opentext™

OpenText™ Documentum™ Content Management

Transformation Services Development Guide

Create and configure new plug-ins.

EDCCT250400-PGD-EN-01

OpenText™ Documentum™ Content Management
Transformation Services Development Guide
EDCCT250400-PGD-EN-01
Rev.: 2025-Nov-19

This documentation has been created for OpenText™ Documentum™ Content Management CE 25.4.
It is also valid for subsequent software releases unless OpenText has made newer documentation available with the product, on an OpenText website, or by any other means.

Open Text Corporation

275 Frank Tompa Drive, Waterloo, Ontario, Canada, N2L 0A1

Tel: +1-519-888-7111
Toll Free Canada/USA: 1-800-499-6544 International: +800-4996-5440
Fax: +1-519-888-0677
Support: https://support.opentext.com
For more information, visit https://www.opentext.com

Table of Contents

Chapter 1

Overview

This guide contains an overview of the OpenText Documentum Content Management (CM) Transformation Services architecture and the plug-in framework. It highlights the API methods and provides instructions and examples for creating and configuring new plug-ins.

This guide provides instructions for extending Transformation Services products through web services and new profiles to tailor transformation functionality to fulfill your organization's unique requirements.

This manual is intended for the person who is responsible for developing new transformation plug-ins for Transformation Services products. The reader is assumed to have a basic understanding of Java programming, XML, web service specifications, and the fundamentals of the Documentum platform.

1.1 About Transformation Services

Transformation Services is a suite of Server products that perform transformations and analysis on repository content. The Transformation Services functionality is available through Documentum client applications. Transformation Services consists of these products:

- OpenText™ Documentum™ Content Management Transformation Services - Documents
- OpenText™ Documentum™ Content Management Transformation Services - Media
- OpenText™ Documentum™ Content Management Transformation Services - Audio/Video
- OpenText™ Documentum™ Content Management XML Transformation Services

Transformation Services provides the following functionalities:

- *Transformations*

  Transformation Services carries out a wide range of conversions of documents, media, and audio/video content. For example, Microsoft Office documents into PDF and HTML formats. You can also apply watermarks, PDF overlays, headers, and footers to the transformed PDFs. Media files (images) can be transformed from one format into another with a wide range of options such as changes in the resolution, orientation, and so on. Audio/video files can be transformed from one format into another with options such as updating frame rate, bit rate, resolution, encoding, and so on, in the target files.

- *Metadata Analysis and Attribution*

  Transformation Services extracts attributes from content. Examples of extracted attributes are height and width of image files, and author and subject of documents. The extracted attributes are stored as metadata or rendition attributes. Some rendition attributes can be mapped to object-level attributes.

- *Enhanced Content Previews*

  Transformation Services generates thumbnails and storyboards to enhance the previewing experience of Microsoft Office documents, PDF, images and video files that are present in the repository.

- *Rendition Management*

  Transformation Services stores different formats of a repository file as renditions or related objects. Renditions are alternate formats of content that share the same object attributes and security. Related objects are complete objects on their own with independent metadata and security.

Transformation Services connects to OpenText Documentum Content Management (CM) Server, which means that Transformation Services is available to all applications and clients that are built on the Documentum API. This integration also means that many services are provided transparently to the client application: the server takes care of the work on the back end, and for the client it is business as usual. Thus, Transformation Services allows Documentum customers to content-enable their own applications.

Transformation Services is fully extensible through the standard Documentum API interface, Transformation Services, Transformation Services Software Development Kit (SDK), and JAVA API.

1.1.1 About the Transformation Services SDK

Transformation Services SDK, and the support offered by Transformation Services, provide an extensible plug-in framework that wraps current (and future) plug-in libraries. This framework allows you to implement greater format support in Transformation Services. You can wrap any media processing library and add it to Transformation Services through simple configuration, providing advanced media capabilities to media-enabled applications. Some of these capabilities include:

- Thumbnail generation for all file formats including new formats or proprietary formats
- Conversion between file formats
- Media editing capabilities
- Advanced media analysis. For example, media property extraction and video scene detection
- Dynamic assembly of PowerPoint presentations
- Automatic watermarking

1.1.1.1 Contents of the SDK

The Transformation Services SDK is a set of tools that allow programmers to add media processing technology to Transformation Services through development of new plug-ins and configuring them in the system. The SDK provides a standard interface to all plug-ins and contains the API, programming tools, and documentation that form the basis of the development environment.

The contents of the Transformation Services SDK are as follows:

- Transformation Services Plug-in SDK
- Content Transformation Local Service API (SBO service)
- Content Transformation Remote Service API (WebService)
- Samples

All Javadocs and samples are contained in one ZIP file, CTS_<release-version>_javadocs.zip.

1.1.2 Transformation Services SDK Supportability

Support for the Transformation Services SDK is not provided by Documentum Support. Creation of new Transformation Services plug-ins is beyond the scope of regular Transformation Services support. Help for modifying and creating new Transformation Services plug-ins can be provided by the OpenText Global Technical Services.

1.1.3 Transformation Services integration with other applications

Documentum products, such as Webtop, Digital Asset Manager, client, and xCP Viewer, take advantage of the inherent capabilities of Transformation Services; however, the extent of Transformation Services features exposed to each of these products varies. For example, Webtop can use Transformation Services to transform document files, through customer requests, to either PDF or HTML format. Digital Asset Manager on the other hand, can do the same, but can also utilize full functionality from Transformation Services to transform files automatically upon import, as well as through customer requests, from a wide variety of source formats to a wide number of output formats.

Other clients can use Transformation Services functionality as well. For example, a file can be checked-in to a v-enabled repository with My Documentum for Desktop Client. That file will then be thumbnailed automatically by the Transformation Services server, if enabled to do so.

Transformation Services provides a simple interface for developers to add advanced content-transformation capabilities to their Documentum-based applications. Developers can use a Documentum client-facing product that has been media-enabled, or they can customize their own applications through the Documentum API and Transformation Services.

The broad application of rich content and content management means that developers can quickly bring new vertical applications to market using Transformation Services. For example, a company can provide targeted applications in e-learning enriched with audio and video managed by Documentum. And in all scenarios, organizations can use Documentum to manage all of their digital assets as well as their Web content and documents.

1.2 Transformation Services architecture

Transformation Services is comprised of the following components:

Transformation Services

An extensible framework that allows for support of new and evolving content formats through the creation of plug-ins. Its simple interface exposes advanced media processing capabilities to application developers.

Functionality provided by plug-ins include thumbnail generation, file format transformations, media property extraction, storyboard generation, and format to format transformation.

OpenText Documentum Content Management (CM) Thumbnail Server

For secure, high-performance delivery of thumbnails. The Thumbnail Server is included with Transformation Services and is installed on the Documentum CM Server host.

Streaming Server integration

Provides low-latency delivery of video and audio files directly from the Documentum file store. It requires a third-party streaming server, such as Real Helix Universal or Windows Media Server. The *OpenText Documentum Content Management - Transformation Services Administration Guide (EDCCT250400-AGD)*provides detailed instructions for Streaming Server integration.

1.2.1 How Transformation Services works

When a rich media object is checked into a Transformation Services-enabled repository, the Documentum CM Server queues a Transformation Services request. Similarly, when a user chooses to perform a transformation on an object, Documentum CM Server queues a Transformation Services request. Transformation Services polls the queue for applicable items. When Transformation Services finds a queue item it retrieves the object referenced by the queue item and sends it to the appropriate plug-in for processing based on the object's media format. The plug-in evaluates the file, identifies and extracts properties, generates a thumbnail rendition, generates a low-resolution rendition, and transforms the file from one format to another if requested.

After the plug-in has processed the file, Transformation Services automatically discards the request object and saves the object with media properties and new renditions to the repository.

1.3 Plug-ins

A plug-in is a component of Transformation Services that transforms files to different formats according to the profile source and target formats. Plug-ins also create additional content from imported files, such as JPEG thumbnail representations and low-resolution renditions.

Plug-ins provide the format specific intelligence behind Transformation Services. Transformation Services plug-ins receive the delegated requests based on the formats they can handle. Each individual plug-in participates in this process by advertising the formats it can process, and implementing a simple Java interface that Transformation Services can invoke.

Transformation Services plug-ins operate as a service that implements a standard interface to supporting native libraries. This interface can evaluate files, identify properties, generate thumbnails, generate low-resolution renditions, generate storyboards, and transform files from one format to another. Each plug-in has well-defined properties, such as what type of media it can extract, process, and write.

Each plug-in can process multiple file formats. However, the elements that a plug-in can extract will differ between formats. For example, an image plug-in can extract properties such as width, height, and compression from image files of various formats, such as BMP, EPS, GIF, JPEG, and TIFF, and can convert images from one format to another. A video plug-in can extract properties such as frame rate, duration, and bit rate from video files.

Although v comes with a variety of standard plug-ins, there are a few instances where you want to create a new plug-in. For example, if a new file format emerges, or if you use a proprietary file format, you need to modify a plug-in or you need to create a new plug-in to handle these file formats. If you require a special integration with a best-of-breed library for special transformations, for example you can accomplish the integration through custom plug-ins. Independent software vendors who want to sell their file analysis engine as an out of the box integration with Documentum, can use the Transformation Services SDK to do this.

Transformation Services provides plug-ins to support most industry-standard rich media file types. Each Plug-in has different capabilities for different formats. The standard plug-ins can be edited or used as a template to create your own custom plug-ins.

The *OpenText Documentum Content Management - Transformation Services Administration Guide (EDCCT250400-AGD)*provides a complete list of plug-ins and profiles that invoke plug-ins that are available with this version of Transformation Services.

1.3.1 Plug-in framework

The Transformation Services plug-in framework is completely extensible. You can configure plug-in types and formats, and you can add new plug-ins to the system to support new media formats as required. The Transformation Services SDK is used to create and add your own custom plug-ins.

If you need to add transformation capabilities, file analysis, or the media-processing engine for a new custom application based on formats that are not currently supported by Transformation Services, you can use the SDK to accomplish this.

Figure 1-1 highlights the plug-in framework, which is represented by the plug-in Service, plug-ins and their interfaces to supporting libraries, and XML profiles.

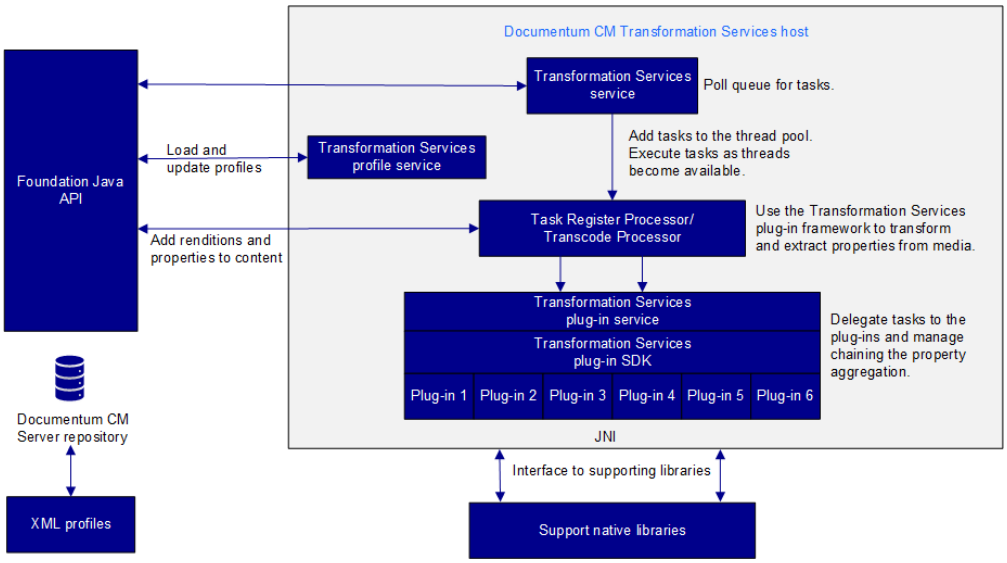

Figure 1-1: Plug-in Framework

1.4 The rendition model through Transformation Services

Documentum can manage multiple files in multiple formats as a single object. These files, known as renditions, can share descriptive metadata, but they also have their own unique metadata. Renditions are representations or copies of an object in which media format and properties can be different from the original. The rendition model is possible through the work of Transformation Services.

For example, a designer has just created a Photoshop file and wants to manage it within Documentum. When the designer checks the file in, Transformation Services automatically creates a thumbnail of the Photoshop file and stores it as a special rendition of the Documentum object. Because Photoshop files tend to be very large, and because not everyone in the design department has access to Photoshop, the

system also creates a full-size JPEG version of the file and stores it as a rendition. This allows colleagues and external business partners to view the image easily; they do not need excessive time or bandwidth to download the low-resolution rendition, and they do not need to have the creative tool (Photoshop) installed to view the original media. In this scenario, the Photoshop file is the primary rendition of the object, and the JPEG file and the thumbnail are alternate renditions. Each rendition has its own media properties, such as height, width, color mode, and compression, which can be displayed by a client interface. The renditions share the same object metadata, such as a description of the image, the copyright, keywords, and the object's lifecycle state.

When a file is versioned, its renditions, including any thumbnail renditions, are not carried forward with the new version of the file, unless your client system is configured to generate new renditions when a file is checked in and versioned.

Renditions of an object can be created manually, by a user by using transformations. For more information, see "Transforming objects" on page 13.

1.5 Transforming objects

Transformation Services allows users to transform media files from one format to another. In addition, users can edit file properties, such as width, height, color mode, and compression, using a parameterized, profile-based process.

For example, to resize an image, a user can select a Resize transformation profile which prompts them to enter the desired height and width. The transformation profile and its parameters are passed as a request to Transformation Services, which maps the request to the appropriate Plug-in and performs the transformation.

Transformation Services provides a set of predefined transformation profiles for most common operations and lets users extend transformation profiles and create new transformation profiles to meet their specific requirements. The following are examples of some transformation profiles:

- Creating a small JPEG will resize an image to fit within 200 x 500 pixels and save it as a JPEG rendition.
- Resizing the H x W JPEG will resize an image to a user-specified height and width and save it as a JPEG rendition.
- Adding a text layer can add a predefined text layer to an image and save it as a GIF rendition.

When a user chooses to perform a transformation, they will have the option (depending on the transformation selected) of creating a new, related object, or a new rendition.

1.5.1 How Transformation Services transform objects

Transformations occur both automatically and on a user-requested basis. Automatic transformations are performed when an object is imported into the repository, and a user has no interaction during the process provided the imported format is rich media enabled. User requested transformations are processed in a similar manner as import requests.

Each transformation request identifies the profile to be used. The request also includes any parameters entered by the user.

Transformation Services profiles are divided into three types. For each repository that is configured for Transformation Services, there are three folders that contain profiles and command line files.

The following is an indication of where transformation components are stored in the repository:

- `/System/Media Server/Profiles/`, stores profiles as dm_media_profile objects. These profiles are public and exposed by the client application, such as Digital Asset Manager. New profiles that you can create are added here.
- `/System/Media Server/System Profiles/`, stores profiles as dm_media_profile objects. These profiles are only used internally by Transformation Services and are not exposed through the client application. These profiles can be re-used in new profiles but must be referenced by an absolute path from the repository. These profiles are used mainly through object import, and the contents of this folder should not be modified.
- /System/Media Server/Command Line Profiles, stores command-line files as dm_document objects.

The user requested transformation process (illustrated in Figure 1-2) is as follows:

1. The application gets a list of available profiles and their required parameters by reading special system objects in the repository, in the folder /System/Media Server/Profiles as described in the preceding list.
2. The user requests a transformation for any rendition of a rich media object by selecting a profile and entering parameters if the profile requires parameters.
3. The application creates a transformation (TRANSCODE_CONTENT) apply call.
4. Documentum CM Server creates a queue object that requests the transformation. The request contains the profile ID, any parameter values required for a given operation. For example, height and width or angle of rotation, and the source and target formats.
5. Transformation Services polls the queue, finds objects for transformation, and matches the requested profile and source format to a particular plug-in.
6. Transformation Services locates the XML command line file for the profile, substitutes parameters as required, and invokes the plug-in to perform the transformation according to the selected profile and parameters.

The transformation occurs in the background.

