opentext™

OpenText™ Documentum™ Content Management

Server Fundamentals Guide

Understand features such as content management services, security services, virtual documents, workflows, and lifecycles.

EDCCS250400-GGD-EN-01

OpenText™ Documentum™ Content Management Server Fundamentals Guide

EDCCS250400-GGD-EN-01
Rev.: 2025-Oct-17

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

Open Text Corporation

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

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

Table of Contents

Chapter 1

Overview

This guide provides an overview of OpenText Documentum Content Management (CM) Server and then discusses the basic features of Documentum CM Server in detail. This guide is intended for system and repository administrators, application programmers, and any other users to obtain a basic understanding of the services and behavior of Documentum CM Server. The guide assumes the reader has an understanding of relational database management system (RDBMS), object-oriented programming, and Structured Query Language (SQL).

This chapter provides an introduction to content management and the features of Documentum CM Server.

1.1 Managed content

Content is information stored as computer data files. It can include word processing, spreadsheet, graphics, video and audio files.

Most content is stored locally on personal computers, organized arbitrarily, and only available to a single user. This means that valuable data is subject to loss, and projects are subject to delay when people cannot get the information they need.

The best way to protect these important assets is to move them to a centralized content management system.

1.1.1 Elements of the content management system

Documentum CM Server manages content in a repository. The repository has three primary elements: a Documentum CM Server, a relational database, and a place to store files.

Everything in a repository is stored as an object. The content file associated with an object is typically stored in a file system. An object has associated metadata. For example, the metadata can be file name, storage location, creation date, and so on. The metadata for each object is stored as a record in a relational database.

For more information about the description of the repository data model, see “Data model“ on page 43.

A data dictionary describes each of the object types in the OpenText™ Documentum™ Content Management system. You can create custom applications that query this information to automate processes and enforce business rules. For more information about what information is available and how it might be used in your OpenText Documentum Content Management (CM) implementation, see “Data dictionary” on page 58.

Documentum CM Server provides the connection to the outside world. When content is added to the repository, Documentum CM Server parses the object metadata, automatically generates additional information about the object, and puts a copy of the content file into the file store. When stored as an object in the repository, there are many ways that users can access and interact with the content.

1.1.1.1 Check out and check in

Content in the repository can be checked out, making it available for edit by one user while preventing other users from making changes. When the edits are complete, the user checks the content back in to the repository. The changes are then visible to other users, who can check out and update the content as needed.

For more information about access control features of Documentum CM Server, see “Concurrent access control” on page 125.

1.1.1.2 Versioning

Documentum CM Server maintains information about each version of a content object as it is checked out and checked in to the repository. At any time, users can access earlier versions of the content object to retrieve sections that have been removed or branch to create a new content object.

For more information about how versions are handled by Documentum CM Server, see “Versioning” on page 117.

1.1.1.3 Virtual documents

Virtual documents are a way to link individual content objects into one larger document.

A content object can belong to multiple virtual documents. When you change the individual content object, the change appears in every virtual document that contains that object.

You can assemble and publish all or part of a virtual document. You can integrate the assembly and publishing services with popular commercial applications such as Arbortext Editor. Assembly can be controlled dynamically with business rules and data stored in the repository.

For more information, see “Virtual documents“ on page 151.

1.1.1.4 Full-text indexing

Documentum CM Server supports the Documentum xPlore index server, which provides comprehensive indexing and search capabilities. By default, all property values and indexable content are indexed, allowing users to search for documents or other objects. The *OpenText Documentum xPlore* documentation describes installation, administration, and customization of the xPlore indexing server.

1.1.1.5 Security

Documentum CM Server provides security features to control access and automate accountability.

1.1.1.5.1 Repository security

Content in the repository is protected on two levels:

- At the repository level

  Documentum CM Server supports several authentication methods. When users attempt to connect to the Documentum CM Server, the system validates their credentials. If invalid, the connection is not allowed.

  Documentum CM Server also provides five levels of user privilege, three extended user privileges, folder security, privileged roles, and basic support for client-application roles and application-controlled objects.

- At the object level

  Documentum CM Server uses a security model based on Access Control Lists (ACLs) to protect repository objects.

