OpenText Document Management, eDOCS Edition

eDOCS DM API Samples Guide

This guide provides sample DM API code for various functions in eDOCS DM.

Last Updated: March 22, 2016

**OpenText Document Management, eDOCS Edition**

**eDOCS DM API Samples Guide**

Rev.: 2016-03-22

**This documentation has been created for software version 10.0 and later versions.**   
It is also valid for subsequent software versions as long as no new document version is shipped with the product or is published at [https://knowledge.opentext.com](https://knowledge.opentext.com#_blank).

**Open Text SA**

40 Avenue Monterey , Luxembourg, Luxembourg L-2163

Tel: 35 2 264566 1

**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: [http://support.opentext.com](http://support.opentext.com#_blank)   
For more information, visit [https://www.opentext.com](https://www.opentext.com#_blank)

**Copyright © 2016 Open Text SA and/or Open Text ULC (in Canada). All Rights Reserved.**

Trademarks owned by Open Text SA or Open Text ULC (in Canada). All rights reserved.

**Disclaimer**

No Warranties and Limitation of Liability

Every effort has been made to ensure the accuracy of the features and techniques presented in this publication. However, Open Text Corporation and its affiliates accept no responsibility and offer no warranty whether expressed or implied, for the accuracy of this publication.

Contents

[1 Chapter 1 - An Overview of the DM Architecture and the DM API 6](#_Toc441149873)

[1.1 Data Tier 6](#_Toc441149874)

[1.2 Transactional Tier 6](#_Toc441149875)

[1.3 Client Tier 7](#_Toc441149876)

[1.4 Forms in the DM API 7](#_Toc441149877)

[1.5 The Role of the DM API in the DM Architecture 8](#_Toc441149878)

[1.6 An Overview of the DM API 8](#_Toc441149879)

[1.7 DM Security Token (DST) 9](#_Toc441149880)

[1.8 Using DM API Objects 9](#_Toc441149881)

[1.8.1 Tokens in the DM API 11](#_Toc441149882)

[2 Chapter 2 – Authentication with the DM API 12](#_Toc441149883)

[2.1 The PCDLogin Object 17](#_Toc441149884)

[2.1.1 Characterization Methods 17](#_Toc441149885)

[2.1.2 Transaction Methods 17](#_Toc441149886)

[2.1.3 Retrieval Methods 18](#_Toc441149887)

[3 Chapter 3 - Adding a Document to DM 19](#_Toc441149888)

[3.1 PCDDocObject Role in the Creation of Documents 19](#_Toc441149889)

[3.1.1 Creating a New Document 19](#_Toc441149890)

[3.1.2 PCDDocObject Characterization Methods 24](#_Toc441149891)

[3.1.3 PCDDocObject Transaction Methods 24](#_Toc441149892)

[3.1.4 PCDDocObject Retrieval Methods 24](#_Toc441149893)

[3.1.5 The PCDPutDoc Object 25](#_Toc441149894)

[3.1.6 PCDPutDoc Characterization Methods 25](#_Toc441149895)

[3.1.7 PCDPutDoc Transaction Methods 25](#_Toc441149896)

[3.1.8 PCDPutDoc Retrieval Methods 26](#_Toc441149897)

[3.2 The PCDPutStream Object 26](#_Toc441149898)

[3.2.1 PCDPutStream Transaction Method 26](#_Toc441149899)

[3.2.2 PCDPutStream Retrieval Methods 26](#_Toc441149900)

[4 Chapter 4 - Retrieving Documents and Metadata from the DM Repository 27](#_Toc441149901)

[4.1 Retrieving Library Documents 27](#_Toc441149902)

[4.2 Returning Version Information 29](#_Toc441149903)

[4.3 Returning Document Metadata 30](#_Toc441149904)

[4.4 The PCDGetDoc Object 31](#_Toc441149905)

[4.4.1 PCDGetDoc Characterization Methods 31](#_Toc441149906)

[4.4.2 PCDGetDoc Transaction Methods 32](#_Toc441149907)

[4.4.3 PCDGetDoc Retrieval and Other Methods 32](#_Toc441149908)

[4.5 PCDGetStream Object 33](#_Toc441149909)

[4.5.1 PCDGetStream Transaction Methods 33](#_Toc441149910)

[4.5.2 PCDGetStream Retrieval and Other Methods 33](#_Toc441149911)

[5 Chapter 5 - Searching for Documents 34](#_Toc441149912)

[5.1 Metadata Searches 34](#_Toc441149913)

[5.2 Full-text Index Searches 34](#_Toc441149914)

[5.3 Hybrid Searches 35](#_Toc441149915)

[5.4 Creating a Search with the DM API 35](#_Toc441149916)

[5.5 The PCDSearch Object 37](#_Toc441149917)

[5.5.1 PCDSearch Characterization Methods 37](#_Toc441149918)

[5.5.2 PCDSearch Transaction Method 39](#_Toc441149919)

[5.5.3 Optional PCDSearch Characterization Methods 39](#_Toc441149920)

[5.5.4 PCDSearch Object Retrieval Methods 39](#_Toc441149921)

[6 Chapter 6 - Retrieving and Modifying Document Metadata 41](#_Toc441149922)

[6.1 Returning Document Metadata 41](#_Toc441149923)

[6.2 Modifying Document Metadata 41](#_Toc441149924)

[6.3 Modifying the Status of a Profile 42](#_Toc441149925)

[6.4 Checking Out Documents 43](#_Toc441149926)

[6.5 Modifying the Read-Only Status 44](#_Toc441149927)

[6.6 Unlocking a Profile 45](#_Toc441149928)

[7 Chapter 7 - Managing Security in eDOCS DM 46](#_Toc441149929)

[7.1 Access Rights and Tokens 46](#_Toc441149930)

[7.2 Deny Rights 47](#_Toc441149931)

[7.3 Working with Trustee Rights 47](#_Toc441149932)

[7.4 Retrieving a List of Trustees and Rights 49](#_Toc441149933)

[7.5 Adding and Removing Trustees to a Profile Access Control List 50](#_Toc441149934)

[7.6 The PCDDocObject Methods for Security Management 53](#_Toc441149935)

[7.6.1 Transaction Methods 53](#_Toc441149936)

[7.6.2 Retrieval Methods 53](#_Toc441149937)

[7.7 PCDTrusteeList Object 53](#_Toc441149938)

[7.7.1 Characterization Methods 54](#_Toc441149939)

[7.7.2 Retrieval Methods 54](#_Toc441149940)

[7.7.3 Helper Methods 55](#_Toc441149941)

[7.8 iTrusteeRights Combinations 55](#_Toc441149942)

[8 Chapter 8 - Document Versions and Attachments 58](#_Toc441149943)

[8.1 An Overview of Versions and Attachments 58](#_Toc441149944)

[8.2 Creating a Version, Sub-version, or Attachment 58](#_Toc441149945)

[8.3 Creating a Sub-version of a Document 61](#_Toc441149946)

[8.4 Adding an Attachment to a Document 63](#_Toc441149947)

[8.5 Retrieving a Version, Sub-version, or Attachment 65](#_Toc441149948)

[8.6 Deleting a Version, Sub-version, or Attachment 67](#_Toc441149949)

[8.7 Determining the Version ID of a Document, Version or Attachment 68](#_Toc441149950)

[8.8 Tokens Used when Working with Versions, Sub-Versions, and Attachments 69](#_Toc441149951)

[9 Chapter 9 - Relating and Unrelating Documents 70](#_Toc441149952)

[9.1 Searching for Related Documents 73](#_Toc441149953)

[10 Chapter 10 - Working with Folders 75](#_Toc441149954)

[10.1 Creating a Folder 75](#_Toc441149955)

[10.2 Adding an Item to a Folder 76](#_Toc441149956)

[10.3 Retrieving Items from a Folder 77](#_Toc441149957)

[10.4 Updating an Item in a Folder 79](#_Toc441149958)

[10.5 Changing an Item’s Location in the Folder 80](#_Toc441149959)

[10.6 Moving an Item after Another Item in a Folder 81](#_Toc441149960)

[10.7 Batch Reordering of Items 81](#_Toc441149961)

[10.8 Copying Contents from One Folder to Another 83](#_Toc441149962)

[11 Chapter 11 - Working with Profile Defaults and ACL Defaults 85](#_Toc441149963)

[11.1 Creating Profile Defaults 85](#_Toc441149964)

[11.2 Creating ACL Defaults 87](#_Toc441149965)

[11.3 Retrieving Profile Defaults 88](#_Toc441149966)

[12 Chapter 12 - Removing Items from DM 91](#_Toc441149967)

[12.1 Deleting a Document or Folder 91](#_Toc441149968)

[12.2 Deleting a Version, Subversion, or Attachment 92](#_Toc441149969)

[12.3 Deleting an Item from a Folder 93](#_Toc441149970)

[12.4 Deleting a Quick Search 94](#_Toc441149971)

[12.5 Deleting a Public Folder (Root Object) 94](#_Toc441149972)

[13 Chapter 13 - The PCDSQL Object 96](#_Toc441149973)

[13.1 PCDSQL Characterization Methods 97](#_Toc441149974)

[13.2 PCDSQL Transaction Method 97](#_Toc441149975)

[13.3 Other Methods 98](#_Toc441149976)

[14 Chapter 14 - Working with Activity Log Entries 100](#_Toc441149977)

[14.1 Retrieving Activity Log Entries 100](#_Toc441149978)

[14.2 Create a Custom Activity Log Entry 101](#_Toc441149979)

[15 Chapter 15 - Working with Public Folders 103](#_Toc441149980)

[15.1 Creating a Public Folder 103](#_Toc441149981)

[15.2 Modifying a Public Folder 105](#_Toc441149982)

[15.3 Retrieving a List of Public Folders 106](#_Toc441149983)

# Chapter 1 - An Overview of the DM Architecture and the DM API

eDOCS DM is a system for the storage, searching, and retrieval of documents, as well as the maintenance of associated document metadata. DM allows users to manage their documents without having to manage details of the implementation of their storage, such as remembering and traversing complicated file system hierarchies. It also allows users to seamlessly maintain customizable data related to but physically distinct from those documents. DM enforces security and customer-defined process and policy. The DM API allows developers to perform these same actions using a small but powerful collection of objects.

The DM architecture is designed to provide server- based document management functions to client-based and server-based applications by providing a three-tiered approach to document management that consist of the Data tier, Transactional tier, and the Client tier.

## Data Tier

The data tier is the DM Repository. The Data Repository is comprised of the following components:

* DM Library(s)
* Document Server(s)
* Full -Text index(s)

The DM Library is a database such as Microsoft SQL or Oracle. The library contains a collection of records known as Document Profiles. A Document Profile contains document metadata, including versioning, security, and the location of the physical file. Document Servers store the physical files on network file servers. The Full -Text index is a searchable collection of document contents.

## Transactional Tier

The Transactional Tier consists of one or more DM Servers. The DM server is responsible all document management functions and acts as an intermediary between the Client tier and DM Repository All interactions form the Client tier must interface with the DM server in order to access the DM Repository. A DM server is capable of connecting to multiple DM repositories and serving several clients simultaneously.

The DM Server is a stateless and free-threaded transaction server. The clients do not remain connected to the DM Server in a session. This implies that one method call is equal to one transaction on the DM Server with the exception of cached search results.

DM Servers may be configured in Fail Over and Load Balancing clusters (FOLB). Any server within the FOLB cluster may process a transaction with the exception of returning cached search results. When the clients initiate a transaction with a DM Server within the cluster the cluster configuration is completely transparent to the Client tier application.

## Client Tier

The Client tier is comprised of the interfaces provided to the end-user, and handles all database and file access. However, Client tier applications are not permitted direct access to DM Library, Document Servers, or Full- Text indexes; therefore, clients are configured to communicate with the DM Server to handle all transactions to the Data tier.

It is important to remember when working the DM API, that client access is stateless and transaction based a DM Security Token must be included when the Client tier submits a transaction.

From a user's perspective, DM is one or more of several interfaces, among them the following:

* DM Webtop,
* DM Extension for Microsoft Windows Explorer
* Application integration for such applications as Microsoft Word and Excel, and Adobe Acrobat

There are several main operations a user might perform in DM:

* Saving a document to DM
* Searching for documents in DM
* Retrieving a document from DM
* Modifying document metadata

## Forms in the DM API

* DM uses forms to access the database:
  + Defines primary table and Joins
  + Defines accessible columns
  + Central to the way that the DM API accesses data
* Forms are used to control the client’s read/write access to the columns in the database.
* Forms provide context information to lookups and specify what methods are available.
* Forms are accessed through the DM API.
* There are no DM API calls to display forms to the client application.
* Client applications must provide their own forms for use in their UI, or render.

When the client applications access the Data tier via the DM API it must specify a form by giving DM a Form name value. This will be seen in several examples throughout this guide. Forms are used to define the fields from a DM database table that can be accessed or modified. Forms are central to how the DM API is used to access the DM Repository. For example, if the client application is attempting an operation on a profile within the DM library, it does so by using a document profile form which contain fields directly mapped to the SQL column name. One thing to remember is that the DM API requires you provide the field name value configured in the form definition to access the SQL column. This is not always the same as the name of the SQL column. To be certain you must verify the field name in the form definition.

* **Note:** The value from “Name” field within the form’s definition is used by the DM API – not the values in the prompt or SQLInfo.

Different operations in DM require different types of forms. The important thing to remember is the Form chosen determines the fields available for use in the programmatic operation and the DM API does not directly render these forms to the user. There are five basic types of forms available.

* **Document Profile**: Used to enter documents or folders into the system, or access document metadata.
* **Search Form**: Used to search for objects and content in the DM Repository.
* **Hit List**: Used to show lists of documents that are a result of a search.
* **Lookup**: Used when presenting the user with validation data or a list of items for them to choose.
* **Maintenance Form**: Used to modify validation data.

## The Role of the DM API in the DM Architecture

The role of the DM API in the DM architecture is twofold. First, the DM API is an integral part of virtually everything that occurs in DM, be it from the perspective of a user or an administrator. The various end-user components of DM, including DM Extensions and Webtop, are essentially front-ends to DM API calls. This is most noticeable in DM Webtop where developers can open the various ASP pages and examine the DM API calls that make up the logic of the web client. The DM API is also used internally by several DM utilities. Second, as described in the next section, the DM API plays an important role in third-party applications and customer in-house applications that modify or build upon the DM architecture.

## An Overview of the DM API

The DM API allows developers to create applications that can perform all the actions of a user, with the added power and flexibility of programmatic automation. Developers using the DM API can also create applications that enhance, extend, or even replace the standard DM user interfaces. There are several types of applications that might use the functionality provided by DM, among them the following:

* + Standalone Windows applications that mimic the DM Explorer Extension in part or whole
  + Web-based applications that mimic the DM Webtop in part or whole.
  + • Application integration components analogous to the DM Extension for Microsoft Outlook COM Add-In or the DM integration macros for Microsoft Word.
  + Applications or application components that integrate an existing enterprise application (for example, a Customer Resource Management, or CRM, application) with DM.
  + Server-based applications that supplement standard DM installations, for example a Windows Service that monitors an incoming FTP site and saves newly deposited documents to a DM library.

In other words, the DM API allows developers to create applications ranging from integration and ancillary components accessing DM to complete replacement front-ends .DM provides developers two APIs with which to access DM, the DM API and the DM Extensions API. This reference guide describes the DM API. From a document-centric perspective, both APIs provide much the same functionality: what can be done with one API can generally, from this perspective, be done with the other API. One aspect that differentiates the APIs is the presence or lack of GUI components. The DM Extensions API provides many GUI components, whereas the DM API provides none. Developers may find the DM Extensions API GUI components a useful alternative to creating their own GUI components. Another aspect that differentiates the APIs is the presentation of DM constructs. The DM API, because it possesses no GUI components, has a leaner set of interfaces that focuses entirely upon DMS operations and does not possess the overhead costs to those GUI components. Because of this, the DM API is often the best choice for developers creating applications that focus primarily on business logic interacting with a DMS and applications in which the developer wants to control all aspects of the user interface.

Developers creating applications using the DM API will need the proper environment and tools. The proper environment could be a production DM environment but is, ideally, a test environment. The test environment will consist of, at a minimum, a DM Server, a database server, a file server, and a client with the DM API and development tools installed.

The DM API provides many functions; however, these functions do not always correspond to a single DM API method, property, or object. It may be required to combine multiple objects, methods and properties to complete one DM function.

The DM API is based on the DM Object Model that exists in the PCDClient.dll. This is located in the \Program Files\Open Text\DM API folder.

## DM Security Token (DST)

The DM server assigns a DM Security Token (DST) to client’s applications after successful login. A DST is required for all transactions to occur between a client application and DM server. A DST is a unique 64-character string used for security.

## Using DM API Objects

A series of common steps are required when using the DM API objects and each object contains methods that can be broken down into three classifications that will be used throughout this guide:

* **Characterization methods:** Used to define properties of objects.
* **Transaction methods:** Causes a transaction to occur between the Client tier application and DM Server. For example, searching for a document or uploading a document.
* **Retrieval methods:** Used after the transaction methods to retrieve information from the DM AP object and the data provided from the DM Server.

**An overview of the steps of using a DM API object:**

1. Create an instance of the object.
2. Call Characterization methods:

* Set the objects DM Security Token(DST) A correct DST must be set for transactions to be successful.
* Specify the library for the object. The library must be specified for each transaction.
* Specify an object – type via form Name.
* Specify properties particular to the object.

1. Call Transaction methods.
2. Perform Error Checking.
3. Call Retrieval Methods to retrieve information populated in the object by the DM Server.

Here is a general example illustrating the steps above:

public void ReturnDocumentMetadata(string library, string docNumber, string DST)

{

            string strPropsAndValues = string.Empty;

            //Create an instance of an Object

            PCDDocObject dmDoc = new PCDDocObject();

            //Call the Characterization methods

            dmDoc.SetDST(DST);  //Set the DST

            dmDoc.SetObjectType("LAWPROF"); // Specify the Form Type

            dmDoc.SetProperty("%TARGET\_LIBRARY", library); //Specify the Library

            dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

            //Call a Transaction Method

            dmDoc.Fetch();

            if (dmDoc.ErrNumber == 0) //Perform some error checking

            {

                //Call a Retrieval Method

                PCDPropertyList docPropList = dmDoc.GetReturnProperties();//Retrieval Method

                docPropList.BeginIter();

                int propListSize = docPropList.GetSize();

                for (int i = 0; i < propListSize; i++)

                {

                    strPropsAndValues = strPropsAndValues + " " + docPropList.GetCurrentPropertyName().ToString();

                    strPropsAndValues = strPropsAndValues + ":" + "  ";

                    strPropsAndValues = strPropsAndValues + " " + docPropList.GetCurrentPropertyValue().ToString();

                    strPropsAndValues = strPropsAndValues + " " + "\n";

                    docPropList.NextProperty();

                }

                MessageBox.Show(strPropsAndValues);

            }

            else

            {

                MessageBox.Show(dmDoc.ErrNumber.ToString() + ":" + " " + dmDoc.ErrDescription);

            }

 }

### Tokens in the DM API

The DM API uses the concept of tokens, special strings passed to properties and methods of the DM API in order to set or retrieve various properties and, in some cases, cause or modify certain actions. Tokens begin with the percent symbol, '%'. Tokens are context specific. Not all tokens work in all contexts. Tokens may have different effects in different contexts.