7. When the transformation is complete, Transformation Services sends the transformed media back to Documentum CM Server and updates the original object with the new rendition and its associated media properties, which are stored as attributes of the rendition. If the object already has a rendition of the same format, the new rendition replaces the previous one, unless it is given a unique page modifier.
8. Transformation Server removes the transformation request item from the queue.

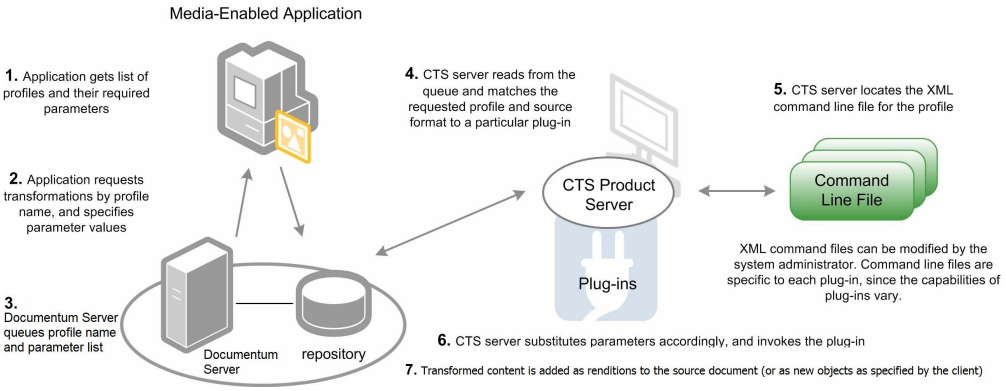

Figure 1-2: The Transformation Process

1.5.2 Transformation Services Local Service API

The creation of transformation requests, as described in "Transforming objects" on page 13, involves the use of the native DMCL apply method to invoke the TRANSCODE_CONTENT event. This process has been augmented with a Java based API that utilizes BOF (Business Object Framework) technology to allow any Java based client to easily create Transformation Services requests. The Local Service API is based upon OpenText™ Documentum™ Content Management Foundation Java API and BOF, so usage of it allows for interaction with other objects and events in the Documentum system. Also it does not need to be collocated with the Transformation Services and can be run from any client environment provided OpenText Documentum Content Management (CM) Foundation Java API can be installed there.

The API is based upon Transformation Services profiles, the creation of requests, and the submission of those requests. No knowledge of the Transformation Services queuing process is necessary because the API abstracts those details behind the interfaces that are used to request transformations. The application developer can work in Java with strongly typed interfaces that clearly define all of the required objects needed to expose the full capabilities of Transformation Services in their custom client application. All that is needed is the inclusion of the .jar files containing the API interface in the applications classpath, and a Transformation Services-enabled repository to connect to. Furthermore, this API is already in use

providing transformation functionality to the Documentum client applications such as Webtop, Digital Asset Manager, and Web Publisher, so it is a tested and proven API.

"Using the Transformation Services Local Service API" on page 29 describes in greater detail how to use the Transformation Services Local Service API.

1.5.3 Transformation Services Remote Service API

Transformation Services Remote Service API, otherwise called as WebService API, is based on OpenText™ Documentum™ Content Management Foundation SOAP API.

1.5.4 Transformation Services SDK

Transformation Services provides all of its intelligent content functionality through the use of Transformation Services plug-ins. These components are Java based components implementing a well-defined Java interface that can be configured and added to a Transformation Services installation so as to provide functionality for particular formats of content. For example, Transformation Services currently deploys distinct plug-ins for handling image, video, or document content. The Transformation Services framework handles the system configuration, the receipt and delegation of transformation requests, the coordination of the various plug-in interactions and other useful services.

The Transformation Services plug-in API is the name of this collection of Java interfaces and classes. It is distinct from the Content Transformation API which is used by client applications to make the transformation request. Another difference is that Transformation Services plug-ins must run on the same server as the Transformation Services is deployed on. A plug-in's function is to receive files and details about the request to do work with them, whether it is a request to transform them to a different format or a request to analyze and return their distinguishing properties. In just the same way as Documentum engineering adds new format capabilities to Transformation Services by developing Transformation Services plug-ins, a customer can develop new format capabilities by implementing their own custom Transformation Services plug-in.

This guide provides a detailed process describing how one can go about creating, testing and installing a Transformation Services plug-in. "Creating and installing plug-ins" on page 43 provides detailed instruction. "Example: Creating a TEXT plug-in" on page 47 uses a sample Transformation Services Plug-In to make clear what can be accomplished using this API and demonstrates the best practices in implementing this functionality.

1.5.5 Tasks

In addition to the ability to implement Transformation Services plug-ins to provide format specific functionality, it is possible for developers to provide customized business logic around their interactions with the repository and its objects. Included as part of the Transformation Services plug-in SDK, the Java implementation of this business logic is in the form of a task, which can be deployed on the Transformation Services framework and configured for use with one or more profiles.

Some common and useful applications of a task include changing how content is stored in the repository, updating the metadata of objects based on a transformation, processing the content input to or returned from the transformation, or integrating the transformation with any kind of workflow that can be invoked from Java. With access to a user session and all the data required for transformation tasks it should be possible to implement whatever custom business logic is needed.

Chapter 2

Working with profiles

This chapter describes how to customize and create profiles to match your business needs or workflow.

2.1 Understanding transformation profiles

Profiles are XML files that an application passes to the transformation server when requesting a specific functionality. Essentially, profiles package the transformation functionality for the client application. Profiles define the operations of the plug-ins. Therefore, while a profile is not specific to a single plug-in, the profile is specific to the functionality it provides to the plug-in.

Transformation Services predefined profiles cover most of the common media transformations, such as resize, flip, and rotate. Transformation Services also enable system administrators to modify existing profiles or create new profiles.

Transformation profiles can be parameterized. When the user invokes a parameterized profile, the user is prompted for the parameters required by the profile. For example, a resized profile can include height and width or percentage parameters.

The parameters required by a profile can be assigned validation rules, where the application validates the values entered by the user. The validation rules can check the unit of measure, format, or similar metrics to avoid processing incorrect values.

Each profile contains the profile name, a label (the text that displays in the client application), a description, required attributes, and supported source/target dm_format objects. For example, a resize profile will contain the attributes width and height and any number of supported source and target formats, as in the following examples:

Table 2-1: Profile source and target formats

format source = tiff	target = tiff
format source = tiff	target = jpeg
format source = tiff	target = gif
format source = gif	target = gif
format source = gif	target = bmp

Profiles for which all parameters are defined contain only the name, label, description, and supported source and target formats. Each profile is linked to an actual XML command-line file for a given transformation.

2.1.1 Reusing transformation profiles

Transformation Services allows you to reference other profiles when you create your own Transformation profiles. You can reuse profiles by two methods: chaining and sequencing.

Chaining Profiles

Chaining profiles invoke each profile within them in stages. Transformations also occur one at a time, as the transformation result is required for the next transformation. The result of a chaining profile would be the output of the last profile in the chain.

Sequencing Profiles

Sequencing profiles specify a list of profiles that are executed in parallel. For each profile in the profile sequence, you can specify whether the next profile should wait on the successful completion of the previous task, although this scenario is rare and only used for debugging purposes. If there is no specification, profiles can be executed as soon the tasks threads are available to process them. This means that each profile does not have to be complete before the next profile starts.

2.1.2 Profile types

Profiles are divided into the following two types:

- System profiles
- User profiles

Note: System and user profiles are in different locations by default, but do not have to be. System profiles are internally used. User profiles are used by users in applications.

2.1.2.1 System profiles

System profiles are dm_media_profile objects. You can locate System profiles from the repository at //System/Media Server/System Profiles. These profiles are only used internally by Transformation Services and are not exposed through the client application. You can customize these profiles and also use them to create new profiles but the new profiles must be referenced by an absolute path from the repository.

When rich media-enabled files are imported or checked in to the repository, they are first processed by the *register* profile. This process is called registration. The register profile acts as an initiator, directing the way in which imported content is handled. Only those profiles explicitly referenced in the register profile are invoked on import.

You should now be able pre-configure rendition requirements of clients and the existing legacy capabilities of Transformation Services into the register profile (`register.xml`). You can externalize the rendition requirements of the clients into their respective profile configuration files so that their specific requirements can be configured as needed. In addition, it provides you a mechanism to selectively turn on/off the rendition generation for the clients on a per-repository basis.

2.1.2.2 User profiles

User profiles are dm_media_profile objects. You can locate user profiles from the repository at //System/Media Server/Profiles. These profiles are exposed by the client application, such as Digital Asset Manager. You can add new profiles that you manually invoke in user profiles. These types of transformations give users the flexibility to choose specific files in the repository, and specify parameters (such as header and footer information, watermarks, job options, and security settings) during the transformation.

2.1.2.3 Command line files

Command line files are located in the repository at //System/Media Server/Command Line Files. Command line files are dm_sysobject objects. Command line files and their commands are used by plug-ins that process them.

2.1.3 Invoking a profile

When richmedia_enabled contents are imported or an application request for a transformation it creates a queue item for dm_mediaserver user.

To invoke a transformation you must create a queue item for Transformation Services. An API call creates that queue item – simply a short cut for creating Transformation Services jobs. This call should be executed using an Documentum API tool, such as iapi32.exe (on Windows).

You need to specify an object id for the transformation, the object's format, its page (usually 0), and target format. The API keyword is TRANSCODE_CONTENT. The bulk of the job's attributes goes into the MESSAGE part of the apply call, such as profile id describing the transformation, as well as any other attributes that exist in the profile and are necessary for the transformation to complete.

The following is an example of the apply call to create a thumbnail for the video file:

```
apply,c,09bc614e80005105,TRANSCODE_CONTENT,PAGE,I,0,SOURCE_FORMAT,S,mpeg,MESSAGE,
S,'-profile_id 08bc614e8000152b',TARGET_FORMAT,S,jepg_th
```

The following is an example of the apply call to execute the `rotate` profile:

```
apply,c,09bc614e80005105,TRANSCODE_CONTENT,PAGE,I,0,SOURCE_FORMAT,S,gif,MESSAGE,
S,' -profile_id="080f3aeb80017c6f" -render="true" -target_page_modifier="rotate_45"
 -notifier_user="matrix" -arg_object_id="090f3aeb8001ad40" ',TARGET_FORMAT,S,jepg,
 PRIORITY,I,5
```

Note that in the second example, there is a special argument, arg_object_id, which is a reference to the text in the XML command line file containing the remainder of the

attributes required for the transformation, since the MESSAGE part is limited to 250 characters.

Transformation Services orders the queue list by PRIORITY, so assigning a higher priority (between 1 and 9) results in a job being processed ahead of jobs with lower priority.

2.1.4 Enabling a user profile

When modifying user profiles, ensure that the following tags are present in the profile file:

- <Filter name="CTSProduct" value="<*Documentum Content Transformation Services product installed*>"/>

  Note: The value for "CTSProduct" should be the name of one of the Transformation Services products configured to your repository from the cts_instance_info table.

- <Filter name="Visibility" value="Public"/>

  The profile must be *Public* to be visible to users.

Here is an example from a user profile:

```
<Filters>
    <Filter name="CTSProduct" value="MTS"/>
    <Filter name="Visibility" value="Public"/>
</Filters>
```

2.1.5 Example of generation upon import

The following is an example of how Transformation Services processes thumbnails for gif format files upon import:

1. When you import content (in this case, a GIF format file), the Documentum CM Server checks if the content is rich media enabled. Only rich media enabled content is added to the media server queue. REGISTER_ASSET admin method is used to add the queue item.
2. Transformation Services checks the queue every 10 seconds by default. When Transformation Services finds a queue item, it signs off a maximum of 10 items by default. If there were 100 items and one Transformation Services instance running against the repository, this instance signs off all the items in 10 steps.
3. After the first sign-off is done, Transformation Services starts to process the items using the register profile. It does not remove an item from the queue until the transformation is done or has failed.

   The following is an example of the register_legacy profile to illustrate the gif thumbnail creation:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Profile SYSTEM "ProfileSchema.dtd">
<Profile name="register_legacy" label="Register Profile(Legacy)"
```

```
description="Automatically process imported files(Legacy)"
related_objects_only="false">
    <Formats>
        <! -- Properties Pairs -->
        <Format source="bmp" target="bmp"/>
            <Format source="dng" target="dng"/>
        <Format source="eps" target="eps"/>
        <Format source="fpx" target="fpx"/>
        <Format source="gif" target="gif"/>
            <Format source="html" target="html"/>
            <Format source="jpeg" target="jpeg"/>
            <Format source="jpeg2000" target="jpeg2000"/>
            <Format source="jpeglres" target="jpeglres"/>
            <Format source="png" target="png"/>
        <Format source="photoshop3" target="photoshop3"/>
        <Format source="photoshop5" target="photoshop5"/>
        <Format source="photoshop6" target="photoshop6"/>
            <Format source="photoshop7" target="photoshop7"/>
            <Format source="photoshop8" target="photoshop8"/>
            <Format source="pdf" target="pdf"/>
            <Format source="illustrator11" target="illustrator11"/>
            <Format source="illustrator10" target="illustrator10"/>
            <Format source="illustrator9" target="illustrator9"/>
        <Format source="macp" target="macp"/>
        <Format source="pbm" target="pbm"/>
        <Format source="pcd" target="pcd"/>
        <Format source="pdf" target="pdf"/>
        <Format source="pgm" target="pgm"/>
        <Format source="pict" target="pict"/>
        <Format source="pnm" target="pnm"/>
            <Format source="ppm" target="ppm"/>
            <Format source="excel8book" target="excel8book"/>
            <Format source="excel8template" target="excel8template"/>
            <Format source="excel12book" target="excel12book"/>
            <Format source="excel12template" target="excel12template"/>
...
    </Formats>

    <Transcodings>
        <Transcode name="register_legacy" label="Register Profile
            (Legacy)"/>
    </Transcodings>

    <ProfileSequence>
    ...
        <InnerProfile path="/System/Media Server/System Profiles/thumbnail"
        waitOnCompletion="false">
            <InnerTokenMapping LocalProfileToken="jpeg_th" InnerProfileToken=
            "doc_token_targetFormat" Literal="true"/>
        </InnerProfile>
...
    </ProfileSequence>
</Profile>
```

Note: *Source* is the input format and *Target* is the output format.

For register and extract property profiles, the source and the target will be the same.

ProfileSequence: Provides a sequence for content to be processed.

InnerProfile: Points to a profile to be used to process the content.

“waitOnCompletion” is used by chained and sequence profiles.

InnerTokenMapping: Provides the value and the token to be passed to InnerProfile.

4. Transformation Services loads the thumbnail profile based on the inner profile path in the register profile.

   The following is an example of the thumbnail profile to illustrate the GIF thumbnail creation:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Profile SYSTEM "ProfileSchema.dtd">
<Profile name="thumbnail"  label="Thumbnail Generation
Profile"
description="Profile for generating a thumbnail">
    <Formats>
        ...
        <Format source="gif" target="jpeg_th"/>
        ...
    </Formats>

    <Transcodings>
        <Transcode name="imageSize" label="Resize image">
        </Transcode>
    </Transcodings>

    <CommandFilePath mptype="IMAGE3">
        System/Media Server/Command Line Files/thumbnail_imw.xml
    </CommandFilePath>
...
<CommandFilePath mptype="POWERPOINT">
        /System/Media Server/Command Line Files/thumbnail_ppt.xml
    </CommandFilePath>
    </Profile>
```

   Note: In the thumbnail profile, the Source is GIF and Target is jpeg_th. When the rendition is created it will be added as “jpeg_th” format rendition, which is the thumbnail.