  In the ACL model, every content object has an associated ACL. The entries in the ACL define object-level permissions that apply to the object. Object-level permissions are granted to individual users and to groups. The permissions control which users and groups can access the object, and what operations they can perform. There are seven levels of base object-level permissions and five extended object-level permissions.

For information about security options, see “Security services“ on page 77. For information about user administration and working with ACLs, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

1.1.1.5.2 Accountability

Documentum CM Server provides auditing and tracing facilities. Auditing keeps track of specified operations and stores a record for each in the repository. Tracing provides a record that you can use to troubleshoot problems when they occur.

Documentum CM Server also supports electronic signatures. In custom applications, you can require users to sign off on a document before passing the document to the next activity in a workflow, or before moving the document forward in its lifecycle. Sign-off information is stored in the repository.

For more information about auditing and tracing facilities, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

1.2 Process management features

The process management features of Documentum CM Server enforce business rules and policies when users create and manipulate content. The primary process management features of Documentum CM Server are workflows and lifecycles.

1.2.1 Workflows

The Documentum CM Server workflow model lets you develop process and event-oriented applications for content management. The model supports both the manual and automatic workflows.

You can define workflows for individual documents, folders containing a group of documents, and virtual documents. A workflow definition can include simple or complex task sequences, including sequences with dependencies. Workflow and event notifications are automatically issued through standard electronic mail systems, while content remains under secure server control. Workflow definitions are stored in the repository, allowing you to start multiple workflows based on one workflow definition.

Workflows are created and managed using OpenText™ Documentum™ Content Management Workflow Designer. The *OpenText Documentum Content Management Workflow Designer* documentation contains detailed information.

For information about the object types that support workflows, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.

1.2.2 Lifecycles

Many documents within an enterprise have a recognizable lifecycle. A document is created using a defined process of authoring and review, used, and then either superseded or discarded.

Documentum CM Server life cycle management services let you automate the stages of document life. The stages in a lifecycle are defined in a policy object stored in the repository. For each stage, you can define prerequisites to be met and actions to be performed before an object can move into that particular stage.

For information about how lifecycles are implemented, see “Lifecycles“ on page 209. For information about the object types that support lifecycles, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.

1.3 Distributed services

A OpenText Documentum CM system installation can have multiple repositories. Documentum CM Server provides built-in, automatic support for a variety of configurations. For more information about complete description of the features supporting distributed services, see *OpenText Documentum Content Management - Server and Server Extensions Installation Guide (EDCSY250400-IGD)*.

1.4 OpenText Documentum CM features

1.4.1 OpenText™ Documentum™ Content Management Trusted Content Services

OpenText Documentum Content Management (CM) Trusted Content Services add enhanced security features to Documentum CM Server. The features are:

- Digital shredding of content files
- Strong electronic signatures
- Ability to encrypt content in file store storage areas

The following sections contain detailed information:

- “Trusted Content Services security features” on page 79
- “Encrypted file store storage areas” on page 107
- “Digital shredding” on page 108
- “Signature requirement support” on page 97
- “ACLs” on page 91

For more information about ACLs and to add, modify, and remove entries, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

1.4.2 Content Services for EMC Centera

The Content Services for EMC Centera (CSEC) provides support for Centera storage hosts. You can use content-addressed storage areas, the repository representation of a Centera storage host. These storage areas are particularly useful if the repository is storing large amounts of relatively static data that must be kept for a specified interval. CSEC provides content storage with guaranteed retention and immutability.

Note: It is possible to apply retention to content without CSEC if you have OpenText™ Documentum™ Content Management Retention Policy Services.

For more information on CSEC, see "Document retention and deletion" on page 127 and "Setting content properties and metadata for content-addressed storage" on page 134.

For more information about content-addressed storage areas, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

1.4.3 Content Storage Services

The Content Storage Services allows you to create and use content storage and migration policies. These policies automate the assignment of content to storage areas, eliminating manual, error-prone processes and ensuring compliance with company policy with regard to content storage. Storage polices also automate the movement of content from one storage area to another, thereby enabling policy-based information lifecycle management.

The Content Storage Services also enables the content compression and content duplication checking and prevention features. Content compression is an optional configuration choice for file store and content-addressed storage areas. Content duplication checking and compression is an optional configuration choice for file store storage areas.