Tokens are passed to object methods as parameters and always begin with a % symbol, and must be all uppercase enclosed in quotes. This syntax is required because they are system variables defined in the DM object model, and are not defined in the local development environment.

* **Notes:** Tokens must be all in uppercase.  
  Must be surrounded by double quotes

- Example: “%TOKEN\_NAME”

The three purposes of Tokens are:

1. Specify characteristics or properties for the object.

Example: (“%OBJECT\_IDENTIFIER”)

1. Specify directives for server-side operation to manipulate the tables in the database.
2. Specify return values.

Example: (“%VERSION\_ID”)

# Chapter 2 – Authentication with the DM API

All significant sequences of DM API calls begin with authentication. The DM API, however, does not maintain or pass any state (such as authentication state) between its separately created objects. Therefore, in order to provide a security context for DM API objects, the objects are characterized, or qualified, with a DST, or Document Security Token. The DST is a unique hexadecimal string generated from various user account and environment properties.

A document security token (DST) is initially constructed by DM when a user logs on via a client application to the server and is required for DM transactions. The DST can be used across multiple DM Servers — when the user logs on to other DM Servers to access other Document libraries, the server information is appended to the DST — or the client application can use multiple DSTs.

Generally, authentication is performed once in a given application, and the DST is stored in a variable for later use. After DST acquisition, DM API objects are characterized using the SetDST method common to most creatable DM API objects. The SetDST method takes as its single argument a string, the DST. Those objects that do not have a SetDST method are generally not creatable and are returned by other, creatable objects, and thus are authenticated by proxy.

For some purposes, especially in the initial exploratory and prototyping phases of application development, acquiring the DST from the DM Extensions API DOCSObjects.Application and BCSession objects (part of the DM Extensions API) is a handy shortcut. If an instance of DM Extensions is open and connected, this eliminates the need to add logon code or write code to gather user credentials. Many of the samples in this guide assume authentication has been achieved, and the DST is passed in as a parameter in the samples. Here is a sample of using a BCSession object:

public string BCSessionLoginExample( ) //Call a logoff function

{

string strDST = string.Empty;

BCSession objBCSession = new BCSession();

try

{

Hummingbird.DM.Extensions.Interop.DOCSObjects.Application objDMapp = new Hummingbird.DM.Extensions.Interop.DOCSObjects.Application();

objBCSession.LogonEx();

strDST = objDMapp.DST;

}

catch (Exception e)

{

MessageBox.Show(e.ToString());

}

return strDST;

}

Another advantage is that authentication is performed for the developer: the user is presented with a DM Extensions logon user interface (UI) component (unless Auto Login is enabled, in which case there is no dialog and logon is automatic). This may occur prior to the custom DM API application starting up if the user has some form of DM Extensions session open. If the user is already logged on to DM Extensions, the DST is already available. If the user is not logged on to DM, the user is prompted to log on and, given a successful logon, the DST is available. In either case, the DST can be retrieved using the DOCSObjects.Application.DST property.

Note that simply creating the DOCSObjects.Application object does not in itself cause the user to be prompted to log on; the logon is presented (if the user is not already logged on) when a property or method of the Application object (such as DOCSObjects.Application.DST) is accessed.

The downside to acquiring the DST from the DM Extensions API is that, in the form above, only the user logged on to the workstation can be authenticated. This would likely not be a problem for an end-user application, but it would be a problem for a server-side application handling multiple user requests or incorporating business logic that requires API operations to be handled on behalf of multiple network or library accounts. In such cases, authentication is best achieved using the PCDClient.PCDLogin object.

public string SimplePCDLoginExample(string Library, string User, string Password)

{

string strDST;

PCDLogin oLogin = new PCDLogin();

try

{

oLogin.AddLogin(0,Library,User,Password);

oLogin.Execute();

}

catch (Exception e)

{

MessageBox.Show(e.ToString());

}

strDST = oLogin.GetDST();

return strDST;

}

After creation, the PCDLogin object can optionally be characterized by the SetServerName method which allows the developer to choose the DM Server to which the client logs on. If the method is not called, the Login object will attempt to log on to the default DM Login Server. The argument for SetServerName can be the server's short name, fully qualified DNS name, IP Address, or, potentially, any string that resolves to the IP Address of the server, such as "localhost" or "127.0.0.1" (if the code is executing on the DM Server). Thus, the following are equivalent:

objPCDLogin.SetServerName "192.168.1.101"

objPCDLogin.SetServerName "DMSERVER"

objPCDLogin.SetServerName "DMSERVER.PROD.LOCAL

Once the PCDLogin object has been properly characterized, a library must be chosen in preparation for passing it to the PCDLogin.AddLogin method. The library name can be drawn either statically, such as from a resource string as in the sample above, or dynamically, from the PCDGetLoginLibs object. This object is one of the few DM API objects that is creatable and which does not require characterization by a DST. Note that a given server can host multiple libraries, not all of which are Login Libraries; those that are not Login Libraries will not be returned. The following code iterates through a list of Login Libraries:

public static void GetLoginLibraryExample()

{

PCDGetLoginLibs objGetLoginLibs = new PCDGetLoginLibs();

objGetLoginLibs.Execute();

int intSize = objGetLoginLibs.GetSize();

for (int i = 0; i < intSize; i++)

{

MessageBox.Show(objGetLoginLibs.GetAt(i));

}

}

The PCDLogon.AddLogin method is used as follows:

AddLogin(Int16 iNetworkType, String Library, String zUserName, String zPassword)

The first parameter to PCDLogin.AddLogin specifies the logon type, either a Library Login or one of the varieties of network logon, as shown in the list below.

* 0 = Library Login
* 1 = Netware Bindery Network Login
* 2 = Netware NDS Network Login
* 8 = Microsoft Network Login

There are two DM login formats, Library Login and Network Login. A Library Login authenticates the user against information in the library database while the Network Login authenticates the user against information maintained by the Network Operating System.

The second parameter to PCDLogin.AddLogin specifies the library or a network container (for example, an Active Directory Services Domain), depending upon whether a Library Login or Network Login is being attempted. The third and fourth parameters, user ID and password, take values that also depend upon the type of logon.

The code below demonstrates the Library Login. The Add Login method is called once. Note that the password for the Library Login can be either an Attaché password or, if the library account has a network alias, the network password for the network alias.

public string LibraryPCDLoginExample(string Library, string User, string Password)

{

string strDST;

PCDLogin oLogin = new PCDLogin();

try

{

// The iNetworkType is set to O for a Library Login

oLogin.AddLogin(0,Library,User,Password);

oLogin.Execute();

}

catch (Exception e)

{

MessageBox.Show(e.ToString());

}

strDST = oLogin.GetDST();

return strDST;

}

The code below demonstrates a Network logon. The AddLogin method is called twice, once to specify the library (with an empty string for userID and password) and once to specify the network container, user ID, and password. Note that for the Network Login, only the network operating system (NOS) user ID and password are acceptable. The process is similar for Netware logons. Netware logons require an appropriate Bindery or NDS client be installed on the DM API client machine.

public string PCDLoginExample( string Domain, string Library, string User, string Password)

{

string strDST;

PCDLogin oLogin = new PCDLogin();

try

{

oLogin.AddLogin(0, Library, "", "");

oLogin.AddLogin(8,Domain, User, Password);

oLogin.Execute();

}

catch (Exception e)

{

MessageBox.Show(e.ToString());

}

strDST = oLogin.GetDST();

return strDST;

}

After the PCDLogin.Execute method is called, several properties of the PCDLogin object are populated with values that the developer may find useful. The following code demonstrates the use of some of these properties:

strDOCSUserName = objPCDLogin.GetDOCSUserName()

strLoginLibrary = objPCDLogin.GetLoginLibrary()

strPrimaryGroup = objPCDLogin.GetPrimaryGroup()

strServerName = objPCDLogin.GetServerName()

Optionally, after the PCDLogin.Execute method is called, if there were no errors, other useful information can be gathered from the PCDPropertyList object, a child object to the PCDLogin object. The PCDPropertyList class is instantiated by many DM API objects and provides a dynamic list of name-value pairs that depend upon the context of the object's instantiation. These pairs can be retrieved by means of iteration or by direct access given a property name. Note that in the current implementation of DM, the property name "%LOGIN\_TIME" does not possess a value; the logon time is subsumed by the "%LOGIN\_DATE" value.

public void LibraryPCDLoginPropExample(string Library, string User, string Password)

{

PCDLogin oLogin = new PCDLogin();

try

{

oLogin.AddLogin(0, Library, User, Password);

oLogin.Execute();

PCDPropertyList LoginProps = oLogin.GetReturnProperties();

LoginProps.BeginIter();

string strPropsAndValues = string.Empty;

int LoginPropListSize = LoginProps.GetSize();

for (int i = 0; i < LoginPropListSize; i++)

{

strPropsAndValues = strPropsAndValues + " " + LoginProps.GetCurrentPropertyName().ToString();

strPropsAndValues = strPropsAndValues + ":" + " ";

strPropsAndValues = strPropsAndValues + " " + LoginProps.GetCurrentPropertyValue().ToString();

strPropsAndValues = strPropsAndValues + " " + "\n";

LoginProps.NextProperty();

}

MessageBox.Show(strPropsAndValues);

}

## The PCDLogin Object

The PCDLogin object is used to login to validate the username, password and login mode. As with all DM API objects it’s methods are broken down into three distinct types: Characterization, Transaction, and Retrieval.

### Characterization Methods

**int** **AddLogin**(**short** *iNetworkType***, string** *zUnitName***, string** *zUserName***, string** *zPassword*)

This method specifies the login information.

**iNetworkType**: Specifies the logon type, either a Library Login or one of the varieties of network logon, as shown in the list below:

* 0 = Library Login
* 1 = Netware Bindery Network Login
* 2 = Netware NDS Network Login
* 8 = Microsoft Network Login

**zUnitName:** The name of the DM Library, Windows domain, or Netware NDS tree. The name of DM Library is specified when the iNetworkType is set “0”. The name of the domain is used when set to 1,2, or 8.

**zUserName:** The username for the specified user.

**zPassword:** The password for the specified user (unencrypted).

### Transaction Methods

**int** **Execute**()

This method authenticates the login information with the DM Server, the results are passed back to the PCDLogin object. No parameters are required.

### Retrieval Methods

**string** **GetDST**()

This method retrieves the DM Security Token after a successful Execute method call.

**string** **GetLoginLibrary**()

This method retrieves the name of the DM Library that was used to successfully log in.

**string** **GetDOCSUserName**()

This method retrieves the username that was used to successfully log in to the DM Library.

**string** **GetPrimaryGroup**()

This method retrieves the user’s primary group configured in DM.

**PCDPropertyList** **GetReturnProperties**()

This method returns a PCDPropertyList object of values that include the following:

* Target Library
* DM User Name
* Last Login Time and Date
* Current login time and date
* Current primary group

# Chapter 3 - Adding a Document to DM

The creation of a document in a DM library consists of two main conceptual steps, creating the Document Profile (a record in the PROFILE table) and uploading the physical file that resides in the document store. These two conceptual actions break down into the following concrete programming steps that require the PCDDocObject, PCDPutDoc, and PCDPutStream objects. This section describes how the PCDDocObject is used in profile creation, and how the PCDDocObject, PCDPutDoc objects are used together to upload the physical file. The methods for each object can be broken down into the three following types:

* Characterization Methods
* Transaction Methods
* Retrieval Methods

## PCDDocObject Role in the Creation of Documents

The PCDDocObject is an extremely flexible object used in document creation as well as other database entities, and is the only DM API object that is capable of read and write access to the database. It is actually one of the most common used DM API objects Although the PCDDocObject has many methods, only the methods needed to create a profile will be discussed in this chapter.

### Creating a New Document

The list below is an overview of the conceptual steps required to create a new document.

1. Create the Document Profile.
   1. Create the PCDDocObject.
   2. Characterize the PCDDocObject with authentication information via the DST.
   3. Characterize the PCDDocObject with the library name.
   4. Characterize the PCDDocObject with the library profile form to be used.
   5. Specify values for required profile fields and optional profile fields.
   6. Optionally verify that the document can be created.
   7. Characterize the PCDDocObject for testing profile creation using the tokens "%VERIFY\_ONLY" and "%YES".
   8. Call the create method of the PCDDocObject.
   9. Check the PCDDocObject error number and description.
   10. If an error exists, handle it appropriately.
   11. If no errors exist, characterize the PCDDocObject as before. Then characterize it for actual document creation by using the tokens %VERIFY\_ONLY" and "%NO" instead of the tokens "%VERIFY\_ONLY" and "%YES".
   12. Call the Create method of the PCDDocObject.
2. Create the physical file.
   1. Capture the Document Number and Version ID from the PCDDocObject.
   2. Create a PCDPutDoc object and a PCDPutStream object.
   3. Set the DST on the PCDPutDoc object.
   4. Specify search criteria for the PCDPutDoc object, using values captured from the PCDDocObject: Library, Document Number, Version.
   5. Call the Execute method of the PCDPutDoc object.
   6. Call the PCDPutDoc.NextRow method to set the result row to the first row returned.
   7. Set the PCDPutStream object equal to the object returned by the PCDPutDoc object.
   8. Using the facilities native to the programming environment, programming libraries, or the operating system, read from the local copy of the file to be saved to the library.
   9. Write the bytes read to the PCDPutStream object. Call the SetComplete method of the PCDPutStream object to signal the completion of data transfer.
   10. Create a new PCDDocObject.
   11. Characterize the new PCDDocObject with Profile Form, Library name, Version ID.
   12. Characterize the PCDDocObject for unlocking the document. Specify the Profile Form, Library, and Document Number. Next characterize the object by specifying the tokens "%STATUS" and "%UNLOCK".
   13. Call the Update method of the PCDDocObject to complete the unlocking operation.

* **Note:** The sample below contains additional code to assist in obtaining the physical file location, properties, and error checking.

public void ProfileNewDocumentExample(string libraryName, string documentName, string documentAuthor, string strDST)

{

PCDDocObject dmDoc = new PCDDocObject();

PCDPutDoc dmPutDoc = new PCDPutDoc();

PCDPutStream dmPutStream = new PCDPutStream();

string documentNumber;

string docext;

string versionID;

//Obtain and parse the file extension

OpenFileDialog dlgOpen = new OpenFileDialog();

dlgOpen.ShowDialog();

string strExtension = (System.IO.Path.GetExtension(dlgOpen.FileName));

strExtension = strExtension.Substring(1);

string filePath = dlgOpen.FileName;

string appID = null;

//Determine the APP\_ID based on the extension of the file

switch (strExtension)

{

case "doc":

case "docx":

appID = "MS WORD";

break;

case "xls":

case "xlsx":

appID = "MS EXCEL";

break;

case "ppt":

case "pptx":

appID = "MS POWERPOINT";

break;

case "msg":

appID = "MS OUTLOOK";

break;

}

try

{

//FIRST CREATE THE PROFILE

dmDoc.SetDST(strDST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY",libraryName );

dmDoc.SetProperty("DOCNAME", documentName);

dmDoc.SetProperty("TYPE\_ID", "DEFAULT");

dmDoc.SetProperty("FILE\_EXTENSION", strExtension);

dmDoc.SetProperty("APP\_ID", appID);

dmDoc.SetProperty("AUTHOR\_ID", documentAuthor);

dmDoc.SetProperty("TYPIST\_ID", documentAuthor );

dmDoc.SetProperty("CLIENT\_ID", "default");

dmDoc.SetProperty("MATTER\_ID", "default");

dmDoc.SetProperty("%VERIFY\_ONLY", "%YES");

dmDoc.Create();

if (dmDoc.ErrNumber == 0)

dmDoc.SetProperty("%VERIFY\_ONLY", "%NO");

else

throw (new System.Exception("Error attempting to create profile"));

dmDoc.Create();

if (dmDoc.ErrNumber == 0)

{

documentNumber = dmDoc.GetReturnProperty("%OBJECT\_IDENTIFIER").ToString();

versionID = dmDoc.GetReturnProperty("%VERSION\_ID").ToString();

}

else

{

throw (new System.Exception("Error" + ":" + dmDoc.ErrNumber.ToString() + " " + dmDoc.ErrDescription.ToString()));

}

//SECOND UPLOAD THE DOCUMENT

dmPutDoc.SetDST(strDST);

dmPutDoc.AddSearchCriteria("%TARGET\_LIBRARY", libraryName);

dmPutDoc.AddSearchCriteria("%DOCUMENT\_NUMBER", documentNumber);

dmPutDoc.AddSearchCriteria("%VERSION\_ID", versionID);

dmPutDoc.Execute();

dmPutDoc.NextRow();

dmPutStream = dmPutDoc.GetPropertyValue("%CONTENT") as PCDPutStream;

int byteswritten;

System.IO.FileStream fs;

System.IO.BinaryReader r;

int CHUNK\_SIZE = 65554;

fs = new System.IO.FileStream(filePath, System.IO.FileMode.Open, System.IO.FileAccess.Read);

r = new System.IO.BinaryReader(fs);

int FSize = (int)fs.Length;

byte[] chunk = r.ReadBytes(CHUNK\_SIZE);

while (chunk.Length > 0)

{

dmPutStream.Write(chunk, chunk.Length, out byteswritten);

if (FSize < CHUNK\_SIZE)

{

CHUNK\_SIZE = FSize;

chunk = r.ReadBytes(CHUNK\_SIZE);

}

else

chunk = r.ReadBytes(CHUNK\_SIZE);

}

dmPutStream.SetComplete();

if (dmPutStream.ErrNumber == 0)

{

docext=dmDoc.GetProperty("FILE\_EXTENSION");

MessageBox.Show("FILE HAS UPLOADED SUCCESSFULLY");

}

else

{

MessageBox.Show("ERROR:" + dmPutStream.ErrNumber + " " + dmPutStream.ErrDescription);

}

if (dmDoc != null)

Marshal.FinalReleaseComObject(dmDoc);

dmDoc = null;

fs.Close();

//Unlock Document

dmDoc = new PCDDocObject();

dmDoc.SetDST(strDST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", libraryName );

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", documentNumber);

dmDoc.SetProperty("%STATUS", "%UNLOCK");

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

throw (new System.Exception("ERROR: Could Not Unlock Document -" + dmDoc.ErrDescription));

}

catch (Exception ex)

{

MessageBox.Show(ex.ToString());

}

finally

{

if (dmPutStream != null)

Marshal.FinalReleaseComObject(dmPutStream);

if (dmPutDoc != null)

Marshal.FinalReleaseComObject(dmPutDoc);

if (dmDoc != null)

Marshal.FinalReleaseComObject(dmDoc);

}

}

The %OBJECT\_IDENTIFIER token is system generated when a new row is created in the PROFILE table, and returns the value in the PROFILE.DOCNUMBER column for the newly created row.

The %VERSION\_ID token is VERSION\_ID column on the VERSIONS table. A new row is created in the VERSIONS table whenever a new row is created in the PROFILE table.