5. Transformation Services tries each plug-in in order until a plug-in accepts the transformation. Transformation Services then uses the command line file to perform the transformation on the relevant plug-in. After the transformation is complete, it will exit from that profile. The following is an example of the Image3 thumbnail command line file:

```
<?xml version="1.0" encoding="UTF-8"?>
<IMW_MP_PROPERTIES>
    <OutputFormat>JPEG</OutputFormat>
    <Pages>0</Pages>
    <PROP name="Width" type="unsigned long" token="doc_token_width_tn">100</PROP>
    <PROP name="Height" type="unsigned long" token="doc_token_height_tn">100</PROP>
    <PROP name="Density" type="unsigned long" token="doc_token_dpi">96</PROP>
    <PROP name="enlarge" type="string" token="doc_token_enlarge">false</PROP>
    <Options>-limit memory doc_token_limit_memory -limit map doc_token_limit_map
-limit area doc_token_limit_area -density doc_token_dpi doc_token_sourceFile
-flatten -quality 80 -strip -filter Lanczos -profile "doc_token_cmyk_profile"
-profile "doc_token_rgb_profile" -profile "doc_token_colorProfile"
-colorspace doc_token_colorSpace
    </Options>
</IMW_MP_PROPERTIES>
```

   Note: The text highlighted in boldface are commands to the Image3 plug-in.

Plug-in capability is specified in the config files for each plug-in. These files can be located in plug-in folders, under the *<Documentum Content Transformation Services>*\config\ folder. For example, the Image3 config files are located in *<Documentum Content Transformation Services_Install_Dir>*\config\image3\image3.xml). When Transformation Services is loaded it will create a table in the memory with mapping. The dtd is also located in the same folder and must not be modified.

6. v sends the request to the Image3 plug-in to process the thumbnail.
7. Transformation Services removes the queue item and adds the thumbnail rendition to the content.

2.2 Understanding command line files

When the target plug-in is identified, Transformation Services translates the parameterized profile into a command line file for the plug-in. Figure 1-2 provides more information. The command line file contains detailed instructions for the plug-in. It also contains additional parameters that are not user-selectable. For example, a profile can be written to create a thumbnail that always produces JPEG output. The resulting file format and many of its attributes (such as JPEG quality and color mode) can be controlled at the command level. Command line files are specific to each plug-in, since the capabilities of plug-ins vary.

Default Transformation Services command line files are stored in the repository as XML files that can be used by the Transformation plug-ins as needed. Developers can create their own command line files in any format they wish, as long as the plug-in they create can understand the content. (As long as the content can be extracted as text, for example.)

Command line files do not need to be created for new transformation profiles that re-use other profiles. A transformation profile that serves only to chain or sequence other profiles has no need for CommandFilePath entries.

2.2.1 Transformation Services DTD

All Transformation Services profiles are written in XML. An external document type definition (DTD) reference (ProfileSchema.dtd) defines the legal structure of profile files. All profile files must conform to the Transformation Services DTD.

As defined by the DTD, all XML files consist of:

- elements
- attributes
- entities

The Transformation Services DTD file:

```
<!ELEMENT Profile (Formats, Types?, MediaEnabled*,
Filters, Transcodings, (CommandFilePath+ | ProfileChain | ProfileSequence))>
```

```
<!-- Profile attributes -->
<!ATTLIST Profile
    xmlns:dctm CDATA #IMPLIED
   name CDATA #REQUIRED
   label CDATA #REQUIRED
   related_objects_only (true | false) "false"
   notify_result (true | false) "false"
   operation (transform | extractProperties) "transform"
   taskImpl CDATA "com.documentum.cts.impl.services.task.CTSTask"
   description CDATA #IMPLIED
   dctm:obj_status CDATA #IMPLIED
   dctm:obj_id CDATA #IMPLIED
   dctm:version_label CDATA #IMPLIED
>

<!ELEMENT Formats (Format+)>
<!ELEMENT Format EMPTY>
<!-- Format attributes -->
<!ATTLIST Format
   source CDATA #REQUIRED
   target CDATA #REQUIRED
>
<!-- Types and MediaEnabled nodes are specified just for backward compatibility.
These nodes should be converted to Filters -->
<!ELEMENT Types (Type*)>
<!ELEMENT Type (#PCDATA)>
<!-- Format attributes -->
<!ATTLIST Type
   name CDATA #REQUIRED
>
<!ELEMENT MediaEnabled (#PCDATA)>
<!ATTLIST MediaEnabled
    category CDATA #REQUIRED
>
<!-- Types and MediaEnabled nodes are specified just for backward compatibility.
These nodes should be converted to Filters -->

<!ELEMENT Filters (Filter+)>
<!ELEMENT Filter EMPTY>
<!-- Filter attributes
    Predefined filter names: CTSProduct, AppProduct, Visibility, XMLApp,
ObjectType, VirtualDocOnly
-->
<!ATTLIST Filter
   name CDATA #REQUIRED
   value CDATA #REQUIRED
>

<!ELEMENT Transcodings (Transcode*)>
<!ELEMENT Transcode (Parameter*)>
<!-- Transcode attributes -->
<!ATTLIST Transcode
   name CDATA #REQUIRED
   label CDATA #REQUIRED
>
<!ELEMENT Parameter ((ValueList | ValueRange | Value* | TupleElement+ |
ContentObject+), (depends-on-param | depends-on-format)?)>
<!-- Parameter attributes -->
<!ATTLIST Parameter
   name CDATA #REQUIRED
   label CDATA #REQUIRED
   description CDATA #IMPLIED
   controltype (list | range | text | text-block | color-picker |
object | listbox) #REQUIRED
   datatype (string | integer | float | hex | tupleSequence | content) #REQUIRED
   dql CDATA #IMPLIED
   default CDATA #IMPLIED
   isRequired (true | false) "true"
>
<!ELEMENT ValueList (Value+)>
<!ELEMENT Value (#PCDATA)>
```

```
<!-- Value attributes -->
<!ATTLIST Value
   label CDATA #REQUIRED
>

<!ELEMENT ValueRange (MinValue, MaxValue)>
<!ELEMENT MinValue (#PCDATA)>
<!ELEMENT MaxValue (#PCDATA)>

<!ELEMENT depends-on-param (Value+)>
<!ATTLIST depends-on-param
   param-name CDATA #REQUIRED
   action (hide | maintain_proportion ) #REQUIRED
>

<!ELEMENT depends-on-format (ValueList)>
<!ATTLIST depends-on-format
   format-name (source | target) #REQUIRED
   action (hide | show ) #REQUIRED
>

<!ELEMENT TupleElement (#PCDATA)>
<!ATTLIST TupleElement
   label CDATA #REQUIRED
   name CDATA #REQUIRED
   datatype (string | integer | hex | date) #REQUIRED
>

<!ELEMENT ContentObject EMPTY>

<!-- The remaining elements are for server-side use only.  They should NOT be
     parsed by application other than Media Services. -->
<!ELEMENT CommandFilePath (#PCDATA)>
<!-- CommandFilePath attributes -->
<!ATTLIST CommandFilePath
   mptype CDATA #REQUIRED
>

<!ELEMENT ProfileChain (InnerProfile+)>
<!ELEMENT ProfileSequence (InnerProfile+)>

<!ELEMENT InnerProfile (InnerTokenMapping*)>
<!-- waitOnCompletion will be ignored of the parent element of InnerProfile is
     ProfileChain.  It is intended for use only with ProfileSequence -->
<!-- InnerProfile attributes -->
<!ATTLIST InnerProfile
   path CDATA #REQUIRED
   waitOnCompletion (true | false) "true"
   useTargetFormat (true | false) "true"
   innerProfileId CDATA #IMPLIED
   forClient CDATA #IMPLIED
>
<!-- forClient is required for registering assets based on clients-->

<!ELEMENT InnerTokenMapping EMPTY>
<!-- InnerTokenMapping attributes -->
<!ATTLIST InnerTokenMapping
   LocalProfileToken CDATA #REQUIRED
   InnerProfileToken CDATA #REQUIRED
   Literal (true | false) "false"
>
```

Chapter 3

Using the Transformation Services Local Service API

3.1 Transformation Services Local Service API

The Transformation Services Local Service API is designed to invoke transformation services to be executed by Transformation Services. This API works with the following Transformation Services elements: **user sessions**, **profiles**, **system objects**, **content**, and **transformation requests** including **formats** and **parameters**. The following figure illustrates the sequence of events between these objects that make it possible to accomplish the task of invoking Transformation Services.

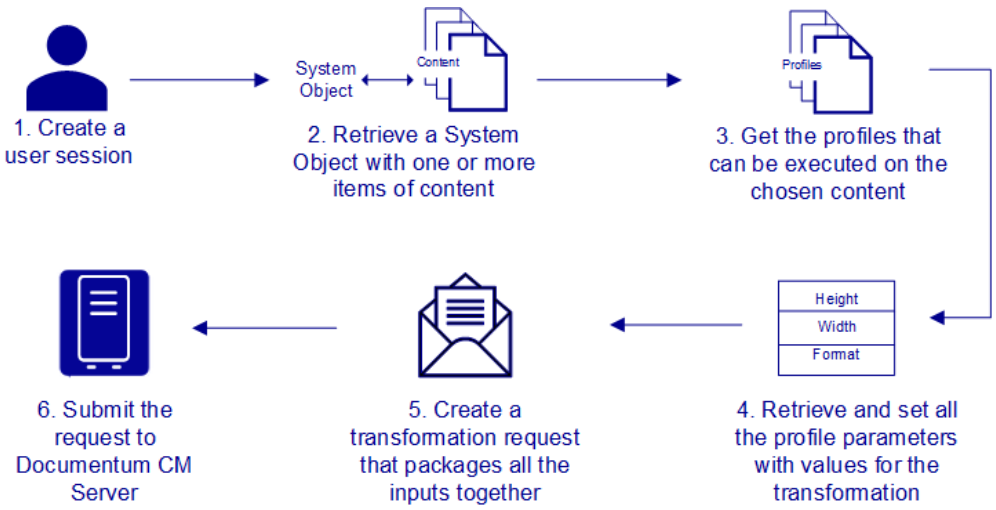

Figure 3-1: Transformation Service execution by Transformation Services API

The process begins with a user session logged into the repository against which Transformation Services is configured, and containing content that needs to be transformed or analyzed by Transformation Services. Depending on the context, this user session can belong to a user that is manipulating a custom application and browsing objects. It can also be a non-interactive custom application that logs a configured user into a repository. In either case, the application must identify some object in the repository that has attached to it some content for sending to Transformation Services. The content can be the object's primary content or another rendition; either is valid.

After the content is found, the operation to be performed on that content must be identified. These potential actions are encapsulated by profiles, each of which

describes a capability that has been deployed by one of the Transformation Services plug-ins configured on that repository, running under Transformation Services. In the case of an interactive client, a user can then be presented with a list of these profiles, and be asked to choose which one they would like to execute on their content. In the case of a non-interactive application, the list of profiles can be searched for a configured profile by name. After it is chosen, the profiles parameters must be filled in. The profile is interrogated for its parameters and after these are retrieved, their values are set. Also, the format for the output of the transformation must be decided.

Now that the object, content, profile, parameters and output format has been explicitly setup by the user session, it is possible to make a request. To do this it is necessary to construct a transformation request object to package all the inputs together. The request will encapsulate all of the selected data, and can be submitted to execute on the Transformation Services server. After the request is submitted it can run asynchronously in Transformation Services, and the result will ultimately be stored in the repository and a notification sent to the requesting user. Alternatively, the submission of the request can be made with a flag that prevents the method from returning until the request is completed.

3.2 Using the Transformation Services Local Service API

The following sections detail how to use the Transformation Services Local Service API.

3.2.1 Prerequisites

The Transformation Services Local Service API requires that the same version of the OpenText Documentum Content Management (CM) Foundation Java API that is used by the current version of Transformation Services that you are using, be installed on the machine where an application using the Transformation Services Local Service API is going to be run or compiled. The BOF implementation of the Transformation Services Local Service API is closely tied to Foundation Java API, and many of the objects correspond directly to objects in the repository. In addition, to run a custom application using the Transformation Services Local Service API it is required to have Transformation Services installed and configured for the repository the application will connect to. This is because the repository must be updated with the required types and data used by the Transformation Services Local Service API, and because transformations will not execute unless there is Transformation Services available to run the request.

3.2.2 Running the Transformation Services Local Service API

The Transformation Services Local Service API is written in Java, making it necessary to include the code in your application's classpath. The interface codes can be found in two .jar files: `ctsTransform.jar` and `dam_services.jar`. The implementation jars are downloaded from the repository during run time.

3.3 Making API calls

At the beginning of this chapter a typical interaction between Transformation Services Local Service API objects was depicted, which resulted in the execution of a transformation request for content stored in the repository. What follows is some sample code that can be used to execute the steps described earlier.

The creation of sessions will be omitted as this is a standard Foundation Java API programming exercise. After the session is available it will be necessary to instantiate the Service Based Objects (or SBOs) which deliver the required objects to the caller:

```
ICTSProfileService m_profileService;
IDfClientX clientX = new DfClientX();
        IDfClient client = clientX.getLocalClient();
        IDfSessionManager sessionManager = client.newSessionManager();
        IDfLoginInfo loginInfo = clientX.getLoginInfo();
        Properties props = new Properties();

        props.load(getClass().getResourceAsStream(TEST_CONFIG));

        loginInfo.setDomain((String)props.get(DOMAIN));
        loginInfo.setUser((String)props.get(USER_NAME));
        loginInfo.setPassword((String)props.get(PASSWORD));

        m_repository = (String)props.get(REPOSITORY);
        sessionManager.setIdentity(m_repository, loginInfo);

        m_profileService = (ICTSProfileService)client.newService
    (ICTSProfileService.class.getName(), sessionManager);
ICTSAddJobService theCTSService = (ICTSAddJobService)client.newService
    (ICTSAddJobService.class.getName(), sessManager);
```

This code returns instances of both the IProfileService and the ICTSService interfaces. The following samples make use of these interfaces for retrieving profiles for execution, and for getting and submitting requests for transformation.

After the services are instantiated, profiles should be retrieved for a given rendition of a system object. As before, the instantiation of an object will be left as implied:

```
ICTSProfileRequest pr = m_profileService.newProfileRequest();
        String profileId = getProfileId();
        pr.setProfileId(profileId);
        log(1, "### GetProfiles with profileId=%s", profileId);
 ICTSProfile[] profiles = m_profileService.getProfiles(m_repository, pr);
```

to get profile by id:

```
ICTSProfile profile = m_profileService.getProfileById(profileId);

to get profile by name
ICTSProfileRequest pr = m_profileService.newProfileRequest();
pr = m_profileService.newProfileRequest();
```

```
        String profileName = "register";
        pr.setProfileName(profileName);
ICTSProfile[] profiles = m_profileService.getProfiles(m_repository, pr);
```

In the following code, we begin by making sure that Transformation Services is available for the repository being used. If it is, we should expect profiles to be returned by the getProfiles(…) method. In addition, one of the parameters to the method, the String[] categories allows the results to be further filtered. In this case only the profiles with the assigned category Public will be returned, so that profiles reserved for internal system use are not exposed. Other categories are available as described in the Javadocs for the IMediaProfile interface.