For more information about Content Storage Services, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

1.4.4 OpenText™ Documentum™ Content Management XML Store and XQuery

The OpenText Documentum Content Management (CM) XML Store is a highly scalable, native XML repository for Documentum CM Server. XML Store adds standards-based XQuery to the XML capabilities of Documentum CM Server. A native XML content store or repository stores persistent XML content as-is, without mapping the XML to database rows and columns. The XML structure is preserved, allowing users to query content at any level of detail (for example, individual elements, attributes, content objects, or metadata attributes), even on very large information sets. As a native XML repository, XML Store provides performance advantages over relational databases and file systems through specialized XML indexing methods, caching, and architecture optimized for XML.

For more information about XML Store and XQuery, see *OpenText Documentum Content Management - Server and Server Extensions Installation Guide (EDCSY250400-IGD)* and *OpenText Documentum Content Management XML Store - Administration Guide (EDCCFMTXML250400-AGD)*.

1.4.5 OpenText™ Documentum™ Content Management Retention Policy Services

The OpenText Documentum Content Management (CM) Retention Policy Services requires that you install the Retention Policy Services Documentum Archive (DAR) file. A DAR file is the executable, binary version of a Documentum Composer project. DAR files are typically used to distribute applications.

Retention Policy Services automates the retention and disposition of content in compliance with regulations, legal requirements, and best practice guidelines.

The product allows you to manage a content retention in the repository through a retention policy: a defined set of phases, with a formal disposition phase at the end. You access Retention Policy Services through Documentum Administrator.

Retention Policy Services policies are created and managed using Retention Policy Services Administrator, an administration tool that is similar to, but separate from, Documentum Administrator.

For information about the various ways to implement document retention, including retention policies, and how those policies affect behaviors, see “Document retention and deletion” on page 127.

For more information about applying a retention policy to a virtual document affects that document, see “Virtual documents and retention policies” on page 156.

For more information about using Retention Policy Services Administrator, see *OpenText Documentum Content Management - Records Client User Guide (EDCRM250400-UGD)*.

1.4.6 Documentum Collaborative Services

Documentum Collaborative Services allow teams to securely work with content as a group from any Documentum Web Development Kit based client. The following collaborative features are supported:

- Rooms: secure areas within a repository with a defined membership. Rooms provide a secure virtual workplace, allowing members to restrict access to content in the room to the room membership.
- Discussions: online comment threads that enable informal or spontaneous collaboration.
- Contextual folders: folders that allow users to add descriptions and discussions. Users capture and express the business-oriented context of a folder hierarchy.
- Notes: simple documents that have built-in discussions and can contain rich text content. Using notes avoids the overhead of running a separate application for text-based collaboration.

The *OpenText Documentum Webtop* documentation contains additional information about Documentum Collaborative Services.

1.5 Internationalization

Internationalization refers to the ability of Documentum CM Server to handle communications and data transfer between itself and various client applications independent of the character encoding they use.

Documentum CM Server runs internally with the UTF-8 encoding of Unicode. The Unicode standard provides a unique number to identify every letter, number, symbol, and character in every language.

Documentum CM Server uses Unicode to:

- Store metadata using non-English characters
- Store metadata in multiple languages
- Manage multilingual web and enterprise content

The Unicode Consortium website contains more information about Unicode, UTF-8, and national character sets. For more information about the summary of Documentum CM Server internationalization requirements, see “Internationalization summary“ on page 237.

1.6 Communicating with Documentum CM Server

The OpenText Documentum CM system provides a full suite of products to give users access to Documentum CM Server.

1.6.1 Applications

The OpenText Documentum CM system provides web-based and desktop client applications.

You can also write your own custom applications. Documentum CM Server supports all the OpenText Documentum CM Application Programming Interfaces (APIs). The primary API is the OpenText™ Documentum™ Content Management Foundation Java API. This API is a set of Java classes and interfaces that provides full access to Documentum CM Server features. Applications written in Java, Visual Basic (through OLE COM), C++ (through OLE COM), and Docbasic can use OpenText Documentum Content Management (CM) Foundation Java API. Docbasic is the proprietary programming language Documentum CM Server uses.