The %TARGET\_LIBRARY token is mandatory must be specified before calling a transaction method, and used to specify the library.

Setting the %VERIFY\_ONLY token to a value of %YES is used to check for errors when calling the CREATE method and will return without actually creating a new entry in the PROFILE Table regardless if an error is returned or not. To actually create the Profile, you must set the %VERIFY only to %NO and call the CREATE method again. This is a useful technique to determine if the Profile can be created successfully.

### PCDDocObject Characterization Methods

**int** **SetDST**(**string** *zDST*)

This method specifies the DST for the object to initiate a transaction with the DM Server.

* *zDST: The Document Security Token from the PCDLogin object.*

**int** **SetObjectType**(**string** *zObjType*)

This method will specify the form name on which the operation will occur.

* *zObjType: The name of the form*

**int** **SetProperty**(**string** *zPropName***, object** *vVal*)

This method specifies the characteristics of the object as properties and corresponding values.

* *zPropName***:** *This value can be a field on the form or a token*
* *vVal: This can be a value for a field on the form, the value for a token, or another token depending upon use.*

### PCDDocObject Transaction Methods

**int** **Create**()

This method causes a transaction to occur between the client tier application and the DM Server. The result of a successful transaction is a new entry created in the primary table specified by the form defined in the SetObjectType characterization method. The values for the fields and tokens are specified by the SetProperty methods, and populate the column values in the database. If the Create method fails an error is returned to the application. It is the developer’s responsibility to check for, and handle errors.

### PCDDocObject Retrieval Methods

**dynamic** **GetReturnProperty**(**string** *zPropName*)

This method retrieves the properties that are returned as a result of successfully creating a Profile.

* *zPropName: Only the %OBJECT\_IDENTIFIER and %VERSION\_ID tokens are valid parameters after a successful CREATE method call.*

### The PCDPutDoc Object

The PCDPutDocObject is used to initialize and manage a set of file pointers, and must be provided with a document number and version identifier. The required information is retrieved from the Profile when it is created.

The Version identifiers are obtained to upload files as different versions for the same Profile. The combination of the document number and version identifier are used to identify an individual handle to a physical file on the document server.

### PCDPutDoc Characterization Methods

**int** **SetDST**(**string** *zDST*)

This method specifies the DST for the object to initiate a transaction with the DM Server

* *zDST: The DST to be used. This DST must be retrieved from the PCDLogin object.*

**int** **AddSearchCriteria**(**string** *zPropName***, object** *vCriteria*)

This method is used to specify the criteria needed to identify the handles to initialize and manage the correct file handles.

*- zPropName****:*** *Value of eDOCS DM Token*

*- vCriteria: The value to which the Token should be set*

*The valid tokens to use are:*

* *%TARGET\_LIBRARY - Specifies the library*
* *%DOCUMENT\_NUMBER - Specifies the document number*
* *%VERSION\_ID - Specifies the version identifier to be retrieved.*

### PCDPutDoc Transaction Methods

**int** **Execute**()

The purpose of this method is to use the criteria specified in the AddSearchCriteria method to determine the correct file pointers to initialize and return a handle for the PCDPutDoc object to manage these file pointers.

### PCDPutDoc Retrieval Methods

**int** **NextRow**()

This method must be called before calling another retrieval method to move the pointer to the correct file pointer.

**dynamic** **GetPropertyValue**(**string** *zPropName*)

This method is used to retrieve a specified property. Only the %CONTENT token is used to retrieve a handle to the PCDPutStream object.

* *zPropName: The %CONTENT token is the only valid parameter*

## The PCDPutStream Object

The PCDPutStream object is used to write the data into a file. The Client tier application must write the data and is the number of bytes is user defined. There are no characterization methods for the PCDPutStream object.

### PCDPutStream Transaction Method

**void** **Write**(**object** *vData***, int** *lBytes***,** out **int** *plBytesWritten*)

This method writes a specified size of bytes to a provided handle from the PCDPutDoc object. The handles are specified to the PCDPutStream object by calling the GetPropertyValue method with the %CONTENT token as a parameter from the PCDPutDoc object. This method returns a value of 0 if the write operation occurred without error.

* *vData: Indicates the data to write*
* *lBytes: Indicates the number of bytes to write*
* *plBytesWritten: Indicates the number of bytes written.*

### PCDPutStream Retrieval Methods

**int** **BytesWritten**()

This method returns the number of bytes that were successfully written by the Write operation.

**int** **SetComplete**()

This method closes associated file pointers and releases the resources used by the PCDPutStream object to write the file.

# Chapter 4 - Retrieving Documents and Metadata from the DM Repository

An important aspect of using eDOCS DM is retrieving documents and associated metadata from the repository. The samples included are extremely basic and included to illustrate the core concepts needed in the retrieval of documents and related information. This chapter will provide information on the DM API objects and code samples needed.

## Retrieving Library Documents

This section will cover how to retrieve a profile and download the contents of a file from the DM repository.

As described in the previous chapter, the two main conceptual steps to creating a library document are creating a Document Profile (database record) and uploading a physical file to the DM Server. Retrieving a library document also involves two main conceptual steps, getting a reference to an already existing document (abstracted as a PCDGetDoc object) and, using that object, getting a reference to a file-like object (abstracted as a PCDGetStream object) from which to read the document's bytes. The following process outlines the basic retrieval of a library document.

1. Create the PCDGetDoc object.
2. Characterize the object with the following:

* DST
* Library Name
* Document Name
* Version ID

1. Retrieve the document object.
2. Call the Execute method.
3. Set the row of the iterable collection of one document to the first and only row.
4. Call the GetPropertyValue method, passing the %CONTENT token, returning a reference to a PCDGetStream object.
5. Download the document using the PCDGetStream object.
6. Call the GetPropertyValue of the PCDGetStream object, passing the token "%ISTREAM\_STATSTG\_CBSIZE\_LOWPART", returning the size in bytes of the physical file.
7. Read bytes from the PCDGetStream object and write them to a physical file.

The code below demonstrates the procedure and with additional features to assist in creation of the local file, and error handling not included in the outline above. The developer must decide the download location and file name.

public void DownLoadDocExample(string libraryName, string docNumber, string strDST, string fileName, string VersionID)

{

string strLib = libraryName;

PCDGetDoc objGetDoc = new PCDGetDoc();

PCDGetStream objGetStream = new PCDGetStream();

objGetDoc.SetDST(strDST);

string Ext = ReturnFileExtension(strLib, strDST, docNumber);

objGetDoc.AddSearchCriteria("%TARGET\_LIBRARY", libraryName);

objGetDoc.AddSearchCriteria("%DOCUMENT\_NUMBER", docNumber);

objGetDoc.AddSearchCriteria("%VERSION\_ID", VersionID);

objGetDoc.Execute();

objGetDoc.SetRow(1);

if (objGetDoc.ErrNumber != 0)

{

MessageBox.Show("Error Number is: " + objGetDoc.ErrNumber + objGetDoc.ErrDescription);

}

objGetStream = objGetDoc.GetPropertyValue("%CONTENT");

if (objGetStream.ErrNumber != 0)

{

MessageBox.Show(objGetStream.ErrDescription);

}

int fileSize = objGetStream.GetPropertyValue("%ISTREAM\_STATSTG\_CBSIZE\_LOWPART");

int bytesRead = 1;

string DownloadFilename = "c:\\" + fileName + "." + Ext;

System.IO.BinaryWriter myBinaryWriter = new System.IO.BinaryWriter(new System.IO.FileStream(DownloadFilename, System.IO.FileMode.Create));

while (bytesRead != 0)

{

byte[] bytesToWrite = objGetStream.Read(fileSize, out bytesRead);

if (bytesRead != 0)

{

myBinaryWriter.Write(bytesToWrite);

}

}

myBinaryWriter.Close();

MessageBox.Show("Document Downloaded Sucessfully");

return;

}

public static string ReturnFileExtension(string libraryName, string DST, string docnum)

{

PCDSearch ps = new PCDSearch();

PCDDocObject dmdoc = new PCDDocObject();

string ext = string.Empty;

string extReturn = string.Empty;

ps.SetDST(DST);

ps.AddSearchLib(libraryName);

ps.SetSearchObject("v\_components");

ps.AddSearchCriteria("DOCNUMBER", docnum);

ps.AddReturnProperty("PATH");

ps.Execute();

int rowsfound = ps.GetRowsFound();

if (rowsfound > 0)

{

ps.NextRow();

ext = ps.GetPropertyValue("PATH");

}

ps.ReleaseResults();

extReturn = System.IO.Path.GetExtension(ext);

extReturn = extReturn.Substring(1);

return extReturn;

}

## Returning Version Information

The PCDGetDoc object, like the PCDPutDoc object, requires the Version identifier and document number be specified. These are the VERSION\_ID field from the VERSION table, and the DOCNUMBER field from the PROFILE table. Since it is possible for a Profile to have many versions of a file it is important to gather this information to download the correct file from the DM Repository.

Building a search becomes extremely important as the tasks become more complex. In the following example the search is based on the VERSIONS table by using the cyd\_cmnversions form. All of the rows with the DOCNUMBER field indicating the Profile that the Version belongs to should be returned. Since multiple versions can be returned for a single profile it the responsibility of the Client tier application to determine which VERSION\_ID is to be used. It is common to write logic into the UI to allow a user to select a Version. However, that is beyond the scope of this sample.

public void ReturnAllVersionInfo(string library, string DST, string docNumber)

{

string msg = string.Empty;

PCDSearch VerSearch = new PCDSearch();

VerSearch.SetDST(DST);

VerSearch.SetSearchObject("cyd\_cmnversions");

//Specify Return Properties

VerSearch.AddReturnProperty("VERSION\_ID");

VerSearch.AddReturnProperty("VERSION");

VerSearch.AddReturnProperty("TYPIST");

VerSearch.AddReturnProperty("COMMENTS");

//Search for all versions that are related to the to the document number.

//DOCNUMBER is the name of the field on the cyd\_cmnversions form

VerSearch.AddSearchCriteria("DOCNUMBER", docNumber);

VerSearch.AddOrderByProperty("VERSION",0);

try

{

VerSearch.Execute();

//Check for errors

if(VerSearch.ErrNumber !=0)

{

MessageBox.Show(VerSearch.ErrNumber.ToString() +": " + VerSearch.ErrDescription);

}

else

{

VerSearch.SetRow(0);

for (int i = 0; i < VerSearch.GetRowsFound(); i++)

{

VerSearch.NextRow();

msg = msg +"VERSION\_ID: " + VerSearch.GetPropertyValue("VERSION\_ID") + "\n";

msg = msg +"VERSION: " + VerSearch.GetPropertyValue("VERSION")+ "\n";

msg = msg +"TYPIST: " + VerSearch.GetPropertyValue("TYPIST")+ "\n";

msg = msg + "COMMENTS: "+ VerSearch.GetPropertyValue("COMMENTS")+ "\n";

msg = msg + "\n";

}

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

MessageBox.Show(msg);

VerSearch.ReleaseResults();

}

## Returning Document Metadata

The sample below illustrates how to use a PCDPropertyList object to return profile metadata.

public void ReturnDocumentMetadata(string library, string DST,string docNumber)

{

PCDDocObject dmDoc = new PCDDocObject();

string strPropsAndValues = string.Empty;

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

try

{

dmDoc.Fetch();

if (dmDoc.ErrNumber == 0)

{

PCDPropertyList docPropList = dmDoc.GetReturnProperties();

docPropList.BeginIter();

int propListSize = docPropList.GetSize();

for (int i = 0; i < propListSize; i++)

{

strPropsAndValues = strPropsAndValues + " " + docPropList.GetCurrentPropertyName().ToString();

strPropsAndValues = strPropsAndValues + ":" + " ";

strPropsAndValues = strPropsAndValues + " " + docPropList.GetCurrentPropertyValue().ToString();

strPropsAndValues = strPropsAndValues + " " + "\n";

docPropList.NextProperty();

}

}

else

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

MessageBox.Show(strPropsAndValues);

}

## The PCDGetDoc Object

The PCDGetDoc object is used exactly like the PCDPutDoc object but is used to initialize and manage a set of file pointers for reading from a file. The PCDGetDoc object requires the DST must be set, and the library, document number, and Version identifier be specified as mentioned in the section above. It is important to remember that an individual document is identified by the document number and Version ID.

### PCDGetDoc Characterization Methods

**int** **SetDST**(**string** *zDST*)

This method specifies the DST (DM Security Token) for the current object. This method needs to be called to specify the DST during the transaction with the DM Server.

*zDST*: Specifies the DST to be used. This DST must be retrieved with the PCDLogin object.

**int** **AddSearchCriteria**(**string** *zPropName***, object** *vCriteria*)

This method specifies the properties for the PCDGetDoc object that are required to initialize and manage the correct file pointers for reading from a file.

* *zPropName:* *The name of the DM Token.*
* *vCriteria*: *The value to which to set the DM Token.*

The valid tokens to use as properties for the PCDGetDoc object are:

**int** **Execute**()

Determines the correct file pointers to initialize to point to a file on the document server based on the parameters provided by the AddSearchCriteria method.

*%TARGET\_LIBRARY – This token specifies the library.*

*%DOCUMENT\_NUMBER – This token specifies the document number*

*%VERSION\_ID – This token specifies the Version identifier to retrieve*.

### PCDGetDoc Transaction Methods

**int** **Execute**()

Determines the correct file pointers to initialize to point to a file on the document server based on the parameters provided by the AddSearchCriteria method.

### PCDGetDoc Retrieval and Other Methods

**dynamic** **GetPropertyValue**(**string** *zPropName*)

This method retrieves a specified property. However, the %CONTENT Token is the only valid value that may be passed to this method. The %CONTENT Token is used to retrieve a handle to the PCDGetStream object so the contents of the file can be read from the document server.

* *zPropName: Only the %CONTENT Token can be used to retrieve the PCDPutStream object.*

**int** **NextRow**()

This method must be called before calling a Retrieval method. It moves the pointer to the correct file pointer.

## PCDGetStream Object

The purpose of this object is to read from the specified stream provided by the PCDGetDoc object. The PCDGetStream object must be used with the PCDGetDoc object because it does not have any characterization methods.

### PCDGetStream Transaction Methods

**dynamic** **Read**(**int** *lBytes***,** out **int** *plBytesRead*)

This method returns the data that was read from the handle set to the PCDGetStream object by calling the GetPropertyValue method with the %CONTENT Token.

* *lBytes: Indicates the size of bytes to read from the stream.*
* *plBytes Read: Indicates the bytes read.*

### PCDGetStream Retrieval and Other Methods

**dynamic** **GetPropertyValue**(**string** *zPropName*)

This method is used to get information regarding the stream. A Token that is often used is the %ISTREAM\_STATSTG\_CBSIZE\_LOWPART Token. This returns the total size of the file to be streamed.

* *zPropName: Specifies the Token about which to retrieve information.*

**int** **BytesRead**()

This method returns the number of bytes that was returned during the Read operation.

**int** **SetComplete**()

This method was closes the file pointer and releases the resources used by the PCDGetStream object to download a file from DM.

# Chapter 5 - Searching for Documents

A core functionality of a DM is the ability to search for documents based on document metadata. Searches in DM include searching the Profile and indexes for documents marked for full-text indexing. The DM API allows both types of searches, as well as searches on other types of information in the library. The type of search that is performed depends upon the form or object used. A form essentially acts as an interface to a table in the database.

eDOCS DM provides forms specifically used for searches. However, because there are no real constraints, it is possible to use a Profile form to perform searches as long as the required fields exist on the form.

There are two types of searches that are performed in DM. The first is a Profile Search that search is based on Profile metadata. There are also Full-Text searches that search within document content. This requires an index be configured for the DM Library where the document resides.

Search results are built on the DM Server and are sent asynchronously to the Client tier application. It is possible to specify the chunk size, and maximum number of results. DM Server also performs the sorting and merging of search results. This consumes memory so it is important that the results are released by the Client tier application as soon as possible. Searches are processed in parallel and can span multiple libraries. The DM Server will only return results that the user has adequate privileges to, and the only viewable field values are those specified.

The three types of search criteria available are:

* Metadata Searches
* Full-text Searches
* Hybrid Searches.

## Metadata Searches

A Metadata search uses search criteria where the search results are generated via SQL queries. These will search the DM Library for items like Profiles, Quick Searches, and Versions. Multiple search criteria specified implies a logical AND. For example, a search could be performed for a particular AUTHOR AND TYPE\_ID. A Logical OR is implied on values within one field or one AddSearchCriteria method call using a comma to separate each search value.

## Full-text Index Searches

A Full-text Index Search searches documents marked full-text indexing. These documents can be searched in three methods – the Profile only, Content only, or Profile and Content. Only one search criterion can be specified for a search. The search criteria properties are:

FULLTEXT\_PROFILE

FULLTEXT\_CONTENT

FULLTEXT\_CONTENT\_PROFILE

Logical OR and logical AND operators can be used to refine search criteria. In addition, the <NOT>, and the <NEAR> operators are also supported. Concatenating these operators provide a powerful search mechanism, especially when coupled with the supported wildcard characters such as “\*”. It is also important to note that searches are not case sensitive.

## Hybrid Searches

Search criteria can be specified using Tokens. One of the supported Tokens is the %EXCLUDE\_FOLDERS token. When specified as part of the search criteria, no folders are returned as part of the search results. By default, folders are returned as part are included in all the search results.

## Creating a Search with the DM API

Executing a search via the DM API only requires the use of one object type. It requires the steps below, and it is extremely important to release resources on the DM server when the search is completed.

The main steps in performing a search are as follows:

1. Create PCDSearch object and characterize it with a DST, Library name, and Form name.
2. Specify return properties.
3. Specify search criteria.
4. Specify the Order By property to sort results
5. Execute the search.
6. Iterate through the results.
7. Release the PCDSearch object results set to release resources on the DM Server.

It is important to note that any property to be searched on must be available on the form used in the search. For instance, if a form does not contain the Matter ID field, it is not possible to search for documents based upon the Matter ID even if a value exists in that column in PROFILE table. The code samples below contain additional error handling for increased readability and only serve to illustrate the core concepts.

