 1))
                    dataMap.put("INPUT_FILE_NAME",
filePath.substring(fnIndex + 1, extIndex));
                else
                    dataMap.put("INPUT_FILE_NAME",
filePath.substring(fnIndex + 1));
            }
            else {
                if (extIndex > 0)
                    dataMap.put("INPUT_FILE_NAME",
filePath.substring(0, extIndex));
                else
                    dataMap.put("INPUT_FILE_NAME", filePath);
            }
        } catch (ECMDocumentException e) {
         throw new DestinationException("Error in destination
system communication - " +
                                            e.getMessage(),

ExceptionCodes.CNT0500900004,
                                            new Object[] {
e.getMessage() });
        }

        return dataMap;
    }

    private void addMapItem(String itemName, String itemValue,
                            HashMap dataMap) {
        HashMap childMap = dataMap;
        String[] elements = itemName.split("\\.");
```

```
        int length = elements.length;

        for (int lcv = 0; lcv < (length - 1); ++lcv) {
            HashMap newChild =
(HashMap)childMap.get(elements[lcv]);

            if (newChild == null) {
                newChild = new HashMap();
                childMap.put(elements[lcv], newChild);
            }
            childMap = newChild;
        }

        childMap.put((length > 0 ? elements[length - 1] :
itemName),
                     itemValue);
    }

    public String determineActualPath(String pathPatternId,
String defaultPattern, DocumentData documentData) throws
DestinationException {
        String pathPattern =
(String)documentData.getProperty(pathPatternId);

        if (pathPattern == null)
            pathPattern = defaultPattern;

        if (pathPattern == null)
            pathPattern = "${BATCHID}";

        HashMap dataMap = generateDataMap(documentData);
        Configuration fmConfig = new Configuration();

        fmConfig.setObjectWrapper(new DefaultObjectWrapper());
        try {
            Template temp =
                new Template(pathPattern, new
StringReader(pathPattern),
                             fmConfig);
            StringWriter strOut = new StringWriter();

            temp.process(dataMap, strOut);
            return strOut.toString();
        } catch (Exception genErr) {
```

```
            throw new DestinationException("Generic Exception
- Failed to generate templated string: " +
                                     pathPattern + " :: " +
                                     genErr.getMessage(),
                                ExceptionCodes.CNT0500000002,
                                   new Object[] { "Failed to
generate templated string: " +
                                         pathPattern + " :: " +
                                          genErr.getMessage()
});
        }
    }

   private void generateXMLDocument(String xmlDirnName, String
xmlFileName,
                                  DocumentData documentData)
throws DestinationException        {
        StringWriter strOut = null;
        try {

            HashMap dataMap = generateDataMap(documentData);

            String templatePath =
             (String)documentData.getProperty(TEMPLATE_FILE);
            String templateString = null;

            if (templatePath == null)
                templatePath = defaultTemplatePath;

            if (templatePath == null) {
                templateString =

(String)documentData.getProperty(TEMPLATE_CONTENTS);

                if (templateString == null)
                    templateString = defaultTemplateString;

                if (logger.isDebugEnabled() == true)
                  logger.debug("Template: " + templateString);
            } else if (logger.isDebugEnabled() == true)
                logger.debug("Template file path is " +
templatePath);

           if ((templatePath == null || templatePath.length()
== 0) &&
```

```
            (templateString == null || templateString.length()
== 0)) {
                throw new DestinationException("There is no
template specified for the side-car file, but a file name has
been speccified.",

ExceptionCodes.CNT0500800011,
                                                null);
            }

            Template temp = null;

            if (templatePath == null) {
                if (logger.isDebugEnabled() == true)
                    logger.debug("Using template string");
                temp =new Template(templateString, new
StringReader(templateString), fmConfig);
            } else {
                String fileSep =
System.getProperty("file.separator");
               int index = templatePath.lastIndexOf(fileSep);
              String fileName = templatePath.substring(index
+ 1);
              String path = templatePath.substring(0, index);

                fmConfig.setDirectoryForTemplateLoading(new
File(path));
                temp = fmConfig.getTemplate(fileName);
            }
             strOut = new StringWriter();
            temp.process(dataMap, strOut);
            InputStream in = new
ByteArrayInputStream(strOut.toString().getBytes());
            //FTP operations for Side file creation and store
            ftpOperations(xmlDirnName,xmlFileName,in);
            documentData.put(new
DataIdentifier(GENERATED_SIDE_NAME),
                              xmlFileName);
        } catch (IOException ioe)  {

          throw new DestinationException("There is no template
specified for the side-car file, but a file name has been
speccified.",