For ease of development, the OpenText Documentum CM system provides a web-based and a desktop development environment. You can develop custom applications and deploy them on the web or desktop. You can also customize components of the OpenText Documentum CM client applications.

1.6.2 Interactive utilities

Documentum Administrator is a web-based tool that lets you perform administrative tasks for a single installation or distributed enterprise from one location.

The Interactive Documentum Query Language (IDQL) utility in Documentum Administrator lets you execute Documentum Query Language (DQL) statements directly. The utility is primarily useful as a testing arena for statements that you want to add to an application. It is also useful when you want to execute a quick query against the repository.

For more information about Documentum Administrator and IDQL, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

1.7 Business Workspace

1.7.1 Overview

A Business Workspace is a specialized folder designed to support enterprise information management and collaboration around a business object. It acts as a synchronized, structured entity that connects data, users, roles, and permissions relevant to a business process.

Key components include:

- Business Object: A meaningful entity in an organization such as a customer, supplier, material, and others that structures a business process.
- Templates: Predefined configurations that guide workspace setup and usage.
- Hierarchy: An organizational structure that defines relationships and access levels.

Capabilities include:

- Business Information: Securely store, organize, and manage business documents in a centralized repository.
- Business Users: Define actual users, their roles, and permissions within the workspace and associated objects.

The following object types and an aspect are created:

- `dm_bws_type_def`
- `dm_bws_template`
- `dm_bws`
- `dm_bws_aspect`

For more information about the object types and aspect, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.

For more information about operations in Business Workspace, see the following documentation:

- *OpenText Documentum Content Management - Workspaces Admin Console Guide (EDCADC250400-ABW)*
- *OpenText Documentum Content Management - Smart View User Guide (EDCCL250400-UGD)*

1.7.2 Business Workspace types

A Business Workspace type is a container object that defines the foundational structure and configuration for Business Workspace. It acts as an umbrella under which all workspace-related artifacts are organized and managed.

Users must have the `dm_bws_admin` or `dm_bws_admin_dynamic` role and the `Create Type` user privilege to create, update, and delete the Business Workspace type definition objects.

Key functions include:

- Container for Artifacts: Includes attributes, roles, permission templates, and Business Workspace templates.
- Structural Definition: Establishes the hierarchy and rules that govern how Business Workspace operate.

Key purposes include:

- Standardize Business Workspace creation and behavior across business processes.
- Ensure consistency in how business objects are managed and collaborated on.

1.7.3 Business Workspace templates

A Business Workspace template defines the folder structure and document type rules for a Business Workspace. It is applied when a Business Workspace instance is created from a specific Business Workspace type.

Users must have the `dm_bws_admin` or `dm_bws_admin_dynamic` role to create, update, and delete the Business Workspace template objects.

Key features include:

- Folder Structure: Creates a predefined layout of folders.
- Document Type Control:
  - Specifies allowed document types per folder.
  - Supports creation of defined or inherited types.
  - Supports definition of a document type in multiple folders.

1.7.4 Business Workspace instances

A Business Workspace instance is the operational workspace created from a Business Workspace type and a Business Workspace template. It serves as the collaborative environment for managing business objects and related content.

Any user can create a Business Workspace instance. If groups are defined in the template, then the users must have the `Create Group` user privilege.

The creation of a Business Workspace instance creates the Business Workspace, its folder structure and the defined roles. The roles have to be filled with users or groups which should have access to the Business Workspace and its folders and documents.

Chapter 2

Session and transaction management

2.1 Session overview

A session is a client connection to a repository. Repository sessions are opened when users or applications establish a connection to a Documentum CM Server.

Users or applications can have multiple sessions open at the same time with one or more repositories. The number of sessions that can be established for a given user or application is controlled by the `dfc.session.max_count` entry in the `dfc.properties` file. The value of this entry is set to 1000 by default and can be reset.

For a web application, all sessions started by the application are counted towards the maximum.

Note: If the client application is running on a Linux operating system, the maximum number of sessions possible is also limited by the number of descriptors set in the Linux kernel.

2.2 Session implementation in Foundation Java API

This section describes how sessions are implemented within Foundation Java API. Foundation Java API is the published and supported programming interface for accessing the OpenText Documentum CM functionality.