After the profiles are retrieved, it should be possible to display them to users as options for transformation, or if the application is non-interactive, to inspect the array for the profiles to be executed. The profiles have names, descriptions and the valid format combinations available through accessors so that this decision can be made. After the profile for execution is chosen, the parameters that can be passed during execution should be retrieved:

```
IMediaProfile profile = (IMediaProfile)getDfSession()
.getObject(profileId);
IProfileParameter[] profileParameters = profile.getParameters();
```

The IProfileParameter[] describes the options which users can choose for the transformation. The array can be traversed and these options displayed. Regardless of the application paradigm, these objects must be used to set the values on the request. Before setting any values the transformation request should be instantiated using the ICTSService:

```
ICTSRequest theRequest = theCTSService.getNewCTSRequest(theRepoName);
        IMediaProfile theProfile = getMediaProfile(theSession, PROFILE);
        String theProfileId = null;
        if (theProfile != null)
        {
            theProfileId = theProfile.getObjectId().toString();
            int userPermit = theProfile.getPermit();
            if (userPermit >= IDfACL.DF_PERMIT_READ)
            {
                System.out.println("User Has Read Permission On Current Profile");
            }
            else
            {
                System.out.println("User Does Not Have Read
                        Permission On Current Profile");
            }
            System.out.println("Current Media Profiles: "
                        + theProfile.getObjectName());
        }
        if (theProfileId != null)
        {
            //String placeholderDoc = null;
            if( theSourceObjectId != null )
            {
                theRequest.setSourceObjectId(theSourceObjectId);
            }
            if( strRelatedId != null )
            {
                theRequest.setRelatedObjectId(strRelatedId);
            }
            theRequest.setMediaProfileId(theProfileId);
            theRequest.setSourceFormat(SOURCE_FORMAT);
            theRequest.setTargetFormat(TARGET_FORMAT);
```

```
            theRequest.setSourcePageModifier("");
            theRequest.setTargetPageModifier("");
            theRequest.setLocale(java.util.Locale.getDefault());
            theRequest.setPriority(1);
            theRequest.setSourcePage(0);
            theRequest.setTargetPage(0);

            IProfileParameter[] profileParameters = theProfile.getParameters();
            // here you can pass in all the parameter values that are required for
                        the transformation of this specific profile
            String value_doc_token_direction = "vertical";

            for (int i=0; i < profileParameters.length; i++)
            {
                String currentParamName = profileParameters[i].getParameterName();

                if (currentParamName.equals("doc_token_direction"))
                {
                    profileParameters[i].setParameterValue
                            (value_doc_token_direction);
                }
            }
            theRequest.setParameters(profileParameters);
            theRequest.setMediaProfileName(theProfile.getString("object_name"));
            theRequest.setMediaProfileName(theProfile.getString("title"));
            String notify_result = theProfile.getNotifyResult();

            if(notify_result != null)
            {
                theRequest.setNotifyResult(notify_result);
                System.out.println("Notify Result: " + notify_result);
            }
            theRequest.setDeleteOnCompletion(false);
            theRequest.setTransformationType("asynchronous");
            theRequest.setZipOutput(true);
            theRequest.setStoreResultInDocbase(true);
            theRequest.setTargetFolder(null);
```

The attributes of the ITransformRequest can be set based on user inputs:

```
transformRequest.setSourceObjectId(m_ObjectId);
transformRequest.setMediaProfileId(m_ProfileId);
transformRequest.setSourceFormat(m_SourceFormats);
transformRequest.setTargetFormat(m_TargetFormat);
transformRequest.setRelatedObjectId(m_RelatedObjectId);
transformRequest.setSourcePageModifier(m_SourcePageModifiers);
transformRequest.setTargetPageModifier(m_NewDesc);
transformRequest.setLocale(LocaleService.getLocale());
transformRequest.setPriority(1);
transformRequest.setSourcePage(0);
transformRequest.setTargetPage(0);

IMediaProfile profile = (IMediaProfile)session.getObject
(new DfId(m_ProfileId));
```

To assign parameters to the request an instance of the chosen profile should be retrieved, and its parameters supplied to the request:

```
IMediaProfile profile = (IMediaProfile)session.getObject(
new DfId(m_ProfileId));
transformRequest.setParameters(profile.getParameters());
```

When the setParameters(…) method is called, the profile parameters are immediately cloned. As a result, any modification of parameters in the future to set their values on the request, are completely independent of the values of the parameters on the profile. This way the profile object can be reused for separate

transformations without having its values set through parameter modifiers called during previous transformations.

The request can also be set with the name and label to be used for display, based on the same values of the profile being executed:

```
transformRequest.setMediaProfileName(profile.getString(
"object_name"));
transformRequest.setMediaProfileLabel(profile.getString
("title"));
```

After the transform request is properly set with required data, it can be submitted for execution using the ICTSService instance:

```
theJobId = theCTSService.addJob(theRequest);
```

3.4 Enabling Transformation Services for Vault

Vault allows Transformation Services to securely access credentials required for running Batch scripts. Depending on your configuration, enable any one of the following Vaults:

- HashiCorp Vault
- Kubernetes native secrets

3.4.1 Prerequisite

The required entries have been added to the Vault server enabling the Batch script for the configured Vault.

- If Vault has been configured with HashiCorp Vault:

  INSTALL_OWNER_PASSWORD

  Key: <*repository_name*>, Value: <*repository_password*>

  For example: Key: `testenv`, Value: `Password@123`

- If Vault has been configured with Kubernetes native secrets:

  INSTALL_OWNER_PASSWORD

  Secret name: <*InstallOwnerPassword*>, Value: <*repository_password*>

  For example: Secret name: `dmadminPassword`, Value: `Password@1234567`

3.4.2 Using the Batch script

Transformation Services also supplies a utility for creating transformation requests that do not require any programming. This Java utility is available on the same machine that Transformation Services is installed. It consists of a simple batch script to invoke the utility, an XML based configuration file which allows the user to select the inputs for the transformation as well as which profile to run and its parameters, and the Java code that implements it.

The utility can be found at the following location on the Transformation Services machine: `C:\<Documentum Content Transformation Services_INSTALL_DIR>\CTS\docbases\<DOCBASE_NAME>\CTSServerScript`

The batch script can be found in the bin subfolder and is named script.bat. This script is configured at the time the Transformation Services is configured for the repository and typically does not require modification. The only exception to this is in the case where the user wants to run the utility with a new or modified XML script. The batch file is initially configured to run the following XML script file:

```
DSCRIPT_CONFIG_DIR=C:\<Documentum Content Transformation Services_INSTALL_DIR>\CTS
\docbases\<<DOCBASE_NAME>>\
    CTSServerScript\config\script.xml
```

This default XML script invokes the utility for its initial intended purpose: to have Transformation Services generate the renditions and media properties for all applicable content in the repository that predated the configuration of Transformation Services. For example, after running this script all images in the repository will have thumbnails generated for them if Transformation Services is configured against the repository. This script file is a good example for investigation into the configuration of the utility.

The root node of a valid XML script file is <MediaServicesScript>. Following this is the specification of the profile that will be run by the script:

```
<Profile path="/System/Media Server/System Profiles/register" >
```

This profile is the one that is run by Transformation Services any time that rich media enabled content is imported into the repository. That is, any content that a presently configured Transformation Services is capable of processing.

Following this is the login information which is necessary because the utility must connect to the repository to create transformation requests. The user that is configured for the utility will be the same one that runs the transformations and receives notification:

```
<LoginContext username="Administrator" passwordfile="C:\<CTS_INSTALL_DIR>\CTS\docbases
\<DOCBASE_NAME>\CTSServerScript\config\file\p1.txt"
    docBase="<DOCBASE_NAME>" domain="" aekFilePath="C:\<Documentum Content
Transformation Services_INSTALL_DIR>\CTS\config\aek.key"/>
```

Note that this entry refers to a `p1.txt` file which contains the user's password. The password can be encrypted using the `aek.key` file specified in the `script.xml`.

By default, the system operator, who is configured to run this utility, has the access to run the configurator of Transformation Services.

By default all applicable documents will be processed:

```
<MaxDocs>
        <All/>
</MaxDocs>
```

It is also possible to restrict the number of requests that will be made by setting an actual number, specified using the NumDoc XML element:

```
<MaxDocs>
        <NumDoc 250/>
</MaxDocs>
```

This is useful in the case where a very large number of documents will be selected for the transformation, and Transformation Services will only be allotted a limited amount of time for batch processing, during off-peak hours for example. In this way the batch requests can be made and completed in time for the system to become available for high priority user tasks.

The sample `script.xml` uses a DQL query for choosing the inputs to transformation:

```
<DQL_Query query="select distinct a.r_object_id,
a.a_content_type from
dm_document a, dm_format b  where a.a_content_type
= b.name and
    b.richmedia_enabled = 1 and  not exists (select 1
from dmr_content_r c
    where a.r_object_id = c.parent_id and  c.i_full_format
= 'jpeg_th') " >
```

Using DQL is the most flexible way in which to select inputs because the user can tailor the query exactly to their needs. The preceding query selects rich media enabled documents that do not yet have a thumbnail so Transformation Services will generate them. The only restriction when using DQL is that the object id of valid dm_sysobject's must be present in the result set of the query. When using the DQL option one can specify the desired target format for the transformation:

```
<Format source="jpeg" target="gif"/>
```

The Format indicates that the request will transform the jpeg format rendition of all the input documents to the output format gif. Optionally one can set the attribute overwrite="true" or "false" which will allow an existing rendition to be overwritten if it is of the same format, or prevent the same. By default, this attribute is set to "false".

Instead of using a Format element one can simply specify the element:

```
<SameAsInputFormat />
```

This element when used will result in the primary content of the documents being chosen, and the same format being submitted as target format. This combination is not generally useful except in the particular case of the register profile which can create multiple outputs of different formats, and only takes as valid input source and target formats that are the same.

The Format element can also be used as an alternative to the DQL_Query element. This will result in documents being chosen for input that have a content_type that matches the "source" value of the Format. This can be a very large set of documents so it should only be used when the user wants to transform all documents that have the format "jpeg" for example. The "target" attribute of the Format element will again determine the output of the transformation for the requests.

The other valid option for selecting the inputs is:

```
<All_Richmedia_Enabled TargetFormat="">
```

This will create a set of input documents similar to the DQL_Query given in the preceding example, with the difference being that the input documents will not only be those without thumbnails, but all that can be processed by the configured Transformation Services. The result will be an even larger set of inputs.

If the chosen profile contains parameters it is possible to set their values:

```
<Parameters name="Height" value="200">
```

The name must match the "name" of the parameter elements in the profile being executed so that the values will be set properly.

The requests being submitted can also be assigned a priority with a value between "1" and "9", where "9" is the highest priority:

```
<Priority value="5"/>
```

This is useful in the case where a large batch of requests for migrating older content is submitted. The priority for these can be set to "1" so that any incoming requests from users will be executed without being delayed by the batch process. For example the Digital Asset Manager sets the priority to its user's requests at "5", while this script uses the default value of "1".

This script utility also logs all of the requests it submits for review. The location of the log file can be specified in the script:

```
<LogFile>C:\<Documentum Content Transformation Services_INSTALL_DIR>\CTS\docbases
\<DOCBASE_NAME>
\CTSServerScript
\logs</LogFile>
```

Run the `setPassword.bat` file located at `\CTSServerScript\bin\`. This updates the repository user password in the `p1.txt` file located at `\CTSServerScript\config\pfile\` in the following format:

For HashiCorp Vault: `INSTALL_OWNER_PASSWORD/`*`<repository_name>`*

For Kubernetes native secretes: *`<installOwnerPassword>`*

3.5 Transformation Services real-time SBO Java client setup

1. Install OpenJDK 17.x or later in your workstation.
2. Install a Java IDE. For example, Eclipse.
3. Create a new Java Project.
4. Copy the Foundation Java API and Transformation Services libraries to the client workstation from your Transformation Services host and add them to the project's Class/Build path.

   The following Foundation Java API libraries can be found in the Transformation Services host at `C:\Documentum\Shared\`:

   - `All-MB.jar`
   - `aspectjrt.jar`
   - `bpmutil.jar`
   - `ci.jar`
   - `collaboration.jar`
   - `commons-codec-<packaged-version>.jar`
   - `commons-lang-<packaged-version>.jar`
   - `configservice-api.jar`
   - `configservice-impl.jar`
   - `dfc.jar`
   - `DmcRecords.jar`
   - `dms-client-api.jar`
   - `jaxb-impl.jar`
   - `log4j-core-<packaged-version>.jar`
   - `jakarta.activation-api.jar`
   - `jakarta.servlet-api.jar`
   - `jakarta.xml.bind-api.jar`
   - `jaxb-core.jar`
   - `json.jar`
   - `log4j-api-<packaged-version>.jar`
   - `log4j-1.2-api-<packaged-version>.jar`
   - `messageArchive.jar`
   - `messageService.jar`

- subscription.jar
- workflow.jar
- xml-apis.jar
- xtrim-server.jar

The Transformation Services library, ctsTransform.jar, can be found in the Transformation Services host at:

C:\Documentum\CTS\lib\

5. Create or copy from a Transformation Services host the dfc.properties file and place it in the project's Class/Build path. For example,

```
dfc.data.dir=C\:/Documentum
dfc.search.ecis.enable=false
dfc.search.ecis.host=
dfc.search.ecis.port=
dfc.tokenstorage.dir=C\:/Documentum/apptoken
dfc.tokenstorage.enable=false
dfc.docbroker.host[0]=<Docbroker_Host_or_IP>
dfc.docbroker.port[0]=<Docbroker_Port>
dfc.globalregistry.repository=<Docbase>
dfc.globalregistry.username=dm_bof_registry
dfc.globalregistry.password=<Bof_Registry_Password>
```

6. Create the log4j.properties file and put it in the project's Class/Build path. For example,

```
rootLogger.level=WARN
rootLogger.appenderRefs=A1, F1
rootLogger.appenderRef.A1.ref=STDOUT
rootLogger.appenderRef.F1.ref=File
monitorInterval=5

#------------------- CONSOLE --------------------------
appender.A1.type=Console
appender.A1.name=STDOUT
appender.A1.layout.type=PatternLayout
appender.A1.layout.pattern=%d{ABSOLUTE} %5p [%t] %c - %m%n
appender.A1.filter.threshold.type=ThresholdFilter
appender.A1.filter.threshold.level=ERROR

#------------------- FILE -------------------------
appender.F1.type=RollingFile
appender.F1.name=File
appender.F1.fileName=C\:\\Documentum\\logs\\log4j.log
appender.F1.filePattern=log4j.%d{yyyy-MM-dd}
appender.F1.layout.type=PatternLayout
appender.F1.layout.pattern=%d{ABSOLUTE} %5p [%t] %c - %m%n
appender.F1.policies.type=Policies
appender.F1.policies.time.type=TimeBasedTriggeringPolicy
appender.F1.policies.time.interval=1
appender.F1.policies.time.modulate=true
appender.F1.policies.size.type=SizeBasedTriggeringPolicy
appender.F1.policies.size.size=10MB
appender.F1.strategy.type=DefaultRolloverStrategy
appender.F1.strategy.max=5

#---

appender.FClientAppender1.type=RollingFile
appender.FClientAppender1.name= ClientAppender1
appender.FClientAppender1.fileName=C:/ctsws_client/DFCSBOClient/config/
Realtime_SBO_Client_log.txt
appender.FClientAppender1.filePattern=C:/ctsws_client/DFCSBOClient/config/
```

```
Realtime_SBO_Client_log.%d{yyyy-MM-dd}-%i
appender.FClientAppender1.layout.type=PatternLayout
appender.FClientAppender1.layout.pattern=%d{ABSOLUTE} %5p [%t] %c - %m%n
appender.FClientAppender1.policies.type=Policies
appender.FClientAppender1.policies.time.type=TimeBasedTriggeringPolicy
appender.FClientAppender1.policies.time.interval=1
appender.FClientAppender1.policies.time.modulate=true
appender.FClientAppender1.policies.size.type=SizeBasedTriggeringPolicy
appender.FClientAppender1.policies.size.size=10OMB
appender.FClientAppender1.strategy.type=DefaultRolloverStrategy
appender.FClientAppender1.strategy.max=5

logger.RClientAppender1.name = com.emc.documentum.cts.lb
logger.RClientAppender1.level = INFO
logger.RClientAppender1.additivity = false
logger.RClientAppender1.appenderRef.FClientAppender1.ref = ClientAppender1
```

Similarly we can add logger and appender for:
com.documentum.services.cts.impl.transform
com.emc.documentum.cts.common.