The following code demonstrates performing a profile search (based on document metadata):

public void ProfileSearchExample(string library, string DST)

{

PCDSearch dmSearch = new PCDSearch();

dmSearch.SetDST(DST);

//Set Search Form and Library

dmSearch.SetSearchObject("LAWPROF");

dmSearch.AddSearchLib(library);

//You can set maximum rows returned

dmSearch.SetMaxRows(5);

//Add Return Properties

dmSearch.AddReturnProperty("DOCNUM");

dmSearch.AddReturnProperty("DOCNAME");

dmSearch.AddReturnProperty("AUTHOR\_ID");

dmSearch.AddReturnProperty("CLIENT\_ID");

dmSearch.AddReturnProperty("MATTER\_ID");

dmSearch.AddReturnProperty("TYPIST\_ID");

dmSearch.AddReturnProperty("TYPE\_ID");

dmSearch.AddReturnProperty("LASTEDITDATE");

dmSearch.AddReturnProperty("%DOCS\_LIBRARY\_NAME");

//Specify prperties to search for. Wildcards are allowed

dmSearch.AddSearchCriteria("DOCNAME", "\*TEST");

//Option to exclude folders from being returned

dmSearch.AddSearchCriteria("%EXCLUDE\_FOLDERS", "");

//Specify Return Order

dmSearch.AddOrderByProperty("DOCNAME", 0);

dmSearch.Execute();

string msg = " ";

for (int i = 0; i < dmSearch.GetRowsFound(); i++)

{

dmSearch.NextRow();

msg = msg +"DOCNUMBER: "+ dmSearch.GetPropertyValue("DOCNUM") + "\n";

msg = msg +"DOCNAME: "+dmSearch.GetPropertyValue("DOCNAME") + "\n";

msg = msg +"AUTHOR: "+dmSearch.GetPropertyValue("AUTHOR\_ID") + "\n";

msg = msg +"CLIENT: "+dmSearch.GetPropertyValue("CLIENT\_ID") + "\n";

msg = msg +"MATTER: "+dmSearch.GetPropertyValue("MATTER\_ID") + "\n";

msg = msg +"TYPIST: " +dmSearch.GetPropertyValue("TYPIST\_ID") + "\n";

msg = msg +"DOCTYPE: "+ dmSearch.GetPropertyValue("TYPE\_ID") + "\n";

msg = msg +"LAST EDIT: "+dmSearch.GetPropertyValue("LASTEDITDATE")+"\n";

msg = msg + "\n";

}

MessageBox.Show(msg);

//Release the search results to free server resources

dmSearch.ReleaseResults();

}

The following code demonstrates a Full-text Search (based on document content):

public void FullTextSearchExample(string library, string DST)

{

PCDSearch dmSearch = new PCDSearch();

dmSearch.SetDST(DST);

//Set Search Form and Library

dmSearch.SetSearchObject("LAWQBE");

dmSearch.AddSearchLib(library);

//You can set maximum rows returned

dmSearch.SetMaxRows(5);

//Add Return Properties

dmSearch.AddReturnProperty("DOCNUM");

dmSearch.AddReturnProperty("DOCNAME");

dmSearch.AddReturnProperty("AUTHOR\_ID");

dmSearch.AddReturnProperty("CLIENT\_ID");

dmSearch.AddReturnProperty("MATTER\_ID");

dmSearch.AddReturnProperty("TYPIST\_ID");

dmSearch.AddReturnProperty("TYPE\_ID");

dmSearch.AddReturnProperty("LASTEDITDATE");

dmSearch.AddReturnProperty("%DOCS\_LIBRARY\_NAME");

//YOU MUST ADD THE RELEVANCE FOR A FT SEARCH

dmSearch.AddReturnProperty("%FT\_SCORE");

dmSearch.AddSearchCriteria("FULLTEXT\_CONTENT", "Search Text");

//Option to exclude folders from being returned

dmSearch.AddSearchCriteria("%EXCLUDE\_FOLDERS", "");

//Specify sort based on relevance:

dmSearch.AddOrderByProperty("%FT\_SCORE", 1);

//Execute and return results.

dmSearch.Execute();

string msg = "";

for (int i = 0; i < dmSearch.GetRowsFound(); i++)

{

dmSearch.NextRow();

msg = msg + "DOCNUMBER: " + dmSearch.GetPropertyValue("DOCNUM") + "\n";

msg = msg + "DOCNAME: " + dmSearch.GetPropertyValue("DOCNAME") + "\n";

msg = msg + "AUTHOR: " + dmSearch.GetPropertyValue("AUTHOR\_ID") + "\n";

msg = msg + "CLIENT: " + dmSearch.GetPropertyValue("CLIENT\_ID") + "\n";

msg = msg + "MATTER: " + dmSearch.GetPropertyValue("MATTER\_ID") + "\n";

msg = msg + "TYPIST: " + dmSearch.GetPropertyValue("TYPIST\_ID") + "\n";

msg = msg + "DOCTYPE: " + dmSearch.GetPropertyValue("TYPE\_ID") + "\n";

msg = msg + "LAST EDIT: " + dmSearch.GetPropertyValue("LASTEDITDATE") + "\n";

msg = msg + "RELEVANCE: " + dmSearch.GetPropertyValue("%FT\_SCORE") + "\n";

msg = msg + "\n";

}

MessageBox.Show(msg);

//Release the search results to free server resources

dmSearch.ReleaseResults();

}

## The PCDSearch Object

The PCDSearch object is a generalized search object for searching the DM Repository. The PCDSearch object follows the same generic pattern as other DM API objects. The DST, library, and object type (Form) must be specified. It also contains the standard Characterization, Transaction, and Retrieval methods.

### PCDSearch Characterization Methods

**int** **SetDST**(**string** *zDST*)

This method specifies the DST (DM Security Token) for objects to initiate a transaction with the DM Server. DM uses the DST for security purposes. The SetDST method requires one parameter:

* *zDST: Specifies the DM Security Token to be used. The DST must be retrieved from the PCDLogin object.*

**int** **SetSearchObject**(**string** *zObjName*)

This method specifies the search object-type to be used. This is the form name. The form name specifies the Document Object (or table) to search and fields available that are viewable.

* *zObjName: Specifies the name of the Search Form to be used.*

**int** **AddSearchLib**(**string** *zLibName*)

This method specifies the library to search. To specify more than one library, this method is called more than one time.

* *zLibName: Specifies the name of the library to search.*

Note: If specifying more than one library be aware that forms can be different in different libraries. It is easiest to search on fields that are common between different forms. However, if a field is returned that does not exist in the library a string value of “UNKNOWN PROPERTY” is returned.

**int** **AddReturnProperty**(**string** *zPropName*)

This method specifies a column of field to be returned after a successful search has been executed.

* *zPropName: The name of the field in a Form that is to be returned*

**int** **SetReturnProperties**(**object** *pPropList*)

This method specifies a list or columns or fields to be returned after a successful search has been executed.

* *pPropList: The list of fields stored in the PCDPropertyList object.*

Note: As part of using the PCDSearch object return properties must be specified. Return properties are the field /columns from the table to be returned. If no return properties are specified, it is possible for a search to complete successfully but have no results returned. This is because no return properties have been specified.

**int** **AddSearchCriteria**(**string** *zPropName***, string** *zCriteria*)

This method specifies the criteria for a search.

* *zPropName: The field name to search, or Token*
* *zCriteria: The value for which to search*

**int** **SetSearchCriteria**(**object** *pUnk*)

* *pUnk : The list of search criteria that includes the property name and property values.*

### PCDSearch Transaction Method

**int** **Execute**()

This method call causes a search to be invoked on the DM Server.

### Optional PCDSearch Characterization Methods

The PCDSearchobject provides optional Characterization methods that can be used to improve performance or change how the results are returned by the DM Server.

**int** **SetRow**(**int** *ulRowNbr*)

* *ulRowNbr: The location is the list results to which to set the pointer. To set to the beginning of the row, a value of 0 is used.*

**int** **NextRow**()

This method iterates to the next row in the list of results returned by the DM Server.

**int** **ReleaseResults**()

This releases all the results from where the current pointer is to the beginning of the list. It is important to call this method as quickly as possible.

### PCDSearch Object Retrieval Methods

**int** **GetRowsFound**()

This method returns the number of results (rows) found for a particular search. Values may differ depending on the maximum number of rows specified to be returned.

**dynamic** **GetPropertyValue**(**string** *zPropName*)

This method returns a string value of the specified property. The property must exist as a return property. If the property does not exist on the Form a value of **%UNKNOWN\_PROPERTY** is returned.

* *zPropName*: *The string value of the field name of the column in the Form .*

**PCDPropertyList** **GetReturnProperties**()

This method returns a list of property names associated with values in the PCDPropertyList object.

# Chapter 6 - Retrieving and Modifying Document Metadata

The DM API provides the functionality to retrieve and change document metadata.

## Returning Document Metadata

The following code shows how to return document metadata:

public void ReturnDocumentMetadata(string library, string docNumber, string DST)

{

PCDDocObject dmDoc = new PCDDocObject();

string strPropsAndValues = string.Empty;

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.Fetch();

PCDPropertyList docPropList = dmDoc.GetReturnProperties();

docPropList.BeginIter();

int propListSize = docPropList.GetSize();

for (int i = 0; i < propListSize; i++)

{

strPropsAndValues = strPropsAndValues +" "+ docPropList.GetCurrentPropertyName().ToString();

strPropsAndValues = strPropsAndValues + ":" + " ";

strPropsAndValues = strPropsAndValues + " " + docPropList.GetCurrentPropertyValue().ToString();

strPropsAndValues = strPropsAndValues + " " + "\n";

docPropList.NextProperty();

}

MessageBox.Show(strPropsAndValues);

}

## Modifying Document Metadata

The DM API provides the ability to change document metadata. The following code demonstrates changing the name of a document:

public void ModifyDocumentMetaData(string library, string DST, string docNumber)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("DOCNAME", "NEW DOC NAME");

try

{

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

}

catch(Exception ex)

{

MessageBox.Show(ex.Message);

}

}

## Modifying the Status of a Profile

Modifying the Status field is central to managing documents being modified by multiple users. The Status field is a read-only field on the Form. Therefore, the %STATUS Token along with additional Tokens are used to allow the Client tier application to modify this value. These Tokens are:

* **%LOCK\_FOR\_CHECKOUT /%LOCK**

This Token changes the STATUS field on the PROFILE table to a value of 3 which is equivalent to “Checked Out”. This will also create a new entry in the CHECKOUT table. Other Tokens can be used along with %LOCK\_FOR\_CHECKOUT in separate PCDDocObject.SetProperty method calls to add additional information to other fields in the CHECKOUT table. Weather a document is locked or checked- out is the same from a DM perspective. The reason the two different tokens exist is that the %LOCK Token may indicate that the profile is simply unavailable; however, the LOCK\_FOR\_CHECKOUT may indicate the file needs to be downloaded as well. This logic is determined by the Client tier application. The additional Tokens used to modify the CHECKOUT table are:

* **% CHECKIN\_LOCATION** – This Token indicates the drive or directory where the document where the document was copied. It modifies the LOCATION filed on the CHECKOUT table
* **%CHECKOUT\_COMMENTS** – This Token specifies comments concerning the reason that the document was checked out. This modifies the COMMENTS field in the table.
* **%CHECKIN\_DATE**– This token specifies when the document is to be returned, or when the Profile is to be unlocked. This modifies the CHECKIN\_DATE field in the table.
* **%UNLOCK**

To unlock a profile, the %UNLOCK Token is used. This Token changes the STATUS filed in the Profile table to a value of 0, which is equivalent to available. It also removes the entry in the CHECKOUT table.

* **%MAKE\_READ\_ONLY**

Making a profile Read-Only changes the STATUS field on the PROFILE table to 19, which is equivalent to Read-Only. Making this change does not allow any changes or additions to be made to the Profile. This includes addition of versions or attachments.

* **%REMOVE\_READ\_ONLY**

Removing the Read-Only STATUS from a Profile makes it possible for changes and additions to the Profile.

It is important to note that for a change in the STATUS to occur, it must be a valid Status Change Transition; if the transition is invalid an error will be returned.

Valid Transitions are:

* + %LOCK\_FOR\_CHECKOUT or %LOCK to %UNLOCK
  + **%**UNLOCK to %LOCK\_FOR\_CHECKOUT /%LOCK or %MAKE\_READ\_ONLY
  + %MAKE\_READ\_ONLY to %REMOVE\_READ\_ONLY

Changing the STATUS of a Profile can be broken down into the following programmatic steps:

1. Create a PCDDocObject.
2. Specify the DST and Library.
3. Specify the object-type (Form).
4. Specify the property to identify the existing Profile using the %OBJECT\_IDENTIFIER Token.
5. Use the %STATUS Token and one of the additional Tokens above.
6. Invoke the PCDDocObject.
7. Update transaction method call.
8. Check for errors.

Note: If an invalid status transition occurs an error will be reported after the PCDDocObject.Update transaction method call.

## Checking Out Documents

public void CheckingOutDocExample(string DST, string Library, string docNumber, string fileLocation, string dateToReturn, string Comments)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", Library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%STATUS", "%LOCK\_FOR\_CHECKOUT");

dmDoc.SetProperty("%CHECKIN\_LOCATION", fileLocation);

dmDoc.SetProperty("%CHECKIN\_DATE", dateToReturn);

dmDoc.SetProperty("%CHECKOUT\_COMMENTS", Comments);

try

{

dmDoc.Update();

if(dmDoc.ErrNumber !=0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

}

catch(Exception ex)

{

MessageBox.Show(ex.Message);

}

}

## Modifying the Read-Only Status

One of the metadata attributes of a document that can be set is the read-only attribute. The following code demonstrates setting and removing read-only status:

public void MakeDocumentsReadOnly(string DST,string Library, string docNumbers)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", Library);

//You can pass single or multiple docnumbers seperated by a comma

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumbers);

dmDoc.SetProperty("%STATUS", "%MAKE\_READ\_ONLY");

try

{

dmDoc.Update();

if(dmDoc.ErrNumber !=0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

}

}

catch(Exception ex)

{

MessageBox.Show(ex.Message);

}

}

public void RemoveReadOnly(string DST,string Library, string docNumbers)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", Library);

//You can pass single or multiple docnumbers seperated by a comma

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumbers);

dmDoc.SetProperty("%STATUS", "%REMOVE\_READ\_ONLY");

try

{

dmDoc.Update();

if(dmDoc.ErrNumber !=0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

}

}

catch(Exception ex)

{

MessageBox.Show(ex.Message);

}

}

* **Note:** The PCDDocObject.SetProperty method can accept multiple documents as a comma-separated list with "%OBJECT\_IDENTIFIER" Token.

Ex. dmDoc.SetProperty("%OBJECT\_IDENTIFIER","15080,15082,15083");

## Unlocking a Profile

public void UnlockProfileExample(string DST,string Library, string docNumber)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", Library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%STATUS", "%UNLOCK");

try

{

dmDoc.Update();

if(dmDoc.ErrNumber !=0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

}

}

catch(Exception ex)

{

MessageBox.Show(ex.Message);

}

}

# Chapter 7 - Managing Security in eDOCS DM

Managing security is one of the more involved tasks when working with the DM API; however, document security is an integral and essential element of DM. The DM API provides procedures for determining the rights that the current user has for a given document and modifying those rights. Every secured document has an Access Control List (ACL), which lists the users and groups and their respective rights to profiles.

A *Trustee* is a term used to describe a person or group that has access to DOCSObjects. Trustee information consists of a Trustee Name based on either the GROUP\_ID column on the GROUPS table, or the USER\_ID in the PEOPLE Table; a Trustee Flag determines if it is a user or group and trustee rights. The trustee name, flags, and rights are not stored in the same database table as the particular profile; instead, they are stored in the SECURITY table. The SECURITY table consists of the following fields:

* **THING** – This field describes the unique database identifier of an item.
* **PERSONORGROUP** – This field describes the unique database identifier (SYSTEM\_ID) from the PEOPLE or GROUPS table.
* **ACCESSRIGHTS-** This field is the additive rights for the Trustee.

## Access Rights and Tokens

Access Rights in DM are stored as a 32-bit value where a bit indicates a particular right. Rights are additive and the values are used to calculate the sum of rights. Tokens are provided to determine and set these rights; however, rights can be specified using the DM API with numeric values.

|  |  |  |  |
| --- | --- | --- | --- |
| **Hex-Value** | **DB Value** | **Token or Value** | **Description** |
| 0x00000001 | 1 | %PR\_VIEW | View Profile |
| 0x00000002 | 2 | %PR\_EDIT | Edit Profile |
| 0x00000004 | 4 | %PR\_CONTENT\_VIEW | View Document |
| 0x00000008 | 8 | %PR\_CONTENT\_RETRIEVE | Retrieve Document |
| 0x00000010 | 16 | %PR\_CONTENT\_EDIT | Edit Document |
| 0x00000020 | 32 | %PR\_CONTENT\_COPY | Copy Document |
| 0x00000040 | 64 | %PR\_CONTENT\_DELETE | Delete Document |
| 0x00000080 | 128 | %PR\_ACCESS\_CONTROL | Access Control |
| 0x00000100 | 256 | %RIGHT8 | Assign to File |
| 0x00000200 | 512 | %RIGHT9 | View Published |

An example of calculating rights would be if a user has only View Profile and Edit Profile they would have a rights value of 3. These calculations can be used when setting access rights for a particular trustee. A list of rights and their additive values has been provided at the end of this chapter.

## Deny Rights

Deny rights are used to explicitly deny access to a trustee for a particular profile. The Deny rights are commonly used to override the Access rights. This is common in environments where a group may have access, but certain individuals are not permitted access. There are no Tokens available for Deny rights.

|  |  |  |  |
| --- | --- | --- | --- |
| **Hex-Value** | **DB Value** | **Token or Value** | **Description** |
| 0x00010000 | 65536 | None | Deny View Profile |
| 0x00020000 | 131072 | None | Deny Edit Profile |
| 0x00040000 | 262144 | None | Deny View Document |
| 0x00080000 | 534288 | None | Deny Retrieve Document |
| 0x00100000 | 1048576 | None | Deny Edit Document |
| 0x00200000 | 2097152 | None | Deny Copy Document |
| 0x00400000 | 4194304 | None | Deny Delete Document |
| 0x00800000 | 8388608 | None | Deny Access Control |
| 0x01000000 | 16777216 | None | Deny Assign to File |
| 0x02000000 | 33554432 | None | Deny View Published |

## Working with Trustee Rights

When working with Trustee rights it is often required to retrieve a list of rights to either display, modify, add, or delete a trustee. A convenient Token has been provided called %EFFECTIVE \_RIGHTS that returns what rights a user has to a profile.