In Foundation Java API, sessions are objects that implement a session, commonly the `IDfSession` interface. Each session object gives a particular user access to a particular repository and the objects in that repository.

2.2.1 Obtaining a session

Typically, sessions are obtained from a session manager. A session manager is an object that implements the `IDfSessionManager` interface. Session manager objects are obtained by calling the `newSessionManager` method of the `IDfClient` interface. Obtaining sessions from the session manager is the recommended way to obtain a session. This is especially true in web applications because the enhanced resource management features provided by a session manager are most useful in web applications.

By default, if an attempt to obtain a session fails, Foundation Java API automatically tries again. If the second attempt fails, Foundation Java API tries to connect to another server if another is available. If no other server is available, the client application receives an error message. You can configure the time interval between connection attempts and the number of retries.

Each session has a session identifier in the format S*n* where *n* is an integer equal to or greater than zero. This identifier is used in trace file entries, to identify the session to which a particular entry applies. Session identifiers are not used or accepted in Foundation Java API method calls.

2.2.2 Shared and private sessions

Repository sessions are either shared or private.

Shared sessions can be used by more than one thread in an application. In web applications, shared sessions are particularly useful because they allow multiple components of the application to communicate. For example, a value entered in one frame can affect a setting or field in another frame. Shared sessions also make the most efficient use of resources.

Shared sessions are obtained by using an `IDfSessionManager.getSession` method.

Private sessions can be used by the application thread that obtained the session. Using private sessions is only recommended if the application or thread must retain complete control of the session state for a specific transaction.

Private sessions are obtained by using a `newSession` method to obtain a session.

2.2.3 Explicit and implicit sessions

When end users open a session with a repository by an explicit request, that session is referred to as an explicit session. Explicit sessions can be either shared or private sessions.

Because some repositories have more than one Documentum CM Server and the servers are often running on different host machines, Foundation Java API methods let you be specific when requesting the connection. You can let the system choose which server to use or you can identify a specific server by name or host machine or both.

During an explicit session, the tasks a user performs may require working with a document or other object from another repository. When that situation occurs, Foundation Java API seamlessly opens an implicit session for the user with the other repository. For example, suppose you pass a reference to ObjectB from RepositoryB to a session object representing a session with RepositoryA. In such cases, Foundation Java API will open an implicit session with RepositoryB to perform the requested action on ObjectB.

Implicit sessions are managed by Foundation Java API and are invisible to the user and the application. However, resource management is more efficient for explicit sessions than for implicit sessions. Consequently, using explicit sessions, instead of relying on implicit sessions, is recommended.

Both explicit and implicit sessions count towards the maximum number of allowed sessions specified in the `dfc.session.max_count` configuration parameter.

2.2.4 Session configuration

A session configuration defines some basic features and functionality for the session. For example, the configuration defines which connection brokers the client can communicate with, the maximum number of connections the client can establish, and the size of the client cache.

2.2.4.1 dfc.properties file

Configuration parameters for client sessions are recorded in the `dfc.properties` file. This file is installed with default values for some configuration parameters. Other parameters are optional and must be explicitly set. Additionally, some parameters are dynamic and may be changed at runtime if the deployment environment allows. Every client application must be able to access the `dfc.properties` file.

The file is polled regularly to check for changes. The default polling interval is 30 seconds. The interval is configurable by setting a key in the `dfc.properties` file.

When Foundation Java API is initialized and a session is started, the information in this file is propagated to the runtime configuration objects.

For information about setting the connection attempt interval and the number of retries, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

2.2.4.2 Runtime configuration objects

There are three runtime (nonpersistent) configuration objects that govern a session:

- client config object
- session config object
- connection config object

The client config object is created when Foundation Java API is initialized. The configuration values in this object are derived primarily from the values recorded in the `dfc.properties` file. Some of the properties in the client config object are also reflected in the server config object.

The configuration values are applicable to all sessions started through that Foundation Java API instance. The session config and the connection config objects represent individual sessions with a repository. Each session has one session config object and one connection config object. These objects are destroyed when the session is terminated.

For more information about the list of properties in the configuration objects, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.