7. Create a folder. For example, `realtimeclient_config` in the client file system with the following structure and files:

   a. Create an empty `cache` folder. This will be used in the file `preferences.xml` as the value for ServerProperty Cache.

   b. Create an empty `pfile` folder.

   c. Copy the file `aek.key` from a Transformation Services host (`%CTS%\config\`) to the empty pfile folder. This will be used in `preferences.xml` as part of <AekFilePath>.

   d. Copy the file `mspassword.txt` from a Transformation Services host (`%CTS%\docbases\<your_docbase>\config\pfile`) to the `pfile` folder. This will be used in `preferences.xml` as ServerProperty passwordFile.

   e. Create a `preferences.xml` file with following contents. Ensure to update the bold-faced values with relevant path/repository values:

```
<?xml version="1.O" encoding="UTF-8"?>
<ServiceConfiguration ID="CTS Web Services">
<PropertyList>
<ServerProperty Key="Cache" Description="The Temperory Cache Directory"
     Value="C:/realtimeclient_config/DFCSBOClient/config/grx64/cache" />
<ServerProperty Key="AllowDirectTransfer" Description="Allow Direct File
     Transfer From CTS Server to Client. Set it to false if there is a firewall
     restriction" Value="true" />
<ServerProperty Key="CTSWSPingInterval" Description="Interval (in seconds)
     used to specify how frequent the LB should ping its CTS instances for
heart
     rate." Value="3O" />
<ServerProperty Key="FailoverRetries" Description="Allow a number of retries
     if a request fails while waiting on the HTTP response from CTS"
Value="1" />
<ServerProperty Key="FailoverWait" Description="Wait between failover retries"
     Value="1"/>
<ServerProperty Key="InstanceSelector" Description="Specify an implementation
     class for instance selection " Value="com.emc.documentum.cts.lb.workers.
    OccupancyBasedSelector"/>
<ServerProperty Key="CTSOccupancyPollingInterval" Description="Specify
     occupancy polling interval in seconds" Value="7"/>
<ServerProperty Key="ConnectionRetries" Description="Specify connection
     retries (in case Repositories section is not configured )" Value="1O"/>
<ServerProperty Key="AvailabilityRetries" Description="Number of retries when
     CTS instances are not available" Value="2"/>
<ServerProperty Key="AvailabilityWait" Description="Number of seconds to wait
```

```
     for rechecking availability" Value="4"/>
<!-- if local load balancer is used, no need of CTS-WebServices -->
<LoadBalancer type="local" URL="" sendMode=""/>
<!-- Otherwise, a remote CTS-WebServices can be used as Load Balancer, for
    ex: -->
<!-- <LoadBalancer type="remote" URL="http://10.31.158.35:8080/services/
    transformation/LoadBalancer" sendMode="remote"/> -->
<Repositories>
<AekFilePath>C:/realtimeclient_config/DFCSBOClient/config/grx64/aek.key</
AekFilePath>
<LoginContext DocbaseName="lrx64">
<ServerProperty Key="domain" Value=""/>
<ServerProperty Key="userName" Value="Administrator"/>
<ServerProperty Key="passwordFile" Value="C:/realtimeclient_config/
DFCSBOClient/
config/grx64/pfile/mspassword.txt"/>
<ServerProperty Key="maxConnectionRetries" Value="10"/>
</LoginContext>
</Repositories>
</PropertyList>
</ServiceConfiguration>
```

8. Create a `transformation.properties` file with the following contents and place it in the Client Projects Root folder. For example, `C:\eclipse\workspace\CTSRealTimeDFCTestHarness`. This will hold the `preferences.xml` path:

```
#cts ws preferences config location
CTSWSConfig=C:/realtimeclient_config/DFCSBOClient/config/grx64/preferences.xml
```

Note: If the client project cannot find `transformation.properties/preferences.xml`, it will throw an error in the specified log file as per `log4j2.properties` entries, with the exact path where the client project is expecting the `transformation.properties` or `preferences.xml` files.

Chapter 4

Creating and installing plug-ins

This chapter provides an overview of the steps necessary to create and configure your own Transformation Services plug-ins.

4.1 Prerequisites

Ensure that Transformation Services is installed, and that the Transformation Services repository is configured and accessible.

4.1.1 Transformation Services installed and configured for repository

Ensure that Transformation Services is installed, and Transformation Services repository is configured and available.

4.2 Creating a plug-in

Every plug-in must implement the ICTSPlugin interface (refer to JavaDocs for more information). To understand how this interface should be implemented, this guide will use the example of a simple plug-in called “Text”. The focus in this section is to show how to implement the interface, process profiles, and interact with Transformation Services. To support this goal, our example plug-in does not use any complex (or useful) media processing technologies. The plug-in can use some powerful media libraries, and after you understand the basics, it should be quite easy to expose the functionality you desire.

When you create a plug-in, you must implement the interface and make the appropriate configuration updates to enable it. After it is created, it can be added to the system configuration as a fully qualified class name and accompanying path to its configuration file, if required.

4.2.1 Creation process overview

The process for creating a plug-in includes three main steps, as follows.

Plug-in Creation Process

1. Implement the ICTSPlugin Java interface.
2. Write the profiles that specify the operations you want to be executed on your new plug-in.
3. Write command line files that specify the input/output data and operational commands for your plug-in.

4.2.2 ICTSPlugin Interface

The starting point in the creation of any plug-in is through the implementation of the ICTSPlugin interface.

4.2.3 Creating a profile

Transformation Services plug-ins are based on profiles. After the creation of a new plug-in you must configure and generate profiles and command line files. This section describes how to configure an XML profile.

To create a profile:

1. To create a profile you must create an XML file that conforms to the `ProfileSchema.dtd` which can be found in the repository along with the Transformation Services profiles.

   Appendix C, Transformation profiles on page 77 of this document or the profiles located in the Transformation Services configured repository provides examples.

2. Include the name of the plug-in, as returned by the getName() method, in the profile as the mptype attribute of the CommandFilePath element as shown in the following example. This entry is used to determine which plug-in should be invoked to process a profile.

   ```
   <CommandFilePath mptype="IMAGE3">
   ```

   In the preceding example, the plug-in with the name “IMAGE3” will process the profile.

4.3 Testing and installing a plug-in

After creating a new plug-in and before installing a new plug-in on a production environment, it is important to test the plug-in.

4.3.1 Testing process overview

After creating a new Plug-in and before installing it on a production environment, it must be tested in a development environment. Plug-ins must be installed on a test Transformation Services system (on a non-production system). Ensure your Java IDE for development and debugging is installed on the same system.

4.4 Installing a plug-in

After you create a new plug-in, you must make the appropriate configuration updates to enable it. You must add the plug-in to the system configuration in two instances: in the `CTSPluginService.xml` file, and in the profile. This section describes both configurations.

4.4.1 Installation process overview

The process for configuring a plug-in includes five main steps, as follows.

To install a Transformation plug-in:

1. Import your profile and command line file into the Transformation Services configured repository.
2. Modify the Transformation Services configuration `CTSPluginService.xml` file to be aware of the new Plug-in.
3. Add the new Transformation Plug-in code to the Transformation Services installation and product classpath.
4. Test the new profile through execution and verify the results.

4.4.2 Import your profile

After creating your profile and command-line file, it is necessary to import them into the repository, for use by the Transformation Services configured application. For example, to import profiles using Documentum Administrator:

1. Logon to Documentum Administrator and navigate to the location.
2. Select **File** > **Import** > Select the custom profile xml file on our system.
3. Select the Type as **Document(dm_document)**.
4. Select the Format as XML Document.
5. Click **Finish**.
6. Select the MediaProfile from the dropdown list.
7. Click **OK**.

4.4.3 Configuring the CTSPluginService.xml file

The `CTSPluginService.xml` configuration file is located in the folder *<CTS_INSTALL_DIR>*`\config\`. In the file, each CTSPlugin element corresponds to an instance of a Transformation Plug-in in Transformation Services as shown in the following code sample from the `CTSPluginService.xml` file.

```
…
<CTSPlugin DELEGATE_CLASS="com.documentum.cts.plugin.powerpoint1.PowerPointPlugin"
CONFIGFILE="C:\DOCUME~1\CTS\config\powerpoint1\
powerpoint1.xml"/>

<CTSPlugin CONFIGFILE="C:\DOCUME~1\CTS\config\image3\
    image3.xml" DELEGATE_CLASS="com.documentum.cts.plugin.imw.IMWPlugin"/>

</CTSPluginList>
…
```

After you create a new Transformation Plug-in, you must configure it in the `CTSPluginService.xml` file.

To configure the CTSPluginService.xml file:

1. Open the `CTSPluginService.xml` file in an editor.
2. Add an entry for your plugin including the path to its configuration file and ensure that the value of the DELEGATE_CLASS attribute that is the fully qualified name of the class which implements the ICTSPlugin interface.
3. Save and close the file.
4. Restart the Transformation Services service.

Upon startup, an instance of this class will be constructed using the default constructor. The ICTSPlugin method loadConfiguration() will then be invoked passing the value of the CONFIGFILE attribute as an argument.

After you have configured the `CTSPluginService.xml` file, the Transformation plugin will be available to receive transformation requests. The format of the configuration file to be passed in the loadConfiguration() depends on the implementation of the Transformation Plug-in. Transformation Services does not parse these files.

Chapter 5

Example: Creating a TEXT plug-in

This chapter uses an example Transformation plug-in to demonstrate the process of writing a Transformation Plug-in.

5.1 Overview of example – TEXT Plug-in

The example Text Transformation plug-in will try to accomplish some very simple support for text files. It will be capable of taking text files from any of three different operating systems (Macintosh or Windows) and converting them to the format of one of the other operating systems. This is accomplished by switching the characters used for end-of-line (EOL) to the ones appropriate for the target Operating System.

The Text plug-in will also be capable of extracting two basic properties: the number of characters in the file, and the number of white space delimited tokens in the file. The result will be added as metadata to text content.

The Text plug-in example will illustrate the use of profiles, command line files, transformation and property extraction, and the interactions required by Transformation Services of any Transformation plug-in.

5.2 Identify the Transformation plug-in

The first step in creating your Transformation plug-in is to identify it as such. The ICTSPlugin interface contains two methods that are important for identification:

```
String getName()
String getVersion()
```

These methods return String values that allow the instance of the Transformation plug-in to be identified. The returnable values are left to the discretion of the developer of the plug-in.

getVersion()

The value returned by the getVersion() method is useful for keeping track of plug-ins as they evolve over time, and while Transformation Services does not currently require the value returned by this method, it would be advisable to implement it.

getName()

The getName() method is very important. Transformation Services will use the value returned by this method when it attempts to delegate the execution of profiles to the appropriate plug-ins.

Appendix C, Transformation profiles on page 77 contains several examples of profiles. Notice the following entry in each profile:

```
<CommandFilePath mptype="TEXT">
```

When Transformation Services processes this profile, they will look for a plug-in that returns the case-sensitive String value “TEXT” from its getName() method.

The values returned by these methods can be contained within configuration files, and read by the plug-in at the time of initialization. This makes deployment of the plug-in more flexible. This way, it is possible for a single type of plug-in to be deployed more than once to a single Transformation Services instance, simply by changing the value returned by getName() and making the appropriate profile changes. Things like load balancing, and altering the behaviors of the plug-in can be easily accomplished this way. Sample code for the CTSPluginConfig class used by the Text plug-in is included in Appendix C, Transformation profiles on page 77.

5.3 Executing a profile

When the Transformation Services has found the plug-in that should execute a profile it executes the plug-in’s executeProfile() method. An example of the executeProfile () method follows:

```
public ICTSResult[] executeProfile(URL[] mediaURLs,
          ICTSCommandLineFileContent clfContent,
          String[] inputFormatExtensions,
          String outputFormatExtension,
                          boolean clearOnExit)
            throws CTSPluginException;
```

The Text plug-in tests the inputs to this method. For example, all of the actions it can perform require input files, therefore it checks to ensure that the URLs passed are in the mediaURL’s parameter:

```
if(mediaURLs == null || mediaURLs.length == 0)
{
throw new CTSPluginException("Cannot transform a null URL");
}
```

After checking the inputs, the plug-in creates a File object based on the first element of the URL[]. Certain assumptions are made here: there will be only 1 input file, and the URL passed in only supports the “file” protocol, meaning that the input file will only be accessible using the local file system. The Text plug-in only handles this scenario; so further logic is not implemented.

It is at this point that the Text plug-in first attempts to analyze the ICTSCommandLineFileContent that specifies the operation to perform. This class returns 2 pieces of data; the “arguments” which specify the parameters set on the operation by the user executing the profile, and the “content” of the command line file associated with the operation. We will see what the arguments look like later. First, we will need to look at the “content” of the command line file.

5.3.1 Command line file content

The Text plug-in has implemented its different file conversion and property extraction capabilities in separate, useful private methods. To know which one it should call, it needs to analyze the ICTSCommandLineFileContent "content" which we have decided will be implemented as XML. The structure of command line files is entirely up to the developer of the Transformation plug-in. Transformation Services just manages the relationship between profiles and command line files so that it can pass the correct command line file to the plug-in being used. The XML will conform to one of 4 DTDs: TEXT_MP_TO_WINDOWS, TEXT_MP_TO_MAC, TEXT_MP_TO_UNIX and TEXT_MP_EXTRACT_PROPERTIES (these are shown in Appendix D, Command Line files on page 79). The helper method getCommandLineDTDType() uses a DOM XML parser to determine the DTD that is declared in the command line file. The content of the XML is not really important yet; in this simple case we are just setting the type of operation based on the DTD.

If the getCommandLineDTDType() method returns the string TEXT_MP_EXTRACT_PROPERTIES, the algorithm calls the helper method extractProperties(), returning an ICTSResult[] which executeProfile() returns to the caller.

If the getCommandLineDTDType() method returns one of the other three DTDs, we will call one of three other helper methods implemented to do transformations. Because a single folder location is used for temporarily storing the output files, a simple String generation technique using the Random class is used to create the output file's name:

```
Random random = new Random();
String outputFileName = _mpConfig.getCache() + "output_text" + random.
nextInt() + ".txt";
File outputFile = new File(outputFileName);
```

After this file is generated, it can be passed to the appropriate transformation method. There are three methods implemented, each of which corresponds to the three profiles we have created: convertToWindowsCRLF(), convertToMacCR(), convertToUnixLF(). (See the following example.) These also all return an ICTSResult[] to the calling method.

The ICTSResult has two fields that a plug-in can populate on return from the executeProfile() method. The first is a single URL, typically used to indicate the file that is the result of a transformation. Transformation Services currently expects the file to be located on the same machine, protocols other than "file" is not currently supported. The second is an IContentAttrValue[] (a Foundation Java API class) that contains name-value pairs that constitute the metadata properties that result from the operation. A plug-in must construct concrete instances of these interfaces, for the ICTSResult, use the CTSResult implementation that is described in the Transformation Services JavaDocs. The class ContentAttrValue should be used to instantiate properties.

The ICTSResult's returned can contain a variety of results. If the URL is not null, and there are properties in the IContentAttrValue[], Transformation Services will add the

content specified by the URL as a rendition of the target document, and apply the properties of the IContentAttrValue[] to that same rendition. If the URL is null and there are properties present in the IContentAttrValue[], Transformation Services will add the properties to the source rendition for the executeProfile() call. If there is a URL present, but no properties, the file is simply added as a rendition without metadata. An ICTSResult without valid URL or IContentAttrValue[] (where both are null or property array has no elements) is not considered a valid return, and will result in an error.

5.4 convertToWindowsCRLF and others

The three methods used for transforming text files for each operating system are very similar. We will not focus on the algorithm used to change the EOL characters. We are trying to illustrate the concept of file transformation in the context of Transformation Services and plug-ins, so the algorithm for processing the text file is very simplistic. For this example, we will only consider the convertToWindowsCRLF() method since the others are virtually the same. In this case, all EOL characters are replaced with the Windows EOL which is a carriage return (the character '\r', specified in this class by the variable CR) immediately followed by a line-feed (the character '\n', specified in this class by the variable LF).