The following code demonstrates the procedure for determining the current user's rights for a given document and version:

public void CurrentUserDocRights(string library, string DST, string docNumber)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("TARGET\_LIBRARY", library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

try

{

dmDoc.Fetch();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

}

int accessRights = dmDoc.GetReturnProperty("%EFFECTIVE\_RIGHTS");

Dictionary<string, string> dictRightsTokens = new Dictionary<string, string>();

// Example: dictRightsTokens.Add("Key,Value)

dictRightsTokens.Add("View Profile", "%PR\_VIEW");

dictRightsTokens.Add("Edit Profile", "%PR\_EDIT");

dictRightsTokens.Add("View Document Content ", "%PR\_CONTENT\_VIEW");

dictRightsTokens.Add("Retrieve Document Content", "%PR\_CONTENT\_RETRIEVE");

dictRightsTokens.Add("Edit Document Content", "%PR\_CONTENT\_EDIT");

dictRightsTokens.Add("Copy Document Content", "%PR\_CONTENT\_COPY");

dictRightsTokens.Add("Delete Document Content", "%PR\_CONTENT\_DELETE");

dictRightsTokens.Add("Control Access to Document", "%PR\_ACCESS\_CONTROL");

dictRightsTokens.Add("Assign to File", "%RIGHT8");

dictRightsTokens.Add("View Only Published", "%RIGHT9");

string blHasRight;

string msg = " ";

string caption = " ";

foreach (KeyValuePair<string, string> kvp in dictRightsTokens)

{

int intResult = dmDoc.HasRight(kvp.Value.ToString(), (uint)accessRights);

if (intResult == 1)

{

blHasRight = "TRUE";

}

else

{

blHasRight = "FALSE";

}

msg = msg + kvp.Key.ToString() + " = " + blHasRight + "\n";

caption = "Rights for Document# " + docNumber + "\n";

}

MessageBox.Show(msg + caption);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

## Retrieving a List of Trustees and Rights

If security is not applied to the document (that is, the **Security** check box on the Profile form is not selected), then the ACL for the document is empty. If security is applied to the document, then the document will have at least one user on the ACL, the owner of the document. For each user on the ACL, it is possible to discover the rights that the user has for the document in question. The following code demonstrates this:

public void RetrievingTrusteesandRights( string Library, string DST,string docNumber)

{

Dictionary<string, string> dictRightsTokens = new Dictionary<string, string>();

// Example: dictRightsTokens.Add("Key,Value)

dictRightsTokens.Add("View Profile", "%PR\_VIEW");

dictRightsTokens.Add("Edit Profile", "%PR\_EDIT");

dictRightsTokens.Add("View Document Content ", "%PR\_CONTENT\_VIEW");

dictRightsTokens.Add("Retrieve Document Content", "%PR\_CONTENT\_RETRIEVE");

dictRightsTokens.Add("Edit Document Content", "%PR\_CONTENT\_EDIT");

dictRightsTokens.Add("Copy Document Content", "%PR\_CONTENT\_COPY");

dictRightsTokens.Add("Delete Document Content", "%PR\_CONTENT\_DELETE");

dictRightsTokens.Add("Control Access to Document", "%PR\_ACCESS\_CONTROL");

dictRightsTokens.Add("Assign to File", "%RIGHT8");

dictRightsTokens.Add("View Only Published", "%RIGHT9");

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("TARGET\_LIBRARY", Library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

try

{

dmDoc.Fetch();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

int accessRights = dmDoc.GetReturnProperty("%EFFECTIVE\_RIGHTS");

int userModifyControl = dmDoc.HasRight("%PR\_ACCESS\_CONTROL", (uint)accessRights);

if (userModifyControl == 1)

{

dmDoc.FetchTrustees();

PCDTrusteeList trusteeList = dmDoc.GetTrustees();

trusteeList.BeginIter();

for (int i = 0; i < trusteeList.GetSize(); i++)

{

string trusteename = trusteeList.GetCurrentTrusteeName();

int rights = trusteeList.GetCurrentTrusteeRights();

string blHasRight = string.Empty;

string msg = string.Empty;

foreach (KeyValuePair<string, string> kvp in dictRightsTokens)

{

int intResult = dmDoc.HasRight(kvp.Value.ToString(), (uint)rights);

if (intResult == 1)

{

blHasRight = "TRUE";

}

else

{

blHasRight = "FALSE";

}

msg = msg + kvp.Key + "=" + blHasRight + "\n";

}

MessageBox.Show("Rights for user " + trusteename + ":" + "\n" + msg);

msg = string.Empty;

trusteeList.NextTrustee();

}

}

else

{

MessageBox.Show("You do not have rights to modify this document");

}

}

catch(Exception ex)

{

MessageBox.Show(ex.Message);

}

}

## Adding and Removing Trustees to a Profile Access Control List

It is common to add or delete trustees from a Profile. The code below illustrates how to modify the Access Control List by adding or removing trustees.

The following are the conceptual steps needed to add trustee to an Access Control List (ACL):

1. Create an instance of the PCDDocObject.
2. Specify the DST, Library, and Object Type (Form).
3. Specify the unique identifier for the Profile using the %OBJECT\_IDENTIFIER Token (Document Number).
4. Retrieve the Profile by calling the PCDDocObject.Fetch() method.
5. Retrieve the rights of the current user from the Profile with the %EFFECTIVE\_RIGHTS Token. Use this value to determine if the current user has Access Control privileges to modify the ACL.
6. Use the PCDDocObject.FetchTrustees method to retrieve the list of trustees and copy them into the PCDDocObject.
7. Assign a copy of the list of trustees from the PCDDocObject to the PCDTrusteeList by using the PCDDocObject.GetTrustees method.
8. Update the list of trustees by calling the PCDDocObject.UpdateTrustees method.
9. Optionally, update the Profile with any information by calling the PCDDocObject.Update method.

Note: This sample will add a trustee with the same Access Rights as the user logged on. However, you may set the access rights equal to the numeric values in the chart above. It is also important to note the FetchTrustees method from PCDDocObject is called before the AddTrustee method otherwise the original list of trustees will be overwritten.

public void AddTrusteeToACL(string library, string DST, string docNumber, string UserorGroup)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

try

{

dmDoc.Fetch();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

int accessRights = dmDoc.GetReturnProperty("%EFFECTIVE\_RIGHTS");

int accessControl = dmDoc.HasRight("%PR\_ACCESS\_CONTROL", (uint)accessRights);

if (accessControl == 0)

{

MessageBox.Show("You do not have rights to modify Trustees");

return;

}

dmDoc.FetchTrustees();

PCDTrusteeList trusteeList = dmDoc.GetTrustees();

trusteeList.AddTrustee(UserorGroup, 2, accessRights);

dmDoc.SetTrustees(trusteeList);

dmDoc.UpdateTrustees();

dmDoc.Update();

}

catch(Exception ex)

{

MessageBox.Show(ex.Message);

}

}

The following are the conceptual steps needed to remove a trustee to an Access Control List (ACL):

1. Create an instance of the PCDDocObject.
2. Specify the DST, Library, and Object Type (Form).
3. Specify the unique identifier for the Profile using the %OBJECT\_IDENTIFIER Token (Document Number).
4. Retrieve the Profile by calling the PCDDocObject.Fetch() method.
5. Retrieve the rights of the current user from the Profile with the %EFFECTIVE\_RIGHTS Token. Use this value to determine if the current user has Access Control privileges to modify the ACL.
6. Use the PCDDocObject.FetchTrustees method to retrieve the list of trustees and copy them into the PCDDocObject.
7. Assign a copy of the list of trustees from the PCDDocObject to the PCDTrusteeList by using the PCDDocObject.GetTrustees method.
8. Delete the trustee based on name from the PCDTrusteeList object. You must determine the trustee index and delete a trustee based on its location in the PCDTrusteeList object.
9. Update the list of trustees by calling the PCDDocObject.UpdateTrustees method.
10. Optionally, update the Profile with any information by calling the PCDDocObject.Update method.

public void RemoveTrusteeFromACL(string library, string DST, string docNumber, string UserorGroup)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

try

{

dmDoc.Fetch();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

int accessRights = dmDoc.GetReturnProperty("%EFFECTIVE\_RIGHTS");

int accessControl = dmDoc.HasRight("%PR\_ACCESS\_CONTROL", (uint)accessRights);

if (accessRights == 0)

{

MessageBox.Show("You dont have enough rights to modify Trustees");

}

dmDoc.FetchTrustees();

PCDTrusteeList trusteeList = dmDoc.GetTrustees();

int trusteeIndex = trusteeList.GetTrusteeIndex(UserorGroup, 2);

trusteeList.DeleteTrustee(trusteeIndex);

dmDoc.SetTrustees(trusteeList);

dmDoc.UpdateTrustees();

dmDoc.Update();

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

## The PCDDocObject Methods for Security Management

### Transaction Methods

**int** **FetchTrustees**()

This method retrieves a list of Trustees from the SECURITY table. Since the rights are not stored in the PROFILE table, but are in the SECURITY table instead, a separate transaction is required to retrieve the list of trustees and copy them to the PCDDocObject.

### Retrieval Methods

**PCDTrusteeList** **GetTrustees**()

This method returns a PCDTrusteeList object reference to a list of trustees from a PCDDocObject.

**int** **HasRight**(**string** *zRightName***, uint** *uiRightsIn*)

This method returns a value of True if the user or group has the specified rights. If the specified trustee does not have rights a value of false is returned (value of 0).

* *zRightName***:** *This parameter only accepts Tokens.*
* *uiRightsIn: Specifies the additive rights for a user or group.*

## PCDTrusteeList Object

The PCDTrusteeList object is used to read a modify security for Profiles. An item in the PCDTrusteeList object consists of a trustee name, trustee flag and trustee rights.

### Characterization Methods

**int** **AddTrustee**(**string** *zTrusteeName***, int** *iTrusteeFlags***, int** *iTrusteeRights*)

This method adds a new trustee to an existing list of trustees. If the trustee exists in the list, the rights for the existing trustee are updated with new ones.

* *zTrusteeName: The string value name of the trustee. For a user trustee, the name is the USER\_ID column form the PEOPLE table. For a group trustee, the name is the GROUP\_ID from the GROUPS table.*
* *iTrusteeFlags***:** *A numeric value indicating whether the trustee is a user or group. (1= Group, 2= User)*
* *iTrusteeRights*: An integer value of the additive rights for the trustee.

**int** **DeleteTrustee**(**int** *ulNdx*)

This method deletes a trustee from the PCDTrusteeList based on index.

* *ulNdx*: The location in the list where the trustee is to be deleted.

**int** **SetTrusteeRights**(**int** *ulNdx***, int** *iTrusteeRights*)

This method changes the access rights for a trustee at a specified location within the list.

* *ulNdx***:**  *The location of the trustee in the list based on index*
* *iTrusteeRights: The new rights for a specified trustee.*

### Retrieval Methods

**int** **GetTrusteeIndex**(**string** *bzTrusteeName***, int** *iTrusteeFlags*)

This method returns an integer value to the location of the specified user or group.

- *bzTrusteeName***:** The string name of the trustee.

- *iTrusteeFlags:* *Indicates whether the string value specified is a user or group.*

**int** **GetTrusteeRights**(**int** *ulNdx*)

This method returns an integer value of the rights associated with the trustee at a specified location in the list.

*-ulNdx: Specifies the location in the list to get the rights of the trustee.*

**int** **GetSize**()

This method returns the number of trustees in the list.

**string** **GetCurrentTrusteeName**()

This method returns the string value of the trustee.

**int** **GetCurrentTrusteeRights**()

This method returns the value of the rights for the trustee at the current location in the list.

**int** **GetCurrentTrusteeFlags**()

This method returns the value of the trustee at the current location in the list.

### Helper Methods

**int** **BeginIter**()

This method sets the current iteration pointer to the beginning of the list of trustees.

**int** **NextTrustee**()

This method iterates to the next trustee in the list. It will return an error if there are no more items in the list.

## iTrusteeRights Combinations

The list below contains the integer value that represents the additive permissions. The integer value can be passed as part of the PCDTrusteeList.AddTrustee method.

|  |  |
| --- | --- |
| 1 | View Profile |
| 3 | View Profile Edit Profile |
| 5 | View Profile View Document |
| 7 | View Profile Edit Profile View Document |
| 13 | View Profile View Document Retrieve Document |
| 15 | View Profile Edit Profile View Document Retrieve Document |
| 29 | View Profile View Document Retrieve Document Edit Content |
| 31` | View Profile Edit Profile View Document Retrieve Document Edit Content |
| 39 | View Profile Edit Profile View Document Copy |
| 47 | View Profile Edit Profile View Document Retrieve Document Copy |
| 63 | View Profile Edit Profile View Document Retrieve Document Edit Content Copy |
| 127 | View Profile Edit Profile View Document Retrieve Document Edit Content Copy Delete |
| 255 | View Profile Edit Profile View Document Retrieve Document Edit Content Copy Delete Control Access |
| 517 | View Profile View Document Access Published Only |

# Chapter 8 - Document Versions and Attachments

## An Overview of Versions and Attachments

DM manages versions of documents, sub-versions of versions of documents, and attachments to documents by adding a record to the VERSIONS table. Versions, sub-versions, and attachments are all handled similarly via the DM API. When a document is retrieved it is necessary to specify a version identifier because a version is stored as a separate physical file on the document server. Multiple versions and sub-versions can exist for a single profiled document. Attachments are treated the same as a version with the exception that they are identified by a user supplied label rather than a generated version or sub-version label.

When a new version, sub-version, or attachment is added to a profile, a new entry is inserted into the VERSIONS table. The version identifier is the unique identifier in the VERSIONS table specified by the VERSION\_ID column. The VERSION table has other notable columns:

* VERSION: The version of the document is a value such as 1, 2 or 3 for an entry that is Version or Sub-Version. If the entry is an attachment, the VERSION value is 0.
* SUB-VERSION: The letter of the sub-version such as A, B, or C. If the row does not specify a sub-version the column will have a value of “!” for a version, and empty for an attachment.
* VERSION\_LABEL: If the entry is a version or sub-version it is a combination of the VERSIONS and SUB-VERSION column. If the entry is not a sub-version or version the value is the extension value of the attachment.

## Creating a Version, Sub-version, or Attachment

When a new version, sub-version, or attachment is added to a profile a new entry is created in the VERSIONS table. To reference the new entry from the PCDDocObject, the unique version Identifier is required by using the “%VERSION\_ID” Token. The “%VERSION\_ID” and document are both required for uploading and downloading the file. The steps for creating a new version, sub-version, or attachment are:

1. Retrieve the document and lock the profile. The profile must be locked to prevent a conflict. If it is not locked an error will occur when adding a new version, sub-version or attachment.
2. Create a PCDDocObject object.
3. Specify the DST for the library.
4. Specify the Form.
5. Specify the property to identify the existing profile using the “%OBJECT\_IDENTIFIER” Token. When adding a sub-version both the document number and version identifier needs to be included.
6. Specify whether to create a new version, sub-version, or attachment. The PCDDocObject. SetProperty characterization method is used with token indicating which action to take.
7. Create a new version by calling the Update() method to retrieve the newly created version identifier. Use the PCDDocObject.GetReturnProperty() and the “%VERSION\_ID” is used to retrieve the new version identifier.
8. Check for errors.
9. Upload the file into DM
10. Unlock the profile.

public void CreateNewVersion(string library, string DST, string docNumber, string author, string typist)

{

//Obtain the file to be uploaded and parse the file extension

OpenFileDialog dlgOpen = new OpenFileDialog();

dlgOpen.ShowDialog();

string filePath = dlgOpen.FileName;

try

{

//Lock the Profle

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%STATUS", "%LOCK");

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

{

throw (new System.Exception("ERROR: Could Not Lock Document -" + dmDoc.ErrDescription));

}

//Create the version with a new instance of the PCDDocObject

dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%VERSION\_DIRECTIVE", "%PCD\_NEW\_VERSION");

dmDoc.SetProperty("%VERSION\_AUTHOR", author);

dmDoc.SetProperty("%VERSION\_TYPIST", typist);

dmDoc.SetProperty("%VERSION\_COMMENT", "YOUR COMMENTS");

dmDoc.Update();

string versionID = dmDoc.GetReturnProperty("%VERSION\_ID").ToString();

if (dmDoc.ErrNumber != 0)

{

throw (new System.Exception("ERROR: " + dmDoc.ErrDescription));

}

//Upload the new version

PCDPutDoc dmPutDoc = new PCDPutDoc();

PCDPutStream dmPutStream = new PCDPutStream();

dmPutDoc.SetDST(DST);

dmPutDoc.AddSearchCriteria("%TARGET\_LIBRARY", library);

dmPutDoc.AddSearchCriteria("%DOCUMENT\_NUMBER", docNumber);

dmPutDoc.AddSearchCriteria("%VERSION\_ID", versionID);

dmPutDoc.Execute();

dmPutDoc.NextRow();

dmPutStream = dmPutDoc.GetPropertyValue("%CONTENT") as PCDPutStream;

int byteswritten;

System.IO.FileStream fs;

System.IO.BinaryReader r;

int CHUNK\_SIZE = 65554;

fs = new System.IO.FileStream(filePath, System.IO.FileMode.Open,

System.IO.FileAccess.Read);

r = new System.IO.BinaryReader(fs);

int FSize = (int)fs.Length;

byte[] chunk = r.ReadBytes(CHUNK\_SIZE);

while (chunk.Length > 0)

{

dmPutStream.Write(chunk, chunk.Length, out byteswritten);

if (FSize < CHUNK\_SIZE)

{

CHUNK\_SIZE = FSize;

chunk = r.ReadBytes(CHUNK\_SIZE);

}

else

chunk = r.ReadBytes(CHUNK\_SIZE);

}

dmPutStream.SetComplete();

if (dmPutStream.ErrNumber == 0)

{

MessageBox.Show("FILE HAS UPLOADED SUCCESSFULLY");

}

else

{

MessageBox.Show("ERROR:" + dmPutStream.ErrNumber + " " + dmPutStream.ErrDescription);

}

fs.Close();

//Unlock the document

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%STATUS", "%UNLOCK");

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

throw (new System.Exception("ERROR: Could Not Unlock Document -" + dmDoc.ErrDescription));

}

catch (Exception ex)

{

MessageBox.Show(ex.ToString());

}

}

## Creating a Sub-version of a Document

public static CreateSub-version(string docNumber, string targetVersionID, string author, string typist, string DST, string library)

{

//Obtain the file to be uploaded.