2.2.5 Closing repository sessions

From an end user viewpoint, a user or application repository session is terminated when the user or application disconnects or releases the session or when another user assumes ownership of the session. Internally, a released or disconnected session is held in a connection pool, to be reused. For information about how connection pooling is implemented, see "Connection pooling" on page 25.

When a user disconnects or releases a session or a new user assumes ownership of a repository session, all implicit sessions opened for that session are closed.

Objects obtained during a session are associated with the session and the session manager under which the session was obtained. If you close a session and then attempt to perform a repository operation on an object obtained during that session, Foundation Java API opens an implicit session for the operation.

A session manager is terminated using an `IDfSessionManager.close` method. Before terminating a session manager, you must make sure that:

- All sessions are released or disconnected
- All `beginClientControl` methods have a matching `endClientControl` executed
- All transactions opened with a `beginTransaction` have been committed or aborted

2.3 Concurrent sessions

Concurrent sessions are repository sessions that are open at the same time through one Documentum CM Server. The sessions can be for one user or multiple users. By default, a Documentum CM Server can have 100 connections open concurrently. The limit is configurable by setting the `concurrent_sessions` key in the `server.ini` file. You can edit this file using Documentum Administrator. Each connection to a Documentum CM Server, whether an explicit or implicit connection, counts as one connection. Documentum CM Server returns an error if the maximum number of sessions defined in the `concurrent_sessions` key is exceeded.

For more information about the `server.ini` file keys, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

2.4 Inactive repository sessions

Inactive repository sessions are sessions in which the server connection has timed out but the client application has not specifically disconnected from the server. If the client application sends a request to Documentum CM Server, the server reauthenticates the user and, if the user is authenticated, the inactive session automatically reestablishes its server connection and becomes active.

If a session was started with a single-use login ticket and that session times out, the session cannot be automatically restarted by default because the login ticket cannot be reused. To avoid this problem, an application can use `resetPassword`, an `IDfSession` method. This method allows an application to provide either the actual password for the user or another login ticket for the user. After the user connects with the initial login ticket, the application can either:

- Generate a second ticket with a long validity period and then use `resetPassword` to replace the single-use ticket
- Run `resetPassword` to replace the single-use ticket with the actual password of the user

Performing either option will make sure that the user is reconnected automatically if the user session times out.

2.5 Restricted sessions

A restricted session is a repository session opened for a user who connects with an expired operating system password. The only operation allowed in a restricted session is changing the user password. Applications can determine whether a session they begin is a restricted session by examining the value of the computed property `_is_restricted_session`. This property is `T (TRUE)` if the session is a restricted session.

2.6 Connection brokers

A connection broker is a name server for the Documentum CM Server. It provides connection information for Documentum CM Servers and application servers, and information about the proximity of network locations.

When a user or application requests a repository connection, the request goes to a connection broker identified in the client `dfc.properties` file. The connection broker returns the connection information for the repository or particular server identified in the request.

Connection brokers do not request information from Documentum CM Servers, but rely on the servers to regularly broadcast their connection information to them. Which connection brokers are sent server information is configured in the server config object of the server.

Which connection brokers a client can communicate with is configured in the `dfc.properties` file used by the client. You can define primary and backup connection brokers in the file. Doing so ensures that users will rarely encounter a situation in which they can not obtain a connection to a repository.

An application can also set the connection broker programmatically. This allows the application to use a connection broker that may not be included in the connection brokers specified in the `dfc.properties` file. The application must set the connection broker information before requesting a connection to a repository.

For more information about how servers, clients, and connection brokers interact, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*. For more information about setting a connection broker programmatically, see *OpenText Documentum Content Management - Foundation Java API Development Guide (EDCPKCL250400-DGD)*.

2.7 Native and secure connections

Documentum CM Servers, connection brokers, and client applications can communicate using native (non-secure) connections or secure connections that use the secure sockets layer (SSL) protocol.

By default, all Documentum CM Servers and connection brokers are configured to support only native (non-SSL) connections. However, during initial configuration, or at a later time, Documentum CM Server and connection brokers can be configured to support SSL connections. They can be configured to first attempt connection with either native or SSL, and then try the other mode if not successful.

Similarly, all client sessions, by default, request a native connection, but can be configured in the same way as Documentum CM Server and connection brokers.