The method takes two file parameters, one representing the input, and the other one representing the output. We begin by declaring the return ICTSResult[], as well as a FileReader, and a FileWriter which we will use to process the input and output.

```
ICTSResult[] mpResults = null;
FileReader reader = null;
FileWriter writer = null;
```

We will use a while loop to iterate through the characters in the input file. The FileReader returns –1 when the end-of-file (EOF) is reached, and this will set a boolean value that is used as a condition of the while loop termination.

```
boolean done = false;
while(!done)
{
    int aChar = reader.read();
    // The -1 character indicates EOF.
    if(aChar == -1)
    {
         done = true;
    }
```

Every character returned must be processed, and we are concerned with only three conditions. If the character is a carriage-return, we either have an EOL character for a Macintosh file, or we are processing a Windows file, where the next character will be the line-feed. We handle either case if we run into a CR:

```
if(aChar == CR)
{
    // Write a carriage return and line feed for Windows
end-of-line
    writer.write(CR);
    writer.write(LF);
    // Check if this is already a Windows end-of-line.
    int anotherChar = reader.read();
    if(anotherChar != LF)
```

```
    {
        writer.write(anotherChar);
    }
}
```

The second condition we are interested in is the case where the character is a line-feed (LF). This should indicate a Unix input file. We will just replace this with a CR and a LF.

```
else if(aChar == LF)
{
    // Write the next character
    writer.write(CR);
    writer.write(LF);
}
```

The third condition is the catch-all for the case where the character is any other character. Since it is not an EOL character, we just write it to the output file.

```
else
{
    writer.write(aChar);
}
```

After the while loop terminates, we have the desired output. Now we must simply marshal the results for return by assigning our null return variable an CTSResult[] instance of size 1, since there is one file to return. We also invoke the constructor of the CTSResult passing the path to the local file, and an empty IContentAttrValue[]. Note that we can very easily invoke the extractProperties() method on the output file to populate the IContentAttrValue[] with useful metadata which would be added to the rendition. The extractProperties() will be explored in the next section.

```
// Create the ICTSResult[] with the output file and an
empty array of
// IContentAttrValue's
mpResults = new CTSResult[1];
mpResults[0] = new CTSResult(outputFile.getAbsolutePath(),
new IContent
AttrValue[0]);
```

When working with files it is very important to handle errors very carefully, and perform proper cleanup. If care is not taken, many error conditions can arise, such as an inability to delete files that were not closed, running out of available file handles on the operating system and memory leaks to name a few. All of the code in this method that can throw an exception is wrapped in a try .. finally block. The finally block is where we close the FileReader and FileWriter objects, which should perform all the necessary cleanup.

```
finally
{
   try
   {
       // Put the FileReader and FileWriter close() calls in
a finally
 // block to make sure it is done.
       reader.close();
       writer.close();
   }
```

5.5 extractProperties

The extraction of properties is typically a process that involves the analysis of a file that produces metadata. Depending on the type of file, properties can either be stored in the file, like the name of the file's author in its header, or can be the result of running an algorithm on the file, like a calculation of the number of frames in a video based on the frame rate and total duration. In either case, the properties generated should be useful because they can be used to uniquely identify content, or provide insight on how to use it.

In our Text plug-in, we implement a very simple algorithm that extracts two properties: the total number of characters in the file, and the total number of tokens in the file (that is, the number of words). This is a fairly trivial example, though these properties are commonly generated by today's word processing applications, so are likely to be useful to a limited extent. Again, the goal in this case is to illustrate the concept of property extraction, and the way a plug-in should generate and return properties. Also, this example will show how to write and use profiles that can take parameters, and behave differently based on them.

The profile for property extraction with the Text plug-in is called text_extract_properties, and appears in full in Appendix C, Transformation profiles on page 77. This is the first profile that takes parameters: doc_token_extract_chars, and doc_token_extract_tokens. The allowed values for these parameters are "YES" and "NO", so they are essentially boolean arguments which indicate whether the plug-in should extract and generate the properties. As a result, execution of the profile should result in either 1 or 2 properties being generated, at the discretion of the client application or user. [1]

To determine which properties to extract, our method must analyze the parameters passed by the user and the command line file associated with the text_extract_properties profile being executed.

All of this information is encapsulated in the ICTSCommandLineFileContent parameter passed to the executeProfile() method. As mentioned earlier, this class returns two pieces of data; the "arguments" which specify the parameters set on the operation by the user executing the profile, and the "content" of the command line file associated with the operation. The command line file, called text_extract_properties_clf, is an XML file that has two interesting characteristics. First, its DOCTYPE is TEXT_MP_EXTRACT_PROPERTIES, and this is used by the plug-in to identify the operation, and call this method. Second, it contains two "PROP" nodes, each of which corresponds to the parameters that can be passed in with the execution of the profile. Appendix C, Transformation profiles on page 77 provides more information.

[1] A URL should not be returned in the single ICTSResult of the ICTSResult[]. The format inputs to profiles meant only to perform property extraction should have identical source and target formats, which both indicate the source of the operation. For example, Format source="text" target="text". This allows Documentum Transformation Services to check the input content to the operation, and submit the correct file to the plug-in. While the plug-in technically can return a URL, it will be added as a rendition to the target document (in this case, also the source), and will overwrite any content of the same format and page_modifier. This is potentially destructive, and at the very least, is redundant.

First we assign the two ICTSCommandLineFileContent members to String variables:

```
// Parse the Command Line File Content to determine
what properties to
// extract.
String args = clfContent.getArguments();
String content = clfContent.getContent();
```

The arguments should be passed into the method in the following format:

```
"-doc_token_extract_chars="YES" -doc_token_extract_tokens="YES""
```

Then we use a utility method to return the “arguments” as a Java Properties object:

```
Properties argsAsProps = parseAttributes(args);
```

Now we can check to see if the user has specified with a parameter, whether the number of tokens should be extracted:

```
String countTokens = getValue(argsAsProps, _count_tokens_string);
if(countTokens != null && countTokens.equals(YES))
{
    shouldCountTokens = true;
    propArraySize++;
}
```

The variable YES has a value of “YES”. This is the value we look for, and if found, sets the boolean variable shouldCountTokens to true. The int variable propArraySize is initialized at 0, and is incremented for each property we must extract. This will determine the size of the returned IContentAttrValue[].

We also check if the value of the parameter was passed as “NO”. If so, we set the boolean flag, and property array counter appropriately.

If the parameter was not passed, we will look at the command line file content for the default value. Because it is part of the command line file, an administrator can change the default behavior of this profile without changing the plug-in code. Rather, the command line file can be checked out of the repository and altered as required. We have implemented a helper method to read the XML of the command line file, and return the String value of a node with a given name. We use this to get the default value:

```
// Call a helper method to parse the CLF content for
the default value
// of a node with given name.
String defaultCountTokens = getDefaultValueForNodeName(
_num_tokens_name, content);
if(defaultCountTokens!= null && defaultCountTokens.equals(YES))
{
    shouldCountTokens = true;
    propArraySize++;
}
```

At this point we know whether or not we should count the tokens and return this property. The exact same process is executed for the character count property. It is possible that there are no properties to return at this point, so we create an empty array for the return value and exit the method.

```
// In the case that there is nothing to process, just return
if(!shouldCountChars && !shouldCountTokens)
```

```
{
    results[0] = new CTSResult(new IContentAttrValue[0]);
    return results;
}
```

If there are properties that need to be returned, we initialize the array of based on the size specified when analyzing the parameters:

```
IContentAttrValue[] attrValues = new IContentAttrValue
[propArraySize];
```

The algorithm for counting characters uses a while loop very similar to the one described in "convertToWindowsCRLF and others" on page 50. All the characters are read from a FileReader and a counter is incremented for each one unless they are from the standard set of white-space delimiters ('\n' '\t' '\r' '\f'). Regardless of whether the character token count is requested, we will execute this algorithm. The reason is that we need to read the file content into a StringBuffer for submission to the StringTokenizer class, for determining the total number of tokens property. If we have gotten this far, at least one of the properties is requested, and in either case we must read through the file.

After the while loop is terminated, close() is called on the FileReader in a finally block, since we no longer need it. We check the booleans for each property to determine whether or not to return the property. To create the property as an IContentAttrValue, we look at the example of the total number of tokens property:

```
if(shouldCountTokens)
{
    StringTokenizer CLFTokenizer = new
        StringTokenizer(fileContent.toString());
    tokenCounter = CLFTokenizer.countTokens();
    Integer bigInt = new Integer(tokenCounter);
    IDfValue charValue = new DfValue(bigInt, DfValue.DF_INTEGER);
    IContentAttrValue charAttrValue = new
        ContentAttrValue(_num_tokens_name, charValue);
    attrValues[propArraySize -1] = charAttrValue;
}
```

Finally, we create the ICTSResult and return:

```
if(shouldCountTokens)
{
    StringTokenizer CLFTokenizer = new
        StringTokenizer(fileContent.toString());
    tokenCounter = CLFTokenizer.countTokens();
    Integer bigInt = new Integer(tokenCounter);
    IDfValue charValue = new DfValue(bigInt, DfValue.DF_INTEGER);
    IContentAttrValue charAttrValue = new
        ContentAttrValue(_num_tokens_name, charValue);
    attrValues[propArraySize -1] = charAttrValue;
}
```

Finally, we create the ICTSResult and return:

```
// Create the ICTSResult[]; only 1 ICTSResult will be in the
array.
results[0] = new CTSResult(attrValues);
```

5.6 Clean-up

The Text plug-in, like any other that is likely to be written, uses system resources that must be carefully managed to maintain the stability and performance of the system. One common example mentioned in the preceding section is the use of FileReader/Writer classes for opening text files, which must be properly closed after they are no longer needed. The theme of proper resource clean-up is very important, and the API manifests this activity in several ways.

An example is the executeProfile() method:

```
public ICTSResult[] executeProfile(URL[] mediaURLs,
                    ICTSCommandLineFileContent clfContent,
                    String[] inputFormatExtensions,
                    String outputFormatExtension,
                    boolean clearOnExit)
        throws CTSPluginException
```

The final parameter to this method, boolean clearOnExit, is meant to indicate to the plug-in whether the input files specified in the first parameter, URL[] mediaURLs, will be processed again. For example, consider the example of a plug-in that processes large video files. For example, quicktime, mpeg, and avi. These types of files can be very large. Typically, a library that is able to process these files will have to open the file and load it into memory to perform analysis or a transformation. Since they are so large, the amount of memory used, and the time consumed in loading the file is significant, and we would not want to perform such operations more often than necessary. If we are expecting two or more profiles to be executed in quick succession on the same video file, it would be ideal to load it only once, when the first request is made and have it ready in memory for the second invocation.[2]

If the clearOnExit parameter is set to true, this is an indication that the Transformation plug-in should keep the file loaded in memory after executeProfile() is executed. This can be accomplished in many ways, and the general topic of caching resources is beyond the scope of this document, but some aspects of caching must be mentioned. The main issue has to do with uniquely identifying the input file with some sort of key. The reason the clearOnExit parameter exists is because the same input file is passed to separate invocations of the executeProfile() method, and these separate invocations are not aware of each other.

It will make more sense to create a profile and a command line file that specifies the multiple operations desired. But in many cases, it would be better to take advantage of the powerful benefit provided by profiles: the ability to reuse “atomic” profiles that provide basic operations in sequences or chains of profile invocations. *OpenText Documentum Content Management - Transformation Services Administration Guide (EDCCT250400-AGD)* on OpenText My Support (https://support.opentext.com) provides more information. In this case, Transformation Services will be able to extract the file from the repository, and store it temporarily in a location specified by the URL[] mediaURLs parameter. The URLs in this parameter can be used as a unique key used by the Transformation plug-in to cache the file.

[2] Note that the ability to take advantage of this type of feature will rely heavily on the implementation of the supporting media library. The library must itself enable caching of inputs to make this plug-in caching feature useful. Consult your library documentation before deciding how to approach this issue.

```
// This block can be synchronized if the map used is an
// unsynchronized HashMap...
synchronized(this)
{
    for(int i = 0; i < mediaURLs.length; i++)
    {
        URL mediaURL = mediaURLs[i];
        File aFile = new File(mediaURLs.getFile()).
                getAbsolutePath();
        StringBuffer fileContent = (StringBuffer)
            _urlToStringBufferMap.get(mediaURL);
        if(fileContent == null)
        {
            // Create a FileReader for the input file.
            FileReader reader = new FileReader(aFile);

            // Call a fictitious method to get the file
                content from
            // the FileReader...
          fileContent = getFileContent(reader);
        }

        // we want to keep the content around
        if(!clearOnExit)
        {
            _urlToStringBufferMap.put(mediaURL, fileContent);
        }
    }
}
```

The idea behind this algorithm is to avoid opening the input text file if it has already been opened on an earlier invocation, and the earlier invocation requested that it should be cached by setting clearOnExit to true. After the content is read into a StringBuffer, you can do other operations on it without needing to open a file.

So far we have just gone through putting the file content into the cache, and checking if it is already there. We also need to ensure that these resources are cleaned up when they are no longer needed. That is what the completeInputResources(URL[] mediaURLs) method is for. If the Text Transformation plug-in was to implement this, the code can look as follows:

```
synchronized(this)
{
    // remove all the URLs from the cache
    for(int i = 0; i < mediaURLs.length; i++)
    {
     Object key = _urlToStringBufferMap.remove(mediaURLs[i]);
    }
}
```

Transformation Services invokes this method at the end of running a Task. The Transformation plug-in should not delete the input file from the file system. Transformation Services code is responsible for doing this. This method should only clean up other resources associated with the input.

There is one other method used for cleanup, called completeOutputResources(URL[] mediaURLs). It is also invoked by Transformation Services after it completes executing a Task. This method is intended to trigger a Transformation plug-in to clean-up resources associated with an output file. As with the completeInputResources(URL[] mediaURLs) method, the Transformation plug-in should not delete the actual output file from the file system. Transformation Services

typically will store the output file in the repository, and it is possible it will not have done so before calling this method. A common implementation of this method can delete a remote copy of the output file, if the supporting media library/application is located on another network machine, or clear the supporting library's cache associated with the output file.

5.7 Logging

Transformation Services uses Log4j2 to log messages at runtime. Log4j is a tool provided by the Apache Jakarta project that creates and maintains open-source solutions on the Java platform for distribution to the public at no charge. Refer to the Apache website for more information on log4j.

Log4j2 provides logging for Java applications. It has many useful features such as the ability to configure the logging behavior of your applications without modifying the source code, and it provides inheritance of loggers that enables arbitrarily fine granularity in control over what messages get logged. It is also the corporate standard logging tool for Documentum Java applications, and Foundation Java API provides a simplified interface for utilizing it in the DfLogger class. Foundation Java API documentation provides more information.

The sample code for the Text Transformation plug-in (Appendix A, TextPlugin.java on page 71) shows example usage of the Logger returned from the DfLogger class. The configuration for the Log4j2 functionality is found in the log4j2.properties file located in the CTS_HOME\config directory. Upon inspection of this file, you will notice that there are appenders set up for Transformation Services based on Java packages in such a way that the logging for the product is segregated into files based on Transformation plug-ins and core Transformation Services code. In this way, the logging for a particular Transformation plug-in can be directed to a file containing only that Transformation plug-in's relevant logging data. When developing your Transformation plug-in, it would be best to follow this pattern by defining your own appender and log files. You will be able to configure logging for your Transformation plug-in in isolation of the rest of the system.

5.8 Internationalization

Transformation Services supports I18N by supplying a mechanism for users to encode the strings that Transformation Services exposes to client applications in languages other than English. One of these client-exposed elements is profiles. As a Transformation plug-in developer, you can create your own profiles with labels that appear in client applications in whatever language you wish.

The other exposure of strings to clients comes with the properties generated by Transformation plug-ins. The way in which a Transformation plug-in returns properties to Transformation Services is entirely up to the plug-in developer.

It is recommended to save the properties in English and the localization for these properties will be processed with the locale mapping files on the repository when an UI client requests the property values. The locale mapping files are saved under `\System\Media Server\Resources\<LanguageCode_CountryCode>\mappings`.

5.9 Suggested approaches and helpful hints

The following are some suggested approaches and helpful hints when developing your own Transformation plug-ins.

5.9.1 XML configuration files

As mentioned in "Configuring the CTSPluginService.xml file" on page 46, adding a new Transformation plug-in involves specifying both the class that implements the ICTSPlugin interface for your Transformation plug-in, and the absolute path to the configuration file used by your Transformation plug-in. This file is passed to the loadConfiguration() method of your Transformation plug-in. This file is not parsed by Transformation Services, it is intended only for your Transformation plug-in, and as a result, can be written in any format the developer desires. However, by conforming to a particular DTD, you can use a convenient utility class provided by Transformation Services to parse the file: `com.documentum.cts.plugin.common.config.CTSPluginConfig`.

"Example: Testing the TEXT plug-in" on page 63 describes how you must configure your Java development environment to run Transformation Services, and includes the classpath settings required to access this class. The DTD for this XML is included in Appendix D, Command Line files on page 79, under the name `MP_Config.dtd`.

The following is the Text Transformation plug-in's implementation of the loadConfiguration() method:

```
public void loadConfiguration(String configFile)
        throws CTSPluginException
{
    _mpConfig = new CTSPluginConfig();
    _mpConfig.parse(configFile);
    _logger.info("Text CTS Plug-In, " + getVersion() +
" initialized.");
}
```

After the CTSPluginConfig is initialized, its public accessor methods can be called to retrieve the values stored in the XML configuration file. The source code for the Text Transformation plug-in illustrates this usage.