OpenFileDialog dlgOpen = new OpenFileDialog();

dlgOpen.ShowDialog();

string filePath = dlgOpen.FileName;

try

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY",library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%STATUS", "%LOCK");

dmDoc.Update();

//Create the version with a new instance of a PCDDocObject

dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%VERSION\_ID", targetVersionID);

dmDoc.SetProperty("%VERSION\_DIRECTIVE", "%PCD\_NEWSUB-VERSION");

dmDoc.SetProperty("%VERSION\_AUTHOR", author);

dmDoc.SetProperty("%VERSION\_TYPIST", typist);

dmDoc.SetProperty("%VERSION\_COMMENT", "YOUR COMMENTS");

dmDoc.Update();

string versionID = dmDoc.GetReturnProperty("%VERSION\_ID").ToString();

//Upload the new version

PCDPutDoc dmPutDoc = new PCDPutDoc();

PCDPutStream dmPutStream = new PCDPutStream();

dmPutDoc.SetDST(DST);

dmPutDoc.AddSearchCriteria("%TARGET\_LIBRARY", library);

dmPutDoc.AddSearchCriteria("%DOCUMENT\_NUMBER", docNumber);

dmPutDoc.AddSearchCriteria("%VERSION\_ID", versionID);

dmPutDoc.Execute();

dmPutDoc.NextRow();

dmPutStream = dmPutDoc.GetPropertyValue("%CONTENT") as PCDPutStream;

int byteswritten;

System.IO.FileStream fs;

System.IO.BinaryReader r;

int CHUNK\_SIZE = 65554;

fs = new System.IO.FileStream(filePath, System.IO.FileMode.Open,

System.IO.FileAccess.Read);

r = new System.IO.BinaryReader(fs);

int FSize = (int)fs.Length;

byte[] chunk = r.ReadBytes(CHUNK\_SIZE);

while (chunk.Length > 0)

{

dmPutStream.Write(chunk, chunk.Length, out byteswritten);

if (FSize < CHUNK\_SIZE)

{

CHUNK\_SIZE = FSize;

chunk = r.ReadBytes(CHUNK\_SIZE);

}

else

chunk = r.ReadBytes(CHUNK\_SIZE);

}

dmPutStream.SetComplete();

if (dmPutStream.ErrNumber == 0)

{

MessageBox.Show("FILE HAS UPLOADED SUCCESSFULLY");

}

else

{

MessageBox.Show("ERROR:" + dmPutStream.ErrNumber + " " + dmPutStream.ErrDescription);

}

fs.Close();

//Unlock the document with new instance of a PCDDocObject

dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%STATUS", "%UNLOCK");

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

throw (new System.Exception("ERROR: Could Not Unlock Document -" +

dmDoc.ErrDescription));

}

catch (Exception ex)

{

MessageBox.Show(ex.ToString());

}

}

## Adding an Attachment to a Document

public void AddAttachmentToDocument(string docNumber, string author, string typist, string DST,string library)

{

//Obtain the new file and parse the file extension

OpenFileDialog dlgOpen = new OpenFileDialog();

dlgOpen.ShowDialog();

string strExtension = (System.IO.Path.GetExtension(dlgOpen.FileName));

strExtension = strExtension.Substring(1);

string filePath = dlgOpen.FileName;

try

{

//Lock the Document

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%STATUS", "%LOCK");

dmDoc.Update();

//Create the new attachment

dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%VERSION\_DIRECTIVE", "%ADD\_ATTACHMENT");

dmDoc.SetProperty("%ATTACHMENT\_ID", strExtension);

dmDoc.SetProperty("%VERSION\_AUTHOR", author);

dmDoc.SetProperty("%VERSION\_TYPIST", typist);

dmDoc.SetProperty("%VERSION\_COMMENT", "YOUR COMMENTS");

dmDoc.Update();

string versionID = dmDoc.GetReturnProperty("%VERSION\_ID").ToString();

//Upload the attatchment

PCDPutDoc dmPutDoc = new PCDPutDoc();

PCDPutStream dmPutStream = new PCDPutStream();

dmPutDoc.SetDST(DST);

dmPutDoc.AddSearchCriteria("%TARGET\_LIBRARY", library);

dmPutDoc.AddSearchCriteria("%DOCUMENT\_NUMBER", docNumber);

dmPutDoc.AddSearchCriteria("%VERSION\_ID", versionID);

dmPutDoc.Execute();

dmPutDoc.NextRow();

dmPutStream = dmPutDoc.GetPropertyValue("%CONTENT") as PCDPutStream;

int byteswritten;

System.IO.FileStream fs;

System.IO.BinaryReader r;

int CHUNK\_SIZE = 65554;

fs = new System.IO.FileStream(filePath, System.IO.FileMode.Open,

System.IO.FileAccess.Read);

r = new System.IO.BinaryReader(fs);

int FSize = (int)fs.Length;

byte[] chunk = r.ReadBytes(CHUNK\_SIZE);

while (chunk.Length > 0)

{

dmPutStream.Write(chunk, chunk.Length, out byteswritten);

if (FSize < CHUNK\_SIZE)

{

CHUNK\_SIZE = FSize;

chunk = r.ReadBytes(CHUNK\_SIZE);

}

else

chunk = r.ReadBytes(CHUNK\_SIZE);

}

dmPutStream.SetComplete();

if (dmPutStream.ErrNumber == 0)

{

MessageBox.Show("FILE HAS UPLOADED SUCCESSFULLY");

}

else

{

MessageBox.Show("ERROR:" + dmPutStream.ErrNumber + " " + dmPutStream.ErrDescription);

}

fs.Close();

//Unlock the document with new instance of a PCDDocObject

dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%STATUS", "%UNLOCK");

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

throw (new System.Exception("ERROR: Could Not Unlock Document -" +

dmDoc.ErrDescription));

}

catch (Exception ex)

{

MessageBox.Show(ex.ToString());

}

}

## Retrieving a Version, Sub-version, or Attachment

Retrieving a version, sub-version, or attachment is essentially the same operation as retrieving a document. The difference lies in one call to PCDGetDoc.AddSearchCriteria where the version ID is specified. If the version ID belongs to a document attachment you must use the VERSION\_LABEL as the file extension, otherwise the file extension of the document can be used.

public void DownnloadSpecificVersion(string libraryName, string docNumber, string strDST,string versionID, string fileName)

{

string strLib = libraryName;

string ext = string.Empty;

PCDDocObject dmdoc = new PCDDocObject();

PCDGetDoc objGetDoc = new PCDGetDoc();

PCDGetStream objGetStream = new PCDGetStream();

objGetDoc.SetDST(strDST);

//Determine if the Version ID submitted belongs to a Version or Attachment

//Attachments have a Version Number of 0, which requires the VERSION\_LABEL be used as the file extension

string verNum = ReturnVersionNumber(libraryName, strDST, docNumber, versionID);

if (verNum == "0")

{

ext = ReturnVersionLabel(libraryName, strDST, docNumber, versionID);

}

else

{

ext = ReturnFileExtension(strLib, strDST, docNumber);

}

objGetDoc.AddSearchCriteria("%TARGET\_LIBRARY", libraryName);

objGetDoc.AddSearchCriteria("%DOCUMENT\_NUMBER", docNumber);

objGetDoc.AddSearchCriteria("%VERSION\_ID", versionID);

objGetDoc.Execute();

objGetDoc.SetRow(1);

if (objGetDoc.ErrNumber != 0)

{

MessageBox.Show("Error Number is: " + objGetDoc.ErrNumber + objGetDoc.ErrDescription);

}

objGetStream = objGetDoc.GetPropertyValue("%CONTENT");

if (objGetStream.ErrNumber != 0)

{

MessageBox.Show(objGetStream.ErrDescription);

}

int fileSize = objGetStream.GetPropertyValue("%ISTREAM\_STATSTG\_CBSIZE\_LOWPART");

int bytesRead = 1;

string DownloadFilename = "c:\\" + fileName + "." + ext;

System.IO.BinaryWriter myBinaryWriter = new System.IO.BinaryWriter(new System.IO.FileStream(DownloadFilename, System.IO.FileMode.Create));

while (bytesRead != 0)

{

byte[] bytesToWrite = objGetStream.Read(fileSize, out bytesRead);

if (bytesRead != 0)

{

myBinaryWriter.Write(bytesToWrite);

}

}

myBinaryWriter.Close();

MessageBox.Show("Document Downloaded Sucessfully");

return;

}

public string ReturnVersionLabel(string libraryName, string DST, string docnum, string verID)

{

PCDSearch ps = new PCDSearch();

PCDDocObject dmDoc = new PCDDocObject();

string vLabel = string.Empty;

ps.SetDST(DST);

ps.AddSearchLib(libraryName);

ps.SetSearchObject("v\_versions");

ps.AddSearchCriteria("DOCNUMBER", docnum);

ps.AddSearchCriteria("VERSION\_ID", verID);

ps.AddReturnProperty("VERSION\_LABEL");

ps.Execute();

int rowsfound = ps.GetRowsFound();

if (rowsfound > 0)

{

ps.NextRow();

vLabel = ps.GetPropertyValue("VERSION\_LABEL");

}

ps.ReleaseResults();

return vLabel;

}

public string ReturnFileExtension(string libraryName, string DST, string docnum)

{

PCDSearch ps = new PCDSearch();

PCDDocObject dmdoc = new PCDDocObject();

string ext = string.Empty;

string extReturn = string.Empty;

ps.SetDST(DST);

ps.AddSearchLib(libraryName);

ps.SetSearchObject("v\_components");

ps.AddSearchCriteria("DOCNUMBER", docnum);

ps.AddReturnProperty("PATH");

ps.Execute();

int rowsfound = ps.GetRowsFound();

if (rowsfound > 0)

{

ps.NextRow();

ext = ps.GetPropertyValue("PATH");

}

ps.ReleaseResults();

extReturn = System.IO.Path.GetExtension(ext);

extReturn = extReturn.Substring(1);

return extReturn;

}

public string ReturnVersionNumber(string libraryName, string DST, string docNum, string verID)

{

PCDSearch ps = new PCDSearch();

PCDDocObject dmDoc = new PCDDocObject();

string verNum = string.Empty;

ps.SetDST(DST);

ps.AddSearchLib(libraryName);

ps.SetSearchObject("v\_versions");

ps.AddSearchCriteria("DOCNUMBER", docNum);

ps.AddSearchCriteria("VERSION\_ID", verID);

ps.AddReturnProperty("VERSION");

ps.Execute();

int rowsfound = ps.GetRowsFound();

if (rowsfound > 0)

{

ps.NextRow();

verNum = ps.GetPropertyValue("VERSION");

}

ps.ReleaseResults();

return verNum;

}

## Deleting a Version, Sub-version, or Attachment

The following code deletes a version, sub-version, or attachment, given a document number and the version ID of the item.

public bool DeleteVersion(string libraryName,string DST, string docNumber, string versionID)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", libraryName);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%VERSION\_ID", versionID);

dmDoc.SetProperty("%VERSION\_DIRECTIVE", "%PCD\_DELETE\_VERSION");

dmDoc.Update();

if (dmDoc.ErrNumber == 0)

{

return true;

}

else

{

return false;

}

}

## Determining the Version ID of a Document, Version or Attachment

Most operations on a version, sub-version, or attachment, as well as many other DM API operations, require a version ID. As users generally deal with version labels, rather than version IDs, it is often necessary to translate between the two.

Given a document number and a version label (the attachment ID, in the case of an attachment), the following function returns a version ID.

public string ReturnVersionID(string libraryName, string DST, string docnum, string versionLabel )

{

PCDSearch ps = new PCDSearch();

PCDDocObject dmDoc = new PCDDocObject();

string verID = string.Empty;

//string verLabel = string.Empty;

ps.SetDST(DST);

ps.AddSearchLib(libraryName);

ps.SetSearchObject("VersionsSearch");

ps.AddSearchCriteria("%OBJECT\_IDENTIFIER", docnum);

ps.AddReturnProperty("VERSION\_ID");

ps.AddReturnProperty("VERSION\_LABEL");

ps.Execute();

int rowsfound = ps.GetRowsFound();

for (int i = 0; i < rowsfound; i++)

{

ps.SetRow(i + 1);

if (ps.GetPropertyValue("VERSION\_LABEL") == versionLabel)

{

verID = ps.GetPropertyValue("VERSION\_ID");

ps.ReleaseResults();

}

}

return verID;

}

## Tokens Used when Working with Versions, Sub-Versions, and Attachments

**%VERSION\_DIRECTIVE**: This is specified using the PCDDocObject.SetProperty method prior to the update.

Valid property value tokens to use with %VERSION\_DIRECTIVE are:

* %PCD\_NEW\_VERSION: Specifies to create a new version for the specified profile
* %PCD\_NEWSUB-VERSION: Specifies to create a new sub-version for the specified profile
* %ADD\_ATTACHMENT: Specifies to create a new attachment for the specified profile.
* %ATTACHMENT\_ID: Used only for attachments.

# Chapter 9 - Relating and Unrelating Documents

Relating documents allows for the logical association between documents. The links are bi-directional which allows for effective management of documents. It is possible to relate items within a single library or remote libraries. The DM API does not limit the type of logical associations created. Two documents can be related no matter the status of the profile, version, or permissions.

Once a relation is created, a new entry is inserted into the RELATED table. The associations are bi-directional without transitive properties implied. For example, if document 1 is related to document 2 and document 2 is related to document 3, document 1 is not related to document 3.

To relate or unrelate documents, the steps are similar to changing the status of a profile. The effect of relating a document causes the RELATED field in the PROFILE table to be set to a value of “Y” and the related items are added as entries into the RELATED table.

The PCDDocObject is used to relate and unrelate two profiles. The “%OBJECT\_IDENTIFIER” token specifies the profile that the relation will be created on. The “%RELATED\_ITEM” token specifies the item that will be related to the specified profile.

The conceptual steps needed to relate or unrelate items are:

1. Create a PCDDocObject.
2. Specify the DST and library.
3. Specify the object type as a profile form.
4. Specify the document number of the profile using %OBJECT\_IDENTIFIER token.
5. Specify the tokens to relate or unrelate the documents.
6. Create a relation via the PCDDocObject.Update method.

Tokens are used to relate or unrelate two documents using the SetProperty() method with the “%RELATE\_DIRECTIVE” token as a property and the “%RELATE\_ITEMS” token as a property value relates two profiles together. Using the “%UNREALATE\_ITEMS” as a property value removes the association. Both concepts are demonstrated below:

public void RelateTwoDocuments(string library, string DST, string docNumber, string RelatedDocNum)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%RELATED\_ITEM", RelatedDocNum);

dmDoc.SetProperty("%RELATE\_DIRECTIVE", "%RELATE\_ITEMS");

try

{

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

public void UnrelateTwoDocuments(string library, string DST, string docNumber, string RelatedDocNum)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%RELATED\_ITEM", RelatedDocNum);

dmDoc.SetProperty("%RELATE\_DIRECTIVE", "%UNRELATE\_ITEMS");

try

{

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

It is possible to relate items in batches. The “%OBJECT\_IDENTIFIER token must be set to a value of “0” and specify a list of documents separated by comma using the “%ITEM\_LIST token.

When relating profiles in batches associations are created between profiles.

This is demonstrated below:

public void RelateDocumentsInBatches(string library, string DST)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", "0");

dmDoc.SetProperty("%RELATE\_DIRECTIVE", "%RELATE\_ITEMS");

dmDoc.SetProperty("%ITEM\_LIST", "15150, 15151, 15152,15153");

try

{

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

It is possible to relate a profile to another profile specified in a remote library. The remote library name must be specified with the document number when using the PCDDocObject.SetProperty method. For example:

dmDoc.SetProperty("%RELATED\_ITEM","MyRemoteLibrary:100”);

public void RelateRemoteLib(string library, string remoteLib, string DST, string docNumber, string RelatedDocNum)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%RELATED\_ITEM",remoteLib +":" + RelatedDocNum);

dmDoc.SetProperty("%RELATE\_DIRECTIVE", "%RELATE\_ITEMS");

try

{

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

## Searching for Related Documents

Searching for items related to a particular profile is not much different than searching for other items; it requires a special object-type to search the RELATED table and tokens to denote the scope of the search.

The conceptual steps needed when searching for related items are:

1. Create a PCDSearch object.
2. Specify the DST and Library.
3. Specify the object-type as “RelatedItemsSearch”.
4. Specify the return properties.
5. Specify the search criteria which includes the document number and scope (local, remote, or both).
6. Perform the search.
7. Iterate through the results.
8. Retrieve the properties.
9. Check for errors.

public void SearchRelatedProfiles(string library, string DST, string docNumber, string remoteLib)

{

PCDSearch dmSearch = new PCDSearch();

dmSearch.SetDST(DST);

dmSearch.AddSearchLib(library);

dmSearch.SetSearchObject("RelatedItemsSearch");

dmSearch.AddReturnProperty("DOCNUMBER");

dmSearch.AddReturnProperty("DOCNAME");

dmSearch.AddReturnProperty("%DOCS\_LIBRARY\_NAME");

dmSearch.AddSearchCriteria("%OBJECT\_IDENTIFIER", docNumber);

dmSearch.AddSearchCriteria("%GET\_RELATED\_ITEMS", "%GET\_ALL\_RELATED");

dmSearch.AddSearchCriteria("%RELATED\_REMOTE\_LIBS", remoteLib);

try

{

dmSearch.Execute();

if (dmSearch.ErrNumber != 0)

{

MessageBox.Show(dmSearch.ErrNumber + ": " + dmSearch.ErrDescription);

}

dmSearch.SetRow(0);

string msg = string.Empty;

for (int i = 0; i < dmSearch.GetRowsFound(); i++)

{

dmSearch.NextRow();

msg = msg + "DOCNUMBER: " + dmSearch.GetPropertyValue("DOCNUMBER") + "\n";

msg = msg + "DOCNAME: " + dmSearch.GetPropertyValue("DOCNAME") + "\n";

msg = msg + "LIBRARY: " + dmSearch.GetPropertyValue("%DOCS\_LIBRARY\_NAME") + "\n";

msg = msg + "\n";

}

MessageBox.Show(msg);

dmSearch.ReleaseResults();

}

catch(Exception ex)

{

MessageBox.Show(ex.Message);

}

}

To search for all related items the “**RelatedItemsSearch**” object-type must be specified.

The “%OBJECT\_IDENTIFIER” token specifies the document number to find related profiles.

Search criteria are specified using a token and the PCDSearch.AddSearchCriteria method. The %GET\_RELATED\_ITEMS token specifies the scope of the related profiles to find. It can take the following property values:

* %GET\_LOCAL\_RELATED: This token returns only related profiles in the current library.
* %GET\_REMOTE\_RELATED: This token returns only related profiles located in remote libraries.
* %GET\_ALL\_RELATED: This token returns all related profiles in the current and remote libraries.

Additionally, particular search libraries can be specified. The “%RELATED\_REMOTE\_LIBS” token specifies the libraries to search. By default, all related items from all libraries are returned if this token is not specified.

The only return properties that can be specified are:

* DOCNUMBER: The document number of the profile.
* %DOC\_LIBRARY\_NAME: The library that the document exists under.
* DOCNAME: The name of the document.

# Chapter 10 - Working with Folders

A folder is a container that holds references to documents and/or other subfolders. Folders are profiled objects, which means they are searchable, and can have security applied. When a folder is created new rows are inserted into the PROFILE and VERSIONS table. It is important to note that although an entry is created in the VERSIONS table, folders do not support versioning. This means there can be only one version of a folder. Subversions and attachments cannot be specified for a folder.

References contained within a folder can be in local or remote libraries and stored in the FOLDER\_ITEMS table Folders do not allow circular references. For example, folder A can contain folder B, but folder B cannot contain folder A.

## Creating a Folder

The steps for creating a folder are similar to creating a profile for a document with a few exceptions:

* The TYPE\_ID can be a value of FOLDER
* The APP\_ID must be a value of FOLDER

The conceptual steps for creating a folder are:

1. Create a PCDDocObject.
2. Specify the DST and Library.
3. Specify the object-type (Form).
4. Specify the Type \_ID (optionally), and APP\_ID as a value of “FOLDER”.
5. Specify any special directives using Tokens.
6. Create the folder.
7. Check for errors.