ExceptionCodes.CNT0500800011,
                                          null);
```

```
        } catch (TemplateException e) {
          throw new DestinationException("There is no template
specified for the side-car file, but a file name has been
speccified.",

ExceptionCodes.CNT0500800011,
                                            null);
        } finally {
            if (strOut != null) {
                try {
                    strOut.close();
                } catch (Exception genErr) {
                    /* Ignore*/
                }
            }
        }
    }
    private static final String BASE_DIRECTORY =
        "destination.ftp.base.directory";
    private static final String SUB_DIRECTORY_PATTERN =
"destination.ftp.subdirectory.pattern";
    private static final String SIDE_BASE_DIRECTORY =
        "destination.ftp.side.base.directory";
    private static final String SIDE_SUB_DIRECTORY_PATTERN =
"destination.ftp.side.subdirectory.pattern";
    private static final String TEMPLATE_FILE =
        "destination.ftp.template.path";
    private static final String FILE_PATTERN =
"destination.ftp.name.pattern";
    private static final String SIDE_FILE_PATTERN =
        "destination.ftp.side.name.pattern";
    private static final String TEMPLATE_CONTENTS =
        "destination.ftp.template";
    private static final String GENERATED_NAME =
        "destination.ftp.generated.file.name";
    private static final String GENERATED_SIDE_NAME =
        "destination.ftp.generated.side.file.name";
   private static final String FTP_SERVER =
"destination.ftp.server";
   private static final String FTP_USERNAME =
"destination.ftp.username";
   private static final String FTP_PASSWORD =
"destination.ftp.password";
    private static final String FTP_PORT =
"destination.ftp.port";}
```

MOCKPHASELISTENER IMPLEMENTATION

```
package oracle.documaker.ecmconnector.mockphaselistener;

import java.util.ArrayList;
import java.util.List;

import oracle.documaker.ecmconnector.connectorapi.PhaseListener;
import oracle.documaker.ecmconnector.connectorapi.Phases;
import
oracle.documaker.ecmconnector.connectorapi.data.ConfigurationData;
import oracle.documaker.ecmconnector.connectorapi.data.DocumentData;
import
oracle.documaker.ecmconnector.connectorapi.exceptions.ExceptionCodes
;
import
oracle.documaker.ecmconnector.connectorapi.exceptions.PhaseException
;

import org.apache.log4j.Logger;

public class MockPhaseListener extends PhaseListener {
    private static Logger logger =
        Logger.getLogger(MockPhaseListener.class.getName());
    private List<Phases> activePhases;

    public MockPhaseListener(ConfigurationData configurationData,
                             String phaseId) throws PhaseException {
        super(configurationData, phaseId);

        activePhases = new ArrayList<Phases>();

        String phaseListString =
            configurationData.getPropertyWithModifier(PHASE_LIST,
phaseId);

        if (phaseListString != null) {
            String[] phaseList = phaseListString.split(",");

            for (String phaseStr : phaseList) {
                Phases phaseInst = Phases.valueOf(phaseStr.trim());

                if (phaseInst != null) {
                    activePhases.add(phaseInst);
                }
                else
                logger.debug("Invalid phase id string - " + phaseStr);
            }
        } else
          throw new PhaseException("A list of phases must be provided
for proper configuration of the MockPhaseListener.",
                                     ExceptionCodes.CNT0500000002,
                                   new Object[] { "A list of phases
must be provided for proper configuration of the MockPhaseListener."
});
    }

    public List<Phases> getActivePhaseIdentifiers() {
        return activePhases;
    }

    public void execute(List<DocumentData> documentList,
                        Phases phaseId) throws PhaseException {
      logger.debug("Executing phase (list) - " + phaseId.toString());
```

```
    }

    public void execute(DocumentData documentData,
                        Phases phaseId) throws PhaseException {
        logger.debug("Executing phase (single) - " +
phaseId.toString());
    }

    public void execute(Phases phaseId) throws PhaseException {
      logger.debug("Executing phase (none) - " + phaseId.toString());
    }

    public void cleanUp() {
    }
```