5.9.2 JNI and Java-COM bridges

Some media libraries are available only in native compiled implementations, and are not offered in Java. The Transformation Services SDK requires you to implement your Transformation plug-in in Java, but it is still important to be able to leverage non-Java technologies. Several of the Transformation plug-ins shipped with Transformation Services use native interfaces either through JNI or Java-COM bridges.

For more information on the use of JNI, refer the online documentation on Oracle website.

There are numerous reasons for being cautious in utilizing JNI. Performance can suffer during the translation to native code, it is difficult to deal with the interface, and error handling requires greater attention, as uncaught errors in native code can cause the JVM to crash, in the case of a Transformation plug-in using JNI crashing Transformation Services. With the presence of these issues it seems there is a little to recommend using JNI: aside from the fact that there can be no other way to interface with useful media libraries. When required, the interfaces you generate between your Java and native implementations should be as simple and concise as possible, and the frequency of calls should be minimized.

Another popular native implementation on the Windows platform for media libraries is COM. Accessing such libraries can be accomplished through creating a JNI interface to an implementation that the Transformation plug-in developer creates, which then would wrap the COM objects that supply the media processing. This can also be done more directly by using a Java-COM bridge (JACOB). The technique involved requires generation of Java classes the wrap the COM interfaces you need to invoke. The bridge implements the communication between the Java classes and the native implementation using JNI. There are several open source JACOB implementations available on SourceForge.net website.

This is a useful approach because the difficulties of using JNI directly can be avoided. Also, because the project is open source the Transformation plug-in developer has the option of debugging the implementation of JACOB, and fixing bugs if required.

5.9.3 Client libraries

There are many useful applications that a Documentum user might need to work with, and as a result, many different file formats that can be of importance. Since Transformation Services can only concentrate on a certain number of important file formats, and it does handle quite a few, the Transformation Services SDK has been created for developers to add the file format support they require.

While there are so many important file formats, there is a definite lack of server-ready processing engines to handle them. Transformation Services uses such server products to handle the requirements of image and video processing in a scalable, robust manner, through APIs designed to run in a server setting.

Sometimes however, it is necessary to provide format support where there are no proper server products. It is necessary in these cases to interface with client applications, and run them in the server context of Transformation Services. This type of endeavor has it own unique requirements that the Transformation plug-in developer must be aware of.

5.9.3.1 Error handling

When a client application generates an error condition, it is difficult or impossible for the Transformation plug-in utilizing it to notice the error. Often the client application will simply enter a non-responsive state, or a dialog box expecting user intervention will appear on the server. There will be no exceptions raised, nor errors reported since there is no programmatic interface to return them. An appropriate mechanism in this case would be to implement a process monitor in a separate thread that can track the client application, and terminate it if conditions suggest it will not respond. It is also possible to implement a utility that monitors the operating system for dialog boxes with known attributes appearing, For example, anticipated error messages and title bars. When such dialogs appear, it is possible to dismiss or otherwise acknowledge the dialog and recover from the error.

5.9.3.2 Service and desktop interaction

Transformation Services runs as a Windows Service, to minimize the amount of user intervention required to make it active. If the server is rebooted, there is no need for a user to login to the machine to start Transformation Services, which would not only be a security concern, but an unnecessary hassle. A server product should not require login, but it is possible that client applications cannot operate properly without login or desktop operation. A Windows service can be configured to allow desktop interaction, but this can be insufficient without login for some client applications. While this scenario must be avoided, if it cannot, your Transformation plug-in deployment need to modify the Transformation Services settings and require user login.

5.9.3.3 Concurrency

Transformation Services, like any good server product, performs concurrent operations, in this case through a configurable thread pool that executes tasks and delegates jobs to Transformation plug-ins. This decreases the wait time for jobs, and enhances the user's experience. However, Transformation plug-ins are largely dependent on the underlying media processing technology's ability to provide concurrent processing to accomplish this. It is unlikely that a Transformation plug-in using a client application can process concurrent operations, since the client application was likely not designed for this sort of usage. Attempting to do so can generate errors that are difficult to assign to a particular cause, since concurrency problems typically generate unpredictable side effects. To avoid these problems, a client application Transformation plug-in can synchronize access to the application, and ensure that the application is properly terminated at the end of each invocation.

5.9.3.4 JVM crash

As mentioned in "JNI and Java-COM bridges" on page 59, care must be taken in using native implemented libraries where error handling is more difficult. This can also be the case with client applications that are exposed through a DLL. Linking your Transformation plug-in to a system library provided by a client application, through JNI for example, makes Transformation Services very sensitive to the error handling implemented in that Transformation plug-in. Uncaught errors in the system library can cause it to crash, and since Transformation Services runs in a JVM linked to this library, it will also crash. This crash can initially be difficult to detect, since the Windows service cannot be stopped, although log files will show no activity, and no Transformation Services tasks will execute.

Chapter 6

Example: Testing the TEXT plug-in

The best way to test a Transformation plug-in is by installing it on a Transformation Services system as described in Creating and Configuring Transformation plug-ins. To debug your Transformation plug-in, Transformation Services should be installed on the same system where you do your development, and have your Java IDE installed.

Different IDE's will result in different development configuration environments, so the guidelines for running your Transformation plug-in in an IDE will be described in generic fashion.

6.1 Classpath

To run the Transformation Services code, the compiled classes that come with Transformation Services must be in your environment settings. If you are running in an IDE, you will most likely be able to add the .jar files, or directories containing .jar files, to your project's settings. If run from a command line debugger, the classpath will have to be set for in the context of the command prompt, or more broadly from the system classpath. Transformation Services does not modify the system or user classpath, in an attempt to remain as isolated as possible from other Java applications. The classpath for your installation of Transformation Services will be found in a .bat file in a subdirectory of your installation. For example, `C:\Documentum\CTS\server_install\CTSService\startCTS.bat`. This includes .jar files belonging to Foundation Java API which are deployed to the `Documentum\Shared\` folder.

6.2 Parameters

To run Transformation Services from your IDE, the location of the configuration directory must be passed as a parameter to the JVM. The parameter you should use can be found in the .bat file deployed by Transformation Services, mentioned in the preceding section.

6.3 Working directory

Transformation Services deploys two DTD files to the Transformation Services instance home directory. These files must be copied to this location in order for Foundation Java API to parse XML files needed by Transformation Services. To successfully run Transformation Services you must copy the files `ProfileSubmission.dtd` and `ProfileSchema.dtd` to your project's working directory. These files can be found in the "lib" folder of your Transformation Services installation. For example, `C:\Documentum\CTS\lib\`.

Chapter 7

Example: Installing the TEXT plug-in

The steps involved in installing a Transformation Plug-In will vary somewhat depending on the technologies being used. The Text Transformation plug-in described in this document has installation requirements that are common to most Transformation plug-ins. The steps that we will describe will refer to the Text Transformation plug-in as an example for these essential installation tasks, but will also include some hints to assist in installing other kinds of Transformation plug-ins. Also, if you use a third-party product for your Transformation plug-in, its installation will have to be performed in such a way that your Transformation plug-in can utilize it.

- Copy any .dll windows libraries and jar files associated with the new plug-in to the lib folder in the Transformation Services installation directory. The `TextPlugin.class` file can be packed into a .jar file; there are no .dll's required for this plug-in. The .jar file need not be copied to this particular directory, but it would be a good idea to put it in the same location as the rest of the Transformation Services code.
- Create a sub folder of the config directory in the Transformation Services installation directory for your new plug-in. We would create a directory like `C:\Documentum\CTS\config\text\`.
- Copy the required configuration files to the `config` folder you just created. The Text Transformation plug-in's configuration file, `text.xml` (included in Appendix D, Command Line files on page 79) would be copied to the text folder. Ensure that the cache directory is correct for the current machine.
- Add an entry to the `config\CTSPuginService.xml` file in the Transformation Services installation directory so that Transformation Services can load the new Transformation plug-in (ensure that the `config` file location is the one from the preceding step). In the case of the Text Transformation plug-in, the new entry would be:

```
<CTSPlugin DELEGATE_CLASS="com.documentum.cts.plugin.text.
TextPlugin
" CONFIGFILE="C:\DOCUME~1\CTS\config\text\text.xml"/>
```

- Edit the `log4j.properties` file in the *<CTS_HOME>*`\config\` directory to add an appender for the Transformation plug-in. The Text Transformation plug-in would require the following changes:

  1. Add the following logger:

```
logger.RTextAppender.name = com.documentum.cts.plugin.text
logger.RTextAppender.level = INFO
logger.RTextAppender.additivity = false
logger.RTextAppender.appenderRef.FTextAppender.ref = TextAppender
```

  2. Add the following appender information:

```
appender.FTextAppender.type=RollingFile
appender.FTextAppender.name= TextAppender
appender.FTextAppender.fileName=C\:\\DOCUME~1\\CTS\\logs\\Text_log.txt
appender.FTextAppender.filePattern=C\:\\DOCUME~1\\CTS\\logs\\Text_log.%d{yyyy-
MM-dd}-%i
appender.FTextAppender.layout.type=PatternLayout
appender.FTextAppender.layout.pattern=%d{ABSOLUTE} %5p [%t] %c - %m%n
appender.FTextAppender.policies.type=Policies
appender.FTextAppender.policies.time.type=TimeBasedTriggeringPolicy
appender.FTextAppender.policies.time.interval=1
appender.FTextAppender.policies.time.modulate=true
appender.FTextAppender.policies.size.type=SizeBasedTriggeringPolicy
appender.FTextAppender.policies.size.size=100MB
appender.FTextAppender.strategy.type=DefaultRolloverStrategy
appender.FTextAppender.strategy.max=5
```

- The startup parameters for the Transformation Services Service must be modified so that the classpath includes the new Transformation plug-in. This can be done for every Transformation Services instance that the Transformation plug-in should be added to. Make the following changes to the registry key:

  ```
  HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\
  DocumentumCTS\Parameters
  ```

  Add the following:

  ```
  ;C:\DOCUME~1\CTS\lib\TextPlugin.jar
  ```

  before ' -mainclass=...' in the Arguments entry. Be sure to include the semi-colon. This value is based on having copied the `TextPlugin.class` (inside a .jar file) to the Transformation Services lib folder. If it was copied elsewhere, adjust the value.

- Add the command line files (in Appendix C, Transformation profiles on page 77) to the repository folder `\System\Media Server\Command Line Files` folder with a repository client application. When importing these files ensure that the .xml is part of the object name and that the document type is `dm_media_profile`.

- Add the new profiles (in Appendix A, TextPlugin.java on page 71) to the repository in the `\System\Media Server\Profiles\`. Ensure that the `ProfileSchema.dtd` file is in the same directory as the system profiles being imported. When importing this profile ensure that the .xml is not part of the object name and that the document type is `dm_sysobject`.

- Run DQL statements against the repository to either create new richmedia_enabled dm_formats if they do not already exist, or update the richmedia_enabled flags of existing dm_formats that the Transformation plug-in now supports. This enables newly imported documents of these formats to generate queue items for processing by Transformation Services, and tags the format as being processable by Transformation Services in general. The Text Transformation plug-in will run the following updates:

  ```
  update dm_format object set richmedia_enabled = 1 where
  name = 'text'
  update dm_format object set richmedia_enabled = 1 where
  name = 'mactext'
  update dm_format object set richmedia_enabled = 1 where
  name = 'crtext'
  ```

When dm_format's are changed or created, the repository in which these changes are made must be restarted because the dm_format's are cached, and client applications won't detect the updates.

- Stop the Transformation Services Windows Service.
- Start the Transformation Services Windows Service. This allows Transformation Services to update its list of profiles with the new and modified profiles, and also will start the new Transformation plug-in up for the first time.

Chapter 8

Transformation Services Javadocs

The following sections describe the Transformation Services Javadocs and how to use them to modify and create new plug-ins.

8.1 Viewing the Javadocs

Sun Microsystems Javadoc tool generates API documentation in HTML format from doc comments in source code. It compiles a list of packages and their properties into an HTML help format. Appropriate comments describing each class, field, and method have been added to the Transformation Services Client Javadocs as well.

The Transformation Services Javadocs package includes detailed sample code for the Transformation Services SDK, Local Service API, and Remote Service API.

To view the Javadocs:

1. Extract the contents of the `CTS_<release-version>_javadocs.zip` file.
2. Double-click `index.html` in the extracted folder to open the Transformation Services Javadocs.
3. Scroll through the package's indices to find the appropriate class, interface, or exception. For example, if you want a method to access the ICTSResult, first select the Transformation Services SDK link in the left pane, then click the ICTSResult interface from the Class Index. For each class or interface you can view two pages:

   Class: Contains the class/interface description, summary tables, and detailed member descriptions.

   Use: Describes what packages, classes, methods, constructors, and fields use any part of the given class.

Appendix A. TextPlugin.java

The following is the full `TextPlugin.java` that is explained in Chapter 7:

```
package com.documentum.cts.impl.services.task.text;

import java.util.Properties;

import com.documentum.cts.services.task.ICTSCommand;
import com.documentum.cts.exceptions.internal.CTSServicesBaseException;
import com.documentum.cts.impl.services.task.CTSTask;
import com.documentum.cts.plugin.common.ICTSResult;
import com.documentum.fc.common.DfException;
import com.documentum.fc.client.*;

/**
 * ICTSTask to process text
 */
public class TextProcessor extends CTSTask {
    /**
     * Initializes the variables needed by this task
     *
     * @param aICTSCommand the ICTSCommand in
     *                     which the input information is stored
     * @throws com.documentum.cts.exceptions.internal.CTSServicesBaseException
       if a media services error occurs
     * @throws com.documentum.fc.common.DfException
       if a DFC error occurs
     */
    protected void initializeFromCommand(ICTSCommand aICTSCommand)
            throws CTSServicesBaseException, DfException {
        super.initializeFromCommand(aICTSCommand);
/**
 * can do all the additional initialization here
 */
    } /*
* This method does all the post processing
* of the transformation result
* @see com.documentum.cts.impl.services.task.CTSTask#execTx()
*/

    protected void execTx() throws Exception { // Now that the plugin has completed,
we need to add the transformed
// object(s) back into the repository or whetever else you want
// to do with the result..
        if ((_resultsToUse != null) &&
                (_resultsToUse.length != 0)) {
            //or call/override this method to add renditions
            createRenditionOnSourceDocument(_resultsToUse);

//you can either call/override this method to add the media properties
            addProperties(_resultsToUse,
                    getOriginalTargetFormat(),
                    getTargetPageModifier(),
                    getOriginalDocument());

//or you can perform custom post processing here
            processCTSResult(_resultsToUse);
        } else {
            getLogger().error("The CTS Plugin returned no results to process");
            throw new Exception("The CTS Plugin returned no results to process");
        }
    } //or perform any custom processing on this result

    private void processCTSResult(ICTSResult[] _resultsToUse) {
    }
}
```