Similar to a document profile, the same fields must be included when creating a profile for a folder. This includes:

* DOCNAME
* FORM
* AUTHOUR
* TYPIST
* DOCUMENTTYPE
* APPLICATION

public void CreateFolder(string library, string DST, string folderName, string authorName)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("DOCNAME", folderName);

dmDoc.SetProperty("AUTHOR\_ID", authorName);

dmDoc.SetProperty("TYPE\_ID", "FOLDER");

dmDoc.SetProperty("CLIENT\_ID", "DEFAULT");

dmDoc.SetProperty("MATTER\_ID", "DEFAULT");

dmDoc.SetProperty("TYPIST\_ID", authorName);

dmDoc.SetProperty("STORAGE", "ARCHIVE");

dmDoc.SetProperty("APP\_ID", "FOLDER");

try

{

dmDoc.Create();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

## Adding an Item to a Folder

When adding an item to a folder, an entry is created in the FOLDER\_ITEM table using a PCDDocObject object, the **“ContentItem**” object-type is specified. There are a number of fields that can be specified when adding an item to a folder. The commonly used fields and Tokens are:

* **PARENT**: Specifies the document number of the folder
* **PARENT\_VERSION**: Specifies the Version Identifier of the folder. If not provided, DM will provide it automatically
* **DOCNUMBER**: The document number of the item to add to the folder
* **VERSION\_TYPE**: The reference type to the document. By default, it links to the most recent version. The following values may also be used:

S – The specific version

R - The most recent version

P - The most recently published version

* **VERSION**: The Version identifier is specified if the VERSION\_TYPE property has been set to reference a specific version.
* **DISPLAYNAME**: The display name of the item within a folder. If the DISPLAYNAME is not specified as a property, the PROFILE.DOCNAME value of the profile will be used instead.
* **%TARGET\_LIBRARY**: This token specifies the library of the folder.
* **%FOLDERITEM\_LIBRARY\_NAME:** This token specifies the library of the item being added to the folder. When items are added from remote libraries, the libraries must be configured to be remote to each other.

public void AddDocumentToFolder(string library, string DST, string FolderDocNum, string FolderVersionID, string DocToPutInFolder)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("ContentItem");

dmDoc.SetProperty("PARENT", FolderDocNum);

dmDoc.SetProperty("PARENT\_VERSION", FolderVersionID);

dmDoc.SetProperty("DOCNUMBER", DocToPutInFolder);

dmDoc.SetProperty("VERSION\_TYPE", "R");

dmDoc.SetProperty("DISPLAYNAME", "DOC ALIAS IN FOLDER");

try

{

dmDoc.Create();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

The item is added to the folder by calling the PCDDocObject.Create method. This causes a new entry in the FOLDER\_ITEM table. The items added to a folder will be added to the end of the list of items in the folder

## Retrieving Items from a Folder

The conceptual steps for retrieving items from a folder are:

1. Create a PCDPropertyLists object.
2. Specify the DST and Library.
3. Specify the object-type as “**ContentsCollection”**.
4. Specify the folder and folder version from which to receive items.
5. Retrieve the list of items from the folder.
6. Iterate through the list of results.

The steps to retrieve items from a specified folder involves searching the FOLDER\_ITEMS table to determine the items that belong to a specified folder.

A PCDPropertyLists object is used to search and retrieve all items within a folder. The “**ContentsCollection”** object-type is used to find all referenced items within a folder.

The PCDPropertyLists.SetProperty method is used to specify the PARENT and PARENT\_VERSION fields as properties. The PARENT property specifies the document number of the folder from which to retrieve items. The PARENT\_VERSION property specifies the version identifier of the folder.

The PCDPropertyLists.Execute method searches and populates the PCDPropertyLists object.

The **PCDPropertyLists** object should not be confused with the PCDPropertyList object. The PCDPropertyList object is a simple list of properties and values, but does not have any transaction methods; however, the PCDPropertyLists object combines the search mechanism of the PCDSearch object together with a PCDPropertyList object. The return values are a list of properties and values.

public void RetrieveItemsFromFolder(string library, string DST, string FolderDocNum, string FolderVerID)

{

PCDPropertyLists propLists = new PCDPropertyLists();

propLists.SetDST(DST);

propLists.SetObjectType("ContentsCollection");

propLists.SetProperty("%TARGET\_LIBRARY", library);

propLists.SetProperty("PARENT", FolderDocNum);

propLists.SetProperty("PARENT\_VERSION", FolderVerID);

try

{

propLists.Execute();

if (propLists.ErrNumber != 0)

{

MessageBox.Show(propLists.ErrNumber.ToString() +": " + propLists.ErrDescription);

}

string msg = string.Empty;

while (propLists.NextRow() ==1)

{

propLists.NextRow();

msg = msg + "DISPLAYNAME: " + propLists.GetPropertyValue("DISPLAYNAME") + "\n";

msg = msg + "SYSTEM\_ID: " + propLists.GetPropertyValue("SYSTEM\_ID") + "\n";

msg = msg + "DOC NUMBER: " + propLists.GetPropertyValue("DOCNUMBER") + "\n";

msg = msg + "NODE TYPE: " + propLists.GetPropertyValue("NODE\_TYPE") + "\n";

msg = msg + "\n";

}

MessageBox.Show(msg);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

## Updating an Item in a Folder

The conceptual steps used to update an item within a folder are:

1. Create a PCDDocObject object.
2. Specify the DST and library.
3. Specify the object-type as **“ContentItem”**.
4. Specify SYSTEM\_ID of the item within the folder.
5. Specify the properties of the item to update.
6. Make changes to the item.
7. Check for errors.

A PCDDocObject object can be used with the “**ContentItem”** object type to update an existing item within a specified folder. The FOLDER\_ITEM. SYSTEM\_ID column is required in place of the “%OBJECT\_IDENTIFIER”. This is because an item within a folder is stored in the FOLDER\_ITEM table, and not the PROFILE table.

To specify changes to the item in a folder, the PCDDocObject.SetProperty method can be used to modify item properties. The PCDDocObject.Update method is called to make the requested changes.

public void UpdateItemInFolder(string library, string DST, string ItemSysID, string NewDispalayName)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("ContentItem");

dmDoc.SetProperty("SYSTEM\_ID", ItemSysID);

dmDoc.SetProperty("DISPLAYNAME", NewDispalayName);

try

{

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

}

catch(Exception ex)

{

MessageBox.Show(ex.Message);

}

}

## Changing an Item’s Location in the Folder

When items are added to a folder, they are put at the end of the list. The PCDDocObject object and the “ContentItem” object-type can be used to change the location of an item within a folder.

public void ChangeItemLocationInFolder(string library, string DST, string ItemSysID)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("ContentItem");

dmDoc.SetProperty("%CONTENTS\_ITEM", ItemSysID);

dmDoc.SetProperty("%CONTENTS\_DIRECTIVE", "%CONTENTS\_MOVE\_DOWN");

try

{

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

To perform a move operation on an item the “%CONTENTS\_ITEM” token must be used with the SYSTEM\_ID of the item as the property value.

The “%CONTENTS\_DIRECTIVE” token is used to specify the move operation, such as moving up, moving down, to the top, or to the bottom of the list with the folder. The following tokens can be specified as property values for the “%CONTENTS\_DIRECTIVE” token:

* %CONTENTS\_MOVE\_UP: Moves an item up relative to its current position in the folder.
* %CONTENTS\_MOVE\_DOWN: Moves an item down relative to its current position within a folder.
* %CONTENTS\_MOVE\_TO\_TOP: Moves an item to the top of the list of items within a folder.
* %CONTENTS\_MOVE\_TO\_BOTTOM: Moves an item to the bottom of the list of items within a folder.

## Moving an Item after Another Item in a Folder

It is possible to move an item relative to another item in the list. To move an item in the list, use the “%CONTENTS\_DIRECTIVE” Token to specify the move directive. The “%CONTENTS\_ITEM” token specifies the item to move and the “%CONTENT\_AFTER\_ITEM” specifies the item to use as a reference point. Both items are specified using the SYSTEM\_ID of the items in the folder. The following is the property value that can be used for the “%CONTENTS\_DIRECTIVE” token:

* %CONTENTS\_MOVE\_AFTER: Moves an item after an item specified by the %CONTENT\_AFTER\_ITEM.

public void MoveItemRelativeToAnother(string library, string DST, string ItemSysID, string otherItemSysID)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("ContentItem");

dmDoc.SetProperty("%CONTENTS\_ITEM", ItemSysID);

dmDoc.SetProperty("%CONTENTS\_AFTER\_ITEM", otherItemSysID);

dmDoc.SetProperty("%CONTENTS\_DIRECTIVE", "%CONTENTS\_MOVE\_AFTER");

dmDoc.SetProperty("%CONTENTS\_DIRECTIVE", "%CONTENTS\_MOVE\_DOWN");

try

{

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

## Batch Reordering of Items

It is often required that more than one item be reordered. It is possible to specify a completely new order by specifying the order of items in a PCDPropertyList object and the “ContentsCollection” object type.

In the PCDPropertyList object the order is specified as the property and SYSTEM\_ID of the item in the folder to reorder as the property value. The SetProperty method and the following tokens are used to perform the batch reorder.

* %CONTENTS\_PARENT: Specifies the Folder to which to change the order.
* %CONTENTS\_PARENT\_VERSION: Specifies the version identifier for the folder to manipulate.
* %CONTENTS\_REORDER\_ARRAY: Specified with the PCDPropertyList object as a property value.
* %CONTENTS\_DIRECTIVE: Specified with the property value of the CONTENTS\_REORDER\_CONTENTS.

public void BatchReorderItems(string library, string DST, string FolderDocNum, string FolderVerID, List<string> SysIDList)

{

PCDPropertyList propList = new PCDPropertyList();

try

{

//Pass in a list of item SYSTEM\_ID values to build the PCDPropertyList

for (int i = 0; i < SysIDList.Count; i++)

{

int orderNum = i + 1;

propList.AddProperty(orderNum.ToString(), SysIDList[i]);

}

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("ContentsCollection");

dmDoc.SetProperty("%CONTENTS\_PARENT", FolderDocNum);

dmDoc.SetProperty("%CONTENTS\_PARENT\_VERSION", FolderVerID);

dmDoc.SetProperty("%CONTENTS\_REORDER\_ARRAY", propList);

dmDoc.SetProperty("%CONTENTS\_DIRECTIVE", "%CONTENTS\_REORDER\_CONTENTS");

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

## Copying Contents from One Folder to Another

It is possible to copy the contents of a folder to another existing folder. The folders can exist in the same library or a remote library. The following tokens are used to identify the source folder and destination folder.

* %CONTENTS\_SRC\_PARENT: This Token specifies the document number of the source folder.
* %CONTENTS\_SRC\_PARENT\_VERSION: This Token specifies the version identifier of the source folder.
* %CONTENTS\_SRC\_PARENT\_LIBRARY: This Token specifies the library of the source folder.
* %CONTENTS\_DST\_PARENT: This Token specifies the document number of the destination folder.
* %CONTENTS\_DST\_PARENT\_LIBRARY: This Token specifies the destination library. If a remote library, this should be the value of the identifier on the REMOTE\_LIBRARIES table.
* %CONTENTS\_DIRECTIVE: This Token specifies the Token to copy from one folder to another.
* %CONTENTS\_COPY\_CONTENTS: This Token specifies to copy the contents of the source folder into the destination folder.

public void CopyFolderToFolder(string library, string DST, string SRCFolderDocNum, string SRCFolderVerID, string DestFolderDocNum, string DestFolderVerID)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("ContentsCollection");

dmDoc.SetProperty("%CONTENTS\_SRC\_PARENT", SRCFolderDocNum);

dmDoc.SetProperty("%CONTENTS\_SRC\_PARENT\_VERSION", SRCFolderVerID);

dmDoc.SetProperty("%CONTENTS\_SRC\_PARENT\_LIBRARY", library);

dmDoc.SetProperty("%CONTENTS\_DST\_PARENT", DestFolderDocNum);

dmDoc.SetProperty("%CONTENTS\_DST\_PARENT\_VERSION", DestFolderVerID);

dmDoc.SetProperty("%CONTENTS\_DST\_PARENT\_LIBRARY", library);

dmDoc.SetProperty("%CONTENTS\_DIRECTIVE", "%CONTENTS\_COPY\_CONTENTS");

try

{

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

# Chapter 11 - Working with Profile Defaults and ACL Defaults

Defining Profile and Access Control List (ACL) defaults is useful in minimizing the number of items that a user needs to enter when creating a new profile in DM.

* Profile defaults are a template of properties and property values for a particular form.
* ACL defaults provide a default list of trustees for a particular profile.

Profile defaults are specified as application-specific defaults or a combination of the user and group defaults. Defaults for a user override the defaults for the primary group of the user.

Defaults are definable for a particular user, group, or application.

* Defaults for the primary from (default form) for a user are stored in the PROFILE\_DEFAULTS and ACL\_DEFAULTS columns in the PEOPLE table.
* Defaults for a group are stored in the PROFILE\_DEFAULTS and ACL\_DEFAULTS fields in the GROUP table.
* Defaults for a particular application for either a certain user or group are stored in the PROFILE\_DEFAULTS and ACL\_DEFAULTS columns in the FORM\_USER table.

Entries stored in the FORMS\_USER table pertain to a combination of an application, form, and the user or group.

Defaults that are stored in the PEOPLE table pertain to a specific user in DM. These profile defaults are based on fields available in the user’s primary form. The primary group that the user belongs to determine the user’s primary form.

## Creating Profile Defaults

The conceptual steps for creating profile defaults are:

1. Create a PCDDocObject Object.
2. Specify the DST and library.
3. Specify the object-type as “DocsForm”.
4. Specify the form using the “%FORM\_NAME” token, or the “%OBJECT\_IDENTIFIER” token with the SYSTEM\_ID of the form from the database.
5. Optionally, specify an application for the defaults.
6. Put the profile defaults into a PCDPropertyList object.
7. Specify the profile defaults using the %FORM\_PROFILE\_DEFAULTS token.
8. Create the profile defaults using the PCDDocObject.Update method.
9. Check for errors.

The following tokens are used to create or update profile defaults:

* %FORM\_NAME: When this token is used the value specified must be a form accessible to the user as either the primary form or as a form for the specified application. This means the form information must be set in Library Maintenance.
* %FORM\_APPLICATION: If this token is specified the defaults are created in the FORM\_USER table. If this token is not specified as a characterization property defaults are created in the PEOPLE table. Defaults are created for the user that is logged in.
* %FORM\_PROFILE\_DEFAULTS: This token is used to specify the profile defaults with a PCDPropertyList as a property value. If this token is not specified, the profile defaults will be removed.

The profile defaults are specified using the PCDPropertyList object. The list of defaults is returned as property and property value pairs to the PCDPropertyList object. The properties that are specified must exist on the form.

The profile defaults are specified using the %FORM\_PROFILE\_DEFAULTS token as a property and setting the PCDPropertyList object as a property value.

public void CreateProfileDefaults(string library, string DST, string FormName,string AppID,string Author, string DocType)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("DocsForm");

dmDoc.SetProperty("%FORM\_NAME", FormName);

//Optionally Set Application

if (AppID != "")

{

dmDoc.SetProperty("%FORM\_APPLICATION", AppID);

}

// Example: Set Author and Document Type. Must use fields on the form used

PCDPropertyList defaultsList = new PCDPropertyList();

defaultsList.AddProperty("AUTHOR\_ID", Author);

defaultsList.AddProperty("TYPE\_ID", DocType);

//If the %FORM\_PROFILE\_DEFAULTS token is not specified Profile defaults will be removed

dmDoc.SetProperty("%FORM\_PROFILE\_DEFAULTS", defaultsList);

try

{

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

## Creating ACL Defaults

The ACL defaults are created in the same manner as the profile defaults; however, when specifying default trustees in the ACL defaults, a separate token is used to specify each of the trustees being added. “TRUSTEE\_X” where X is the numeric value of the trustee.

The properties of each trustee is defined with the following tokens:

* %TRUSTEE\_ID: This token is used to specify the trustee name.
* %TRUSTEE\_TYPE: This token is used to specify if the trustee is a group (1) or user (2).
* %TRUSTEE\_RIGHTS: This token is used to specify the rights of the trustee.

The following steps are required to create ACL defaults:

1. Create a PCDDocObject Object.
2. Specify the DST and library.
3. Specify the object-type as “DocsForm”.
4. Specify the form using the “%FORM\_NAME” token, or the “%OBJECT\_IDENTIFIER” token with the SYSTEM\_ID of the form from the database.
5. Optionally, specify an application for the defaults.
6. The trustee fields are specified using a PCDPropertyList object. The properties are specified with the trustee name, trustee flag, and trustee rights. A new instance of a PCDPropertyList is required for each trustee being added.
7. Call the PCDDocObject.Update method.
8. Check for errors.

public void CreateACLDefaults(string library, string DST, string FormName, string AppID)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("DocsForm");

dmDoc.SetProperty("%FORM\_NAME", FormName);

//Optionally Set Application

if (AppID != "")

{

dmDoc.SetProperty("%FORM\_APPLICATION", AppID);

}

//A PCDPropertyList Object is declared for each trustee being added

PCDPropertyList pTrustee1 = new PCDPropertyList();

PCDPropertyList pTrustee2 = new PCDPropertyList();

//%TRUSTEE\_TYPE: This token is used to specify if the trustee is a group (1) or user (2).

pTrustee1.AddProperty("%TRUSTEE\_TYPE", "2");

pTrustee1.AddProperty("%TRUSTEE\_RIGHTS", "49");

pTrustee1.AddProperty("%TRUSTEE\_ID", "MSMITH");

pTrustee2.AddProperty("%TRUSTEE\_TYPE", "2");

pTrustee2.AddProperty("%TRUSTEE\_RIGHTS", "127");

pTrustee2.AddProperty("%TRUSTEE\_ID", "DJONES");

//If the %TRUSTEE\_X flag is not specified, or properties not set, ACL defaults will be //cleared.

dmDoc.SetProperty("%TRUSTEE\_0", pTrustee1);

dmDoc.SetProperty("%TRUSTEE\_1", pTrustee2);

try

{

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

If the “%FORM\_APPLICATION” token is specified, the defaults are created in the FORM\_USER table. If this token is not specified as a characterization property defaults are created in the PEOPLE table. Defaults are created for the user that is logged on.

It is important to remember that if profile defaults are not specified. They will be cleared. There are tokens available to inform DM not to clear profile or ACL defaults. These tokens are:

* %KEEP\_EXISTING\_PROFILE\_DEFAULTS: This token informs DM to keep the existing profile defaults when the property value of %YES is used.

dmDoc.SetProperty("%KEEP\_EXISTING\_PROFILE\_DEFAULTS", “%YES”);

* %KEEP\_EXISTING\_ACL\_DEFAULTS”: This token informs DM to keep the existing ACL defaults when the property value of %YES is used.

dmDoc.SetProperty("%KEEP\_EXISTING\_ACL\_DEFAULTS", “%YES”);

## Retrieving Profile Defaults

1. Create a PCDDocObject Object.
2. Specify the DST and library.
3. Specify the object-type as “DocsForm”.
4. Specify the form using the “%FORM\_NAME” token, or the “%OBJECT\_IDENTIFIER” token with the SYSTEM\_ID of the form from the database.
5. Optionally specify an application.
6. Call the PCDDocObject.Fetch method and check for errors.
7. Retrieve the profile defaults. They will be returned as a PCDPropertyList object.