Appendix B. Local Service API sample code

The following is sample code on how to create a asynchronous transformation request. Refer to the sample code bundled with the Transformation Services Javadocs to get a better picture on the usage.

```
package com.documentum.test.services.cts.transform;
import com.documentum.services.cts.df.transform.ICTSRequest;
import com.documentum.services.cts.df.transform.ICTSAddJobService;
import com.documentum.services.dam.df.transform.IMediaProfile;
import com.documentum.services.dam.df.transform.IProfileParameter;
import com.documentum.fc.common.*;
import com.documentum.fc.client.*;
import com.documentum.com.IDfClientX;
import com.documentum.com.DfClientX;
import java.net.URL;
import java.io.*;
public class AddJobServiceTest
{
    final static String PROFILE = "flip";
    final static String TARGET_FORMAT = "gif";
    final static String SOURCE_FORMAT = "jpeg";
    IDfSessionManager m_mgrSession;
    public AddJobServiceTest(String strDocbase,
                             String strUser,
                             String strDomain,
                             String strPassword) throws DfException
    {
        IDfLoginInfo loginInfo = new DfLoginInfo();
        loginInfo.setUser(strUser);
        loginInfo.setPassword(strPassword);
        if (strDomain != null)
            loginInfo.setDomain(strDomain);
        IDfClientX clientx = new DfClientX();
        IDfClient client = clientx.getLocalClient();
        m_mgrSession = client.newSessionManager();
        m_mgrSession.setIdentity(strDocbase, loginInfo);
    }
    public String addJob(String theSourceObjectId,
                         String theRepoName,
                         String strRelatedId) throws DfException
    {
    // create addJob service
        String theJobId = null;
        IDfClient client = null;
        IDfSession session = null;
        try
        {
            IDfSessionManager sessManager = null;
            ICTSAddJobService theCTSService = null;
            try
            {
                IDfClientX clientX = new DfClientX();
                client = clientX.getLocalClient();
                session = getSession(theRepoName);
                sessManager = session.getSessionManager();
                theCTSService = (ICTSAddJobService)client.
                        newService(ICTSAddJobService.class.getName(), sessManager);
            }
            catch(DfException dFex)
            {
                dFex.printStackTrace();
            }
    // create ctsrequest
            if( theCTSService != null )
            {
                ICTSRequest theRequest =
                        createCTSRequest(session, theCTSService,
```

```
                                        theRepoName, theSourceObjectId,
                                        strRelatedId );
                if( theRequest != null )
                {
                    theJobId = theCTSService.addJob(theRequest);
                }
            }
            return theJobId;
        }
        finally
        {
            releaseSession(session);
        }
    }

    private IDfSession getSession(String strDocbase) throws DfException
    {
        return m_mgrSession.getSession(strDocbase);
    }

    private void releaseSession(IDfSession session)
    {
        if(session != null )
        {
            m_mgrSession.release(session);
        }
    }

    private ICTSRequest createCTSRequest(IDfSession theSession,
                                         ICTSAddJobService theCTSService,
                                         String theRepoName,
                                         String theSourceObjectId,
                                         String strRelatedId) throws DfException
    {
        ICTSRequest theRequest = theCTSService.getNewCTSRequest(theRepoName);
        IMediaProfile theProfile = getMediaProfile(theSession,PROFILE);
        String theProfileId = null;
        if (theProfile != null)
        {
            theProfileId = theProfile.getObjectId().
                    toString();
            int userPermit = theProfile.getPermit();
            if (userPermit >= IDfACL.DF_PERMIT_READ)
            {
                System.out.println("User Has Read Permission On Current Profile");
            }
            else
            {
                System.out.println("User Does Not Have Read Permission On Current
Profile");
            }
            System.out.println("Current Media Profiles: " + theProfile.getObjectName());
        }
        if (theProfileId != null)
        {
    //String placeholderDoc = null;
            if( theSourceObjectId != null )
            {
                theRequest.setSourceObjectId(theSourceObjectId);
            }
            if( strRelatedId != null )
            {
                theRequest.setRelatedObjectId(strRelatedId);
            }
            theRequest.setMediaProfileId(theProfileId);
            theRequest.setSourceFormat(SOURCE_FORMAT);
            theRequest.setTargetFormat(TARGET_FORMAT);
            theRequest.setSourcePageModifier("");
            theRequest.setTargetPageModifier("");
            theRequest.setLocale(java.util.Locale.getDefault());
            theRequest.setPriority(1);
```

```
            theRequest.setSourcePage(0);
            theRequest.setTargetPage(0);
            IProfileParameter[] profileParameters =
                    theProfile.getParameters();
    // here you can pass in all the parameter values that are required for the
    // transformation of this specific profile
            String value_doc_token_direction = "vertical";
            for (int i=0; i < profileParameters.length; i++)
            {
                String currentParamName = profileParameters[i].
                        getParameterName();
                if (currentParamName.equals("doc_token_direction"))
                {
                    profileParameters[i].setParameterValue(value_doc_token_direction);
                }
            }
            theRequest.setParameters(profileParameters);
            theRequest.setMediaProfileName(theProfile.getString("object_name"));
            theRequest.setMediaProfileName(theProfile.getString("title"));
            String notify_result = theProfile.getNotifyResult();
            if(notify_result != null)
            {
                theRequest.setNotifyResult(notify_result);
                System.out.println("Notify Result: " +
                        notify_result);
            }
            theRequest.setDeleteOnCompletion(false);
            theRequest.setTransformationType("asynchronous");
            theRequest.setZipOutput(true);
            theRequest.setStoreResultInDocbase(true);
            theRequest.setTargetFolder(null);
        }
        return theRequest;
    }

    /**
     * Returns a list of rendition profiles
     * @param session - the session
     * @param strProfileName - String document ID
     * @return IMediaProfile[] of media profile objects
     * @throws DfException the exception
     */
    private IMediaProfile getMediaProfile(IDfSession session,
                                          String strProfileName) throws DfException
    {
        IMediaProfile mediaProfile = null;
        if (strProfileName == null || strProfileName.length() == 0)
        {
            throw new IllegalArgumentException("Invalid Profile Name");
        }
        try
        {
            mediaProfile = (IMediaProfile)session.
                    getObjectByQualification("dm_media_profile where object_name= '"
                            + strProfileName + "'");
        }
        catch(DfServiceException dfEx)
        {
            dfEx.printStackTrace();
        }
        return mediaProfile;
    }
}
```

Appendix C. Transformation profiles

The following are the sample of transformation profiles that can be used to start your profile creation:

C.1 to_WindowsText.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!-- edited by Documentum Canada  -->
<!DOCTYPE Profile SYSTEM "ProfileSchema.dtd">
<Profile name="to_WindowsText" label="Windows Text conversion"
description="Convert
to Windows Text" taskImpl="com.documentum.cts.impl.services.
task.text.TextProcessor">
   <Formats>
   <Format source="mactext" target="crtext"/>
   <Format source="text" target="crtext"/>
   </Formats>
<Filters>
 <Filter name="CTSProduct" value="MTS"/>
 <Filter name="Visibility" value="Public"/>
 </Filters>
 <Transcodings>
</Transcodings>
  <CommandFilePath mptype="TEXT">
</System/Media Server/Command Line Files/to_WindowsText.xml>
</CommandFilePath>
</Profile>
```

C.2 to_UnixText.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Profile SYSTEM "ProfileSchema.dtd">
<Profile name="to_UnixText" label="UNIX Text conversion"
description=
"Convert to UNIX Text">
    <Formats>
        <Format source="mactext" target="text"/>
        <Format source="crtext" target="text"/>
    </Formats>
 <Filters>
      <Filter name="CTSProduct" value="MTS"/>
      <Filter name="Visibility" value="Public"/>
   </Filters>
<Transcodings>
    </Transcodings>
    <CommandFilePath mptype="TEXT">
        /System/Media Server/Command Line Files/to_UnixText.xml>
    </CommandFilePath>
</Profile>
```

C.3 to_MacText.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Profile SYSTEM "ProfileSchema.dtd">
<Profile name="to_MacText" label="Mac Text conversion"
description=
"Convert to Mac Text">
    <Formats>
        <Format source="crtext" target="mactext"/>
        <Format source="text" target="mactext"/>
    </Formats>
     <Filters>
      <Filter name="CTSProduct" value="MTS"/>
      <Filter name="Visibility" value="Public"/>
   </Filters>
    <Transcodings>
    </Transcodings>
    <CommandFilePath mptype="TEXT">
        /System/Media Server/Command Line Files/to_MacText.xml
    </CommandFilePath>
</Profile>
```

Appendix D. Command Line files

The following are some sample command line files that you can use to start your own command line files:

D.1 text_extract_properties_clf.xml

```
<?xml version="1.0"?>
<!DOCTYPE TEXT_MP_EXTRACT_PROPERTIES [
<!ELEMENT TEXT_MP_EXTRACT_PROPERTIES (PROP*)>
<!ELEMENT PROP (#PCDATA)>
<!ATTLIST PROP
        name CDATA #REQUIRED
        token CDATA #REQUIRED
]>
<TEXT_MP_EXTRACT_PROPERTIES>
    <PROP name="Number of Tokens" token="doc_token_
extract_tokens" >YES</PROP>
    <PROP name="Number of Characters" token="doc_
token_extract_chars" >YES</PROP>
</TEXT_MP_EXTRACT_PROPERTIES>
```

D.2 to_WindowsText.xml

```
<?xml version="1.0"?>
<!DOCTYPE TEXT_MP_TO_WINDOWS [
<!ELEMENT TEXT_MP_TO_WINDOWS (#PCDATA)>
]>
<TEXT_MP_TO_WINDOWS>
<!-- There is nothing interesting to put here.
We only need to know the
DTD to identify the operation. -->
</TEXT_MP_TO_WINDOWS>
```

D.3 to_UnixText.xml

```
<?xml version="1.0"?>
<!DOCTYPE TEXT_MP_TO_UNIX [
<!ELEMENT TEXT_MP_TO_UNIX (#PCDATA)>
]>
<TEXT_MP_TO_UNIX>
<!-- There is nothing interesting to put here.
We only need to know the
DTD to identify the operation. -->
</TEXT_MP_TO_UNIX>
```

D.4 to_MacText.xml

```
<?xml version="1.0"?>
<!DOCTYPE TEXT_MP_TO_MAC [
<!ELEMENT TEXT_MP_TO_MAC (#PCDATA)>
]>
<TEXT_MP_TO_MAC>
<!-- There is nothing interesting to put here.
We only need to know the
DTD to identify the operation. -->
</TEXT_MP_TO_MAC>
```

Appendix E. Configuration files

The following `text.xml` file is a sample configuration file. The `MP_config.dtd` that follows must be used when creating or modifying your own configuration files.

E.1 text.xml

```
<?xml version="1.0"?>
<!DOCTYPE MP_CONFIG SYSTEM "../MP_Config.dtd">
<MP_CONFIG>
<CACHE>C:\Documentum\CTS\cache</CACHE>
<VERSION>Media Services - Text Media Plug-in 1.0</VERSION>
<SERVERNAME>TEXT</SERVERNAME>
<EXTRACT_PROPERTIES>
<EXTRACTABLE>
<DEFAULT>
<PROPERTY name="Number of Tokens"
type="string"/>
<PROPERTY name="Number of Characters"
type="string"/>
</DEFAULT>
</EXTRACTABLE>
<INCLUDE>
<ALL/>
</INCLUDE>
<RESOURCE mpname="Number of Tokens" name
="Number of Tokens"/>
<RESOURCE mpname="Number of Characters" name
="Number of Characters"/>
</EXTRACT_PROPERTIES>
<SUPPORTED_FORMATS>
<SOURCE format="txt">
<TARGET format="txt"></TARGET>
</SOURCE>
</SUPPORTED_FORMATS>
</MP_CONFIG>
```

E.2 MP_Config.dtd

```
<!ELEMENT MP_CONFIG (CACHE, VERSION, SERVERNAME,
EXTRACT_PROPERTIES, SUPPORTED_
FORMATS, FORMAT_MAPPER*) >

<!-- This is a path where the MP will place its output
and intermediate files. -->
<!ELEMENT CACHE (#PCDATA)>

<!ELEMENT VERSION (#PCDATA)>

<!-- This is the name of the MP.  Corresponds to
the return value of getName()
on IMediaPlugin interface.-->
<!ELEMENT SERVERNAME (#PCDATA)>

<!ELEMENT EXTRACT_PROPERTIES (EXTRACTABLE,
INCLUDE, RESOURCE*)>

<!--  SOURCE elements should only have OUTPUT
elements if a child of TRANSFORM
-->
<!ELEMENT SOURCE (TARGET*)>
<!ATTLIST SOURCE
    format CDATA #REQUIRED
```

```
>

<!ELEMENT TARGET (#PCDATA)>
<!ATTLIST TARGET
    format CDATA #REQUIRED
>

<!--  DEFAULT is used when it is not applicable
to supply a list of EXTRACTABLE
properties e.g. when all properties generated
are dynamic and not known at
configuration time.  -->
<!ELEMENT EXTRACTABLE (FORMAT*, DEFAULT)>

<!ELEMENT FORMAT (PROPERTY*)>
<!ATTLIST FORMAT
    value CDATA #REQUIRED
>

<!ELEMENT DEFAULT (PROPERTY*)>

<!ELEMENT PROPERTY (VALUE*)>
<!--  The mpname attribute is used for
internal purposes and should not be
changed.    'name' is the internationalized
value for the property name, and
will be returned by the Media Plug-in    in all
Property objects returned by the
MP SDK.  -->
<!ATTLIST PROPERTY
    name CDATA #REQUIRED
    type  ( string | integer | double | date
    | boolean ) #REQUIRED
>

<!--  The VALUE element exists only for the
option of internationalizing
common values.    If an MP returns a PROPERTY with a
VALUE that has an mpvalue
in this list, the value    returned in the Property
object will take on the
"value" attribute's setting.  -->
<!ELEMENT VALUE (#PCDATA)>

<!--  The INCLUDE element designates which properties
should be extracted
by the MP.    The list will refer the PROPERTY elements
included under the
EXTRACTABLE element, or ALL or NONE elements can be
used to denote inclusion
of all or none of the available    properties.  The EXCLUDE
element allows the
user to EXCLUDE only certain specific properties.  -->
<!ELEMENT INCLUDE (FORMAT* | ALL | NONE | EXCLUDE)>
<!ELEMENT ALL EMPTY>
<!ELEMENT NONE EMPTY>

<!--  The EXCLUDE element exists only for convenience,
in order to exclude
a short list of elements    instead of having to include
a very long list of
elements  -->
<!ELEMENT EXCLUDE (FORMAT*)>

<!ELEMENT SUPPORTED_FORMATS (SOURCE*)>

<!ELEMENT FORMAT_MAPPER (MAP*)>

<!-- The RESOURCE element defines all string substitutions
that should be
performed on PROPERTY name's and value's.    If a given
```

```
property name or value
occurs as an "mpstring" in the RESOURCE list, it will
be returned from the
MP    interface substituted by the "string" value for
the pair.  -->
<!ELEMENT RESOURCE EMPTY>
<!ATTLIST RESOURCE
    mpname CDATA #REQUIRED
    name CDATA #REQUIRED
>
<!ELEMENT MAP EMPTY>
<!ATTLIST MAP
    PluginFormat CDATA #REQUIRED
    DocumentumFormat CDATA #REQUIRED
>
```

Appendix F. TextProcessor.java

The following is a full sample Task class which shows the pre-post processing of a transformation request:

```
package com.documentum.cts.impl.services.task.text;

import java.util.Properties;

import com.documentum.cts.services.task.ICTSCommand;
import com.documentum.cts.exceptions.internal.CTSServicesBaseException;
import com.documentum.cts.impl.services.task.CTSTask;
import com.documentum.cts.plugin.common.ICTSResult;
import com.documentum.fc.common.DfException;
import com.documentum.fc.client.*;

/**
 * ICTSTask to process text
 */
public class TextProcessor extends CTSTask {
    /**
     * Initializes the variables needed by this task
     *
     * @param aICTSCommand the ICTSCommand in
     *                     which the input information is stored
     * @throws com.documentum.cts.exceptions.internal.CTSServicesBaseException
       if a media services error occurs
     * @throws com.documentum.fc.common.DfException
       if a DFC error occurs
     */
    protected void initializeFromCommand(ICTSCommand aICTSCommand)
            throws CTSServicesBaseException, DfException {
        super.initializeFromCommand(aICTSCommand);
/**
 * can do all the additional initialization here
 */
    } /*
* This method does all the post processing
* of the transformation result
* @see com.documentum.cts.impl.services.task.CTSTask#execTx()
*/

    protected void execTx() throws Exception { // Now that the plugin has completed,
    we need to add the transformed
// object(s) back into the repository or whetever else you want
// to do with the result..
        if ((_resultsToUse != null) &&
                (_resultsToUse.length != 0)) {
            //or call/override this method to add renditions
            createRenditionOnSourceDocument(_resultsToUse);

//you can either call/override this method to add the media properties
            addProperties(_resultsToUse,
                    getOriginalTargetFormat(),
                    getTargetPageModifier(),
                    getOriginalDocument());

//or you can perform custom post processing here
            processCTSResult(_resultsToUse);
        } else {
            getLogger().error("The CTS Plugin returned no results to process");
            throw new Exception("The CTS Plugin returned no results to process");
        }
    } //or perform any custom processing on this result

    private void processCTSResult(ICTSResult[] _resultsToUse) {
    }
}
```