The following describes the tokens used:

* %FORM\_NAME: This token is used to specify the name of the form to retrieve based on its FORM\_NAME value in the FORMS table. You may also use the “%OBJECT\_IDENTIFIER” token with the SYSTEM\_ID value on the FORMS table.
* %FORM\_APPLICATION: This token specifies name of the application. The defaults are only retrieved from the FORMS\_USER table.
* %FORM\_PROFILE\_DEFAULTS: This token is used to return the profile defaults to a PCDPropertyList as a property value.

public void RetrieveProfileDefaults(string library, string DST, string FormName, string AppID)

{

PCDPropertyList profDefList = new PCDPropertyList();

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("DocsForm");

dmDoc.SetProperty("%FORM\_NAME", FormName);

if (AppID != "")

{

dmDoc.SetProperty("%FORM\_APPLICATION", AppID);

}

try

{

dmDoc.Fetch();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + ": " + dmDoc.ErrDescription);

return;

}

profDefList = dmDoc.GetReturnProperty("%FORM\_PROFILE\_DEFAULTS");

profDefList.BeginIter();

string defaultsNamesAndValues = string.Empty;

for (int i = 0; i < profDefList.GetSize(); i++)

{

defaultsNamesAndValues = defaultsNamesAndValues + " " + profDefList.GetCurrentPropertyName().ToString();

defaultsNamesAndValues = defaultsNamesAndValues + ":" + " ";

defaultsNamesAndValues = defaultsNamesAndValues + " " + profDefList.GetCurrentPropertyValue().ToString();

defaultsNamesAndValues = defaultsNamesAndValues + " " + "\n";

profDefList.NextProperty();

}

MessageBox.Show(defaultsNamesAndValues);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

# Chapter 12 - Removing Items from DM

The DM API provides the functionality to remove documents, versions, folders, and quick searches. The delete process is a simple task. The most important steps are to correctly identify the object and use the correct methods. The PCDDocObject.Delete method is called to remove a profile, folder, or quick search; however, to remove a version, subversion, or attachment, the PCDDocObject.Update method is called. If the PCDDocObject.Delete method is called, instead of the PCDDocObject.Update to remove a version, subversion, or attachment, the profile would be deleted.

When an item from a folder is removed, it only deletes the reference to the document from the list of items within the folder.

The conceptual steps needed to remove an item from DM are:

1. Create a PCDDocObject object.
2. Specify the DST and library.
3. Specify the object-type or form. (This depends on the item being removed.)
4. Specify the property to identify the profile, version, quick search, or item in the folder.
5. Delete the item with the correct method.
6. Check for errors.

## Deleting a Document or Folder

The following sample code demonstrates how to delete a document. It is important to note that if an item is deleted there is no rollback functionality. If the item is referenced in a folder, all references must be deleted prior to deleting a document.

The property passed with the “%OBJECT\_IDENTIFIER” token is the document number value for the profile or folder.

The following tokens may be used as property values with the “%DELETE\_OPTION” token when deleting a document or folder.

* %DELETE\_ALL: This token deletes both the physical document from the document repository and the entry from the PROFILE table in the database.
* %DELETE\_PHYSICAL\_FILES: This token will remove the physical file from the document repository, but leave the entry in the PROFILE table.

If the “%DELETE\_OPTION” token is not specified, both the profile and physical document will be removed.

public void DeleteDocumentOrFolderExample(string library, string DST, string documentNumber)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetObjectType("DEF\_PROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", documentNumber);

dmDoc.SetProperty("%DELETE\_OPTION", "%DELETE\_ALL");

try

{

dmDoc.Delete();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + " " + dmDoc.ErrDescription.ToString());

}

}

catch (Exception e)

{

MessageBox.Show(e.ToString());

}

}

## Deleting a Version, Subversion, or Attachment

The “%OBJECT\_IDENTIFIER” token and the “%VERSION\_ID” token must be specified. The “%OBJECT\_IDENTIFIER” is the document number of the profile and the “%VERSION\_ID” is the VERSION\_ID field from the VERSIONS table.

The “%VERSION\_DIRECTIVE” token must be specified with the “%PCD\_DELETE\_VERSION” token as a property value.

The following tokens may be used as property values with the “%DELETE\_OPTION” token when deleting a version or subversion:

* %DELETE\_ALL: This token deletes both the physical document from the document repository and the entry from the PROFILE table in the database.
* %DELETE\_PHYSICAL\_FILES: This token will remove the physical file from the document repository, but leave the entry in the PROFILE table.

public void DeleteVersionOrSubVersion(string library, string DST, string docNumber, string versionID)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("DEF\_PROF");

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", docNumber);

dmDoc.SetProperty("%VERSION\_ID", versionID);

dmDoc.SetProperty("%VERSION\_DIRECTIVE", "%PCD\_DELETE\_VERSION");

try

{

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + " " + dmDoc.ErrDescription.ToString());

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.ToString());

}

}

## Deleting an Item from a Folder

When a reference is removed from a folder, it does not remove the actual item from DM. Only the reference within the folder is removed.

The value in the FOLDER\_ITEM.SYSTEM\_ID column is required in place of the “%OBJECT\_IDENTIFIER”. This is because an item within a folder is stored in the FOLDER\_ITEM table, and not the PROFILE table. This prevents the multiple references from being removed if the item is referenced in more than one folder.

public void RemoveItemFromFolder(string library, string DST, string FolderItemSysID )

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("ContentItem");

dmDoc.SetProperty("SYSTEM\_ID",FolderItemSysID);

try

{

dmDoc.Delete();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + " " + dmDoc.ErrDescription.ToString());

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.ToString());

}

}

## Deleting a Quick Search

The “%OBJECT\_IDENTIFIER” property value used for deleting a quick search is the QUICKSEARCH.SYSTEM\_ID value.

public void DeleteQuickSearch(string library, string DST, string QSSearchID)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("cyd\_cmnquicksearch");

dmDoc.SetProperty("%OBJECT\_IDENTIFIER", QSSearchID);

try

{

dmDoc.Delete();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + " " + dmDoc.ErrDescription.ToString());

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.ToString());

}

}

## Deleting a Public Folder (Root Object)

The value in the ROOT\_OBJECT.SYSTEM\_ID column is required in place of the “%OBJECT\_IDENTIFIER”.

public void DeletePublicFolder(string library, string DST, string RootObjectSysID)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("RootObjects");

dmDoc.SetProperty("SYSTEM\_ID", RootObjectSysID);

try

{

dmDoc.Delete();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + " " + dmDoc.ErrDescription.ToString());

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.ToString());

}

}

# Chapter 13 - The PCDSQL Object

The DM API provides most of the functionality needed to perform basic tasks and customizations. However, it is still possible that there is functionality that is required, but not provided by the DM API. The PCDSQL object provides direct access to the DM library by executing SQL statements directly against the database.

This object provides flexibility, but it is recommended this object be used for reading and modifying custom tables, or when no other DM API object provides the desired functionality. The PCDSQL object uses a DM Security Token (DST) to provide security. It is highly recommended that the client-tier application use this object internally, and not give end-users the ability to execute SQL commands.

In order to use the PCDSQL object, SQL Passthrough must be enabled in the System Parameters of the DM Library. However, for additional security an “Enhanced DST” that includes a license key can be used, which allows the PCDSQL to execute statements against the library regardless if SQL Passthrough is enabled or not. Please contact eDOCS Support to obtain the information needed for using an enhanced DST.

The following code sample is a simple demonstration of using the PCDSQL object. The query must be passed in as a string value with the correct SQL syntax.

public List<string>PCDSQLExample(string library, string DST)

{

List<string> Groups = new List<string>();

PCDSQL dmSQL = new PCDSQL();

dmSQL.SetDST(DST);

dmSQL.SetLibrary(library);

string strSQL = "SELECT GROUP\_ID FROM DOCSADM.GROUPS";

try

{

dmSQL.Execute(strSQL);

if (dmSQL.ErrNumber != 0)

{

MessageBox.Show(dmSQL.ErrNumber.ToString() + " " + dmSQL.ErrDescription.ToString());

}

int numrows = dmSQL.GetRowCount();

if (numrows > 0)

{

for (int i = 0; i < numrows; ++i)

{

Groups.Add(dmSQL.GetColumnValue(1).ToString());

dmSQL.NextRow();

}

dmSQL.ReleaseResults();

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

return Groups;

}

Like other objects in the DM API, the PCDSQL object methods can be broken down into four distinct types; Characterization, Transaction, Retrieval, and Other.

## PCDSQL Characterization Methods

**int** **SetDST**(**string** *zDST*)

This method specifies the DST (Document Security Token) for the object to initiate a transaction with the DM Server. The method requires one parameter.

* *zDST*: Specifies the DM Security Token(DST) to be used. The DST must be retrieved using the PCDLogin object

**int** **SetLibrary**(**string** *zLibrary*)

This method specifies which library to execute a SQL statement against

* *zLibrary*: Specifies the name of the library to search.

## PCDSQL Transaction Method

**int** **Execute**(**string** *sql*)

This method executes the specified SQL statement.

* *sql:* The text of the SQL statement

**PCDSQL Retrieval Methods:**

**int** **GetColumnCount**()

This method returns the number of columns in the result set. It references the current results set within the object.

**dynamic** **GetColumnName**(**int** *lColNdx*)

This method returns the specified column name in the current result set.

* *lColNdx*: An integer value that corresponds to the column number.

**dynamic** **GetColumnValue**(**int** *lColNdx*)

This method returns the value within the specified column in the current SQL result set.

* *lColNdx*: An integer value that corresponds to the column number.

**int** **GetDBVendor**()

This method retrieves the numeric identifier associated with the SQL database vendor.

The return value represents the following:

* 3 – Microsoft SQL Server
* 5 – Oracle Database Server

**int** **GetNextKey**(**string** *zSequenceName*)

This method returns the next generated numeric value of the sequence.

* *zSequenceName:* The name of the sequence being generated. Example: generating a new SYSTEM\_ID value.

**int** **GetRowsAffected**()

This method returns the number of rows in the SQL statement that were affected by a DELETE, INSERT, or UPDATE command.

**int** **GetRowCount**()

This method returns the number of rows that are returned as part of a SELECT statement.

**int** **GetSQLErrorCode**()

This method retrieves the error code returned by the database. This is different than the error code generated by the PCDSQL object.

## Other Methods

**int** **NextRow**()

This method iterates to the next row of the result set.

**int** **SetRow**(**int** *ulRowNbr*)

This method sets the internal pointer to a specified row in the result set.

* *ulRowNbr:* The location in the result set to which to set the pointer.

**int** **ReleaseResults**()

This method releases all the results. It is important this method be called as soon as possible to release the resources associated with the SQL connection pools on the DM Server.

# Chapter 14 - Working with Activity Log Entries

Activity log entries are records that track changes which have been made to profiles in DM.

This information is stored in the ACTIVITYLOG. ACTIVITY\_TYPE column. It is possible to add custom activity log entries.

## Retrieving Activity Log Entries

The conceptual steps needed to return activity log entries are:

1. Create a PCDSearch Object.
2. Specify the DST and library.
3. Specify the object-type with the ACTIVITYLOG table defined as the primary table with the PCDSearch.SetSearchObject method The object-types provide by eDOCS DM are:

* cyd\_activitylog
* v\_activitylog

1. Specify the return properties to be included in the list of results.
2. Specify the search criteria.
3. Perform the search and check for errors.
4. Iterate through the results.
5. Release the results.

public void ReturnActivityLogEntriesForDoc(string library, string DST, string docNumber)

{

PCDSearch dmSearch = new PCDSearch();

dmSearch.SetDST(DST);

dmSearch.AddSearchLib(library);

dmSearch.SetSearchObject("cyd\_activitylog");

dmSearch.AddSearchCriteria("DOCNUMBER",docNumber);

dmSearch.AddReturnProperty("START\_DATE");

dmSearch.AddReturnProperty("START\_TIME");

dmSearch.AddReturnProperty("TYPE");

dmSearch.AddReturnProperty("TYPIST\_ID");

try

{

dmSearch.Execute();

if (dmSearch.ErrNumber != 0)

{

MessageBox.Show(dmSearch.ErrNumber.ToString() + " " +dmSearch.ErrDescription.ToString());

return;

}

string msg = " ";

for (int i = 0; i < dmSearch.GetRowsFound(); i++)

{

dmSearch.NextRow();

msg = msg + "ACTIVITY TYPE: " + dmSearch.GetPropertyValue("TYPE") + "\n";

msg = msg + "TYPIST: " + dmSearch.GetPropertyValue("TYPIST\_ID") + "\n";

msg = msg + "START\_DATE: " + dmSearch.GetPropertyValue("START\_DATE") + "\n";

msg = msg + "START\_TIME " + dmSearch.GetPropertyValue("START\_TIME") + "\n";

msg = msg + "\n";

}

MessageBox.Show(msg);

dmSearch.ReleaseResults();

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

## Create a Custom Activity Log Entry

When creating activity log entries, the following fields are required:

* DOCNUMBER: Specifies the document number.
* AUTHOR\_ID: Specifies the Author.
* TYPIST\_ID: Specifies the Typist.
* TYPE: The numeric value of the activity type.

When specifying a custom Type value, the developer must use a value not reserved by DM. Consult the values listed for the ACTIVITYLOG.ACTIVITY\_TYPE values in the *eDOCS DM/RM Data Dictionary*.

public void AddCustomActivityLogEntry(string library, string DST, string docNumber, string authorID, string typistID)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("cyd\_activitylog");

dmDoc.SetProperty("DOCNUMBER", docNumber);

dmDoc.SetProperty("DESCRIPTION", "MY CUSTOM DESCRIPTION");

dmDoc.SetProperty("TYPE", "100");

dmDoc.SetProperty("AUTHOR\_ID", authorID);

dmDoc.SetProperty("TYPIST\_ID", typistID);

try

{

dmDoc.Create();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + " " + dmDoc.ErrDescription.ToString());

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

# Chapter 15 - Working with Public Folders

Public Folders appear under the Public Folders node in the hierarchical tree in both DM Extensions and DM Webtop, and are references to profiled folders. In order to make a profiled folder public, an entry must be created in the ROOT\_OBJECT table. Important concepts to remember about public folders:

* Making a profiled folder public requires creating an entry in the ROOT\_OBJECT table.
* Security cannot not be set on public folders; they will have the same security as the profiled folder being referenced.
* Developers should only reference profiles representing folders.
* Public folders can have different display names than the referenced folder name.

## Creating a Public Folder

The conceptual steps used to create a public folder are:

1. Create a PCDDocObject object.
2. Specify the DST and library.
3. Specify the object-type as “RootObject”.
4. Specify the document number of the referenced folder.
5. Optionally, specify the display name of the public folder.
6. Optionally specify if the public folder only available to a certain user with the %ROOTOBJECTS\_USER\_NAME.
7. Create the public folder.
8. Check for errors.

A user must have full access privileges to the referenced folder to create a public folder unless they are a member of the DOCS\_SUPERVISOR group. The PCDDoc.SetProperty method is used to specify the following characteristics of the public folder.

* DOCNUMBER: The document number of the profiled folder that will be referenced.
* DISPLAYNAME: The display name of the public folder. If the display name is not specified. the name of the profile will be used instead.
* %ROOTOBJECTS\_USER\_NAME: This token specifies the public folder is only available for a certain user. A value of 0 is inserted in to the ROOT\_OBJECT. USER\_ID column if not specified. This will make the public folder visible to all users.

public void MakeExistingFolderPublic(string library, string DST, string docNumber, string displayName)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("RootObjects");

dmDoc.SetProperty("DOCNUMBER", docNumber);

dmDoc.SetProperty("DISPLAYNAME", displayName);

try

{

dmDoc.Create();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + " " + dmDoc.ErrDescription.ToString());

return;

}

}

catch(Exception ex)

{

MessageBox.Show(ex.Message);

}

}

This code sample will create a new folder and set it as public:

public void CreateNewPublicFolder(string library, string DST, string folderName, string authorName, string displayName)

{

PCDDocObject dmDoc = new PCDDocObject();

string docNumber = string.Empty;

dmDoc.SetDST(DST);

dmDoc.SetObjectType("LAWPROF");

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetProperty("DOCNAME", folderName);

dmDoc.SetProperty("AUTHOR\_ID", authorName);

dmDoc.SetProperty("TYPE\_ID", "FOLDER");

dmDoc.SetProperty("CLIENT\_ID", "DEFAULT");

dmDoc.SetProperty("MATTER\_ID", "DEFAULT");

dmDoc.SetProperty("TYPIST\_ID", authorName);

dmDoc.SetProperty("STORAGE", "ARCHIVE");

dmDoc.SetProperty("APP\_ID", "folder");

try

{

dmDoc.Create();

if (dmDoc.ErrNumber == 0)

{

docNumber = dmDoc.GetReturnProperty("%OBJECT\_IDENTIFIER").ToString();

}

else

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + " " + dmDoc.ErrDescription.ToString());

return;

}

PCDDocObject dmDoc2 = new PCDDocObject();

dmDoc2.SetDST(DST);

dmDoc2.SetObjectType("RootObjects");

dmDoc2.SetProperty("DOCNUMBER", docNumber);

dmDoc2.SetProperty("DISPLAYNAME", displayName);

dmDoc2.Create();

if (dmDoc2.ErrNumber != 0)

{

MessageBox.Show(dmDoc2.ErrNumber.ToString() + " " + dmDoc2.ErrDescription.ToString());

return;

}

}

catch(Exception ex)

{

MessageBox.Show(ex.Message);

}

}

## Modifying a Public Folder

Previously when making changes to an object such as a profile or quick search the %OBJECT\_IDENTIFIER token was used to identify the item. However, %OBJECT\_IDENTIFIER token is NOT used when specifying “RootObject as the object-type. The unique identifier used for a Public folder is the Root\_Object.System\_ID value. The PCDDocObject.Update method is used to modify existing items in the ROOT\_OBJECT table.

This sample will change the display name value of the public folder:

public void ModifyPublicFolder(string library, string DST, string SysID)

{

PCDDocObject dmDoc = new PCDDocObject();

dmDoc.SetDST(DST);

dmDoc.SetProperty("%TARGET\_LIBRARY", library);

dmDoc.SetObjectType("RootObjects");

dmDoc.SetProperty("SYSTEM\_ID", SysID);

//Change Display Name

dmDoc.SetProperty("DISPLAYNAME", "NEW DISPLAY NAME");

try

{

dmDoc.Update();

if (dmDoc.ErrNumber != 0)

{

MessageBox.Show(dmDoc.ErrNumber.ToString() + " " + dmDoc.ErrDescription.ToString());

return;

}

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

}

}

## Retrieving a List of Public Folders

Steps needed for retrieving a list of Public Folders:

1. Create a PCDPropertyLists object.
2. Specify the DST and library.
3. Specify any search criteria for searching for public folders.
4. Specify the object-type as “RootsObjectCollection”.
5. Retrieve the results.
6. Iterate through the results.
7. Retrieve the values.

public void ReturnPublicFolders(string library, string DST)

{

PCDPropertyLists pLists = new PCDPropertyLists();

pLists.SetDST(DST);

pLists.SetProperty("%TARGET\_LIBRARY", library);

pLists.SetObjectType("RootObjectsCollection");

pLists.SetProperty("USER\_ID", "0");

try

{

pLists.Execute();

if (pLists.ErrNumber != 0)

{

MessageBox.Show(pLists.ErrNumber.ToString() + " " + pLists.ErrDescription.ToString());

return;

}

string msg = string.Empty;

while (pLists.NextRow() ==1)

{

pLists.NextRow();

msg = msg + "SYSTEM\_ID: " + pLists.GetPropertyValue("SYSTEM\_ID") + "\n";

msg = msg + "DOCNUMBER: " + pLists.GetPropertyValue("DOCNUMBER") + "\n";

msg = msg + "DISPLAY NAME: " + pLists.GetPropertyValue("DISPLAYNAME") + "\n";

msg = msg + "\n";

}

MessageBox.Show(msg);

}

catch(Exception ex)

{

MessageBox.Show(ex.Message);

}

}