To provide a secure connection to a client, the server must be configured to listen on a secure port. And this is configured in the server config object. To provide a secure connection to a connection broker, both the server and the broker must be configured. Documentum CM Server is configured to project to the connection broker with an SSL session and the connection broker is configured to listen for SSL connections. If you have a distributed installation, and you want to use only SSL connections, make sure that all the elements of your installation are configured to use and can support SSL connections.

To request a secure connection, the client application must have the appropriate value set in the `dfc.properties` file or must explicitly request a secure connection when a session is requested. The security mode requested for the session is defined in the `IDfLoginInfo` object used by the session manager to obtain the session.

The security mode requested by the client interacts with the connection type configured for the server and connection broker to determine whether the session request succeeds and what type of connection is established.

For more information about resetting the connection default for Documentum CM Server and how clients request a secure connection, see *OpenText Documentum*

Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD). The interaction between the Documentum CM Server setting and the client request is described in the associated *Javadocs*, in the description of the `IDfLoginInfo.setSecurityMode` method.

2.8 Connection pooling

Connection pooling is an optional feature that allows an explicit repository session to be recycled and used by more than one user. Connection pooling is an automatic behavior implemented in Foundation Java API through session managers. It provides performance benefits for applications, especially those that execute frequent connections and disconnections for multiple users.

Whenever a session is released or disconnected, Foundation Java API puts the session into the connection pool. This pool is divided into two levels. The first level is a homogeneous pool. When a session is in the homogeneous pool, it can be reused only by the same user. If, after a specified interval, the user has not reclaimed the session, the session is moved to the heterogeneous pool (level-2 pool). From that pool, the session can be claimed by any user.

When a session is claimed from the heterogeneous pool by a new user, Foundation Java API resets automatically any security and cache-related information as needed for the new user. Foundation Java API also resets the error message stack and rolls back any open transactions.

To obtain the best performance and resource management from connection pooling, connection pooling must be enabled through the `dfc.properties` file. If connection pooling is not enabled through the `dfc.properties` file, Foundation Java API only uses the homogeneous pool. The session is held in that pool for a longer period of time, and does not use the heterogeneous pool. If the user does not reclaim the session from the homogeneous pool, the session is terminated.

Simulating connection pooling at the application level is accomplished using an `IDfSession.assume` method. The method lets one user assume ownership of an existing primary repository session.

When connection pooling is simulated using an assume method, the session is not placed into the connection pool. Instead, ownership of the repository session passes from one user to another by executing the assume method within the application.

When an assume method is issued, the system authenticates the requested new user. If the user passes authentication, the system resets the security and cache information for the session as needed. It also resets the error message stack.

For more information about enabling and configuring connection pooling, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*. The associated *Javadocs* contain details about using an assume method.

2.9 Login tickets

A login ticket is an ASCII-encoded string that an application can use in place of a user password when connecting to a repository. Login tickets can be used to establish a connection with the current or a different repository.

Each login ticket has a scope that defines who can use the ticket and how many times the ticket can be used. By default, login tickets may be used multiple times. However, you can create a ticket configured for only one use. If a ticket is configured for just one use, the ticket must be used by the issuing server or another designated server.

Login tickets are generated in a repository session, at runtime, using one of the `getLoginTicket` methods from the `IDfSession` interface.

2.9.1 Login ticket format and scope

A login ticket has the following format:

```
DM_TICKET=ASCII-encoded string
```

The ASCII-encoded string is comprised of two parts: a set of values describing the ticket and a signature generated from those values. The values describing the ticket include information such as when the ticket was created, the repository in which it was created, and who created the ticket. The signature is generated using the login ticket key installed in the repository.

For troubleshooting purposes, Foundation Java API supports the `IDfClient.getLoginTicketDiagnostics` method, which returns the encoded values in readable text format.

The scope of a login ticket defines which Documentum CM Servers accept the login ticket. When you generate a login ticket, you can define its scope as:

- The server that issues the ticket
- A single server other than the issuing server. In this case, the ticket is automatically a single-use ticket.
- The issuing repository. Any server in the repository accepts the ticket.
- All servers of trusting repositories. Any server of a repository that considers the issuing repository a trusted repository may accept the ticket.

A login ticket that can be accepted by any server of a trusted repository is called a global login ticket. An application can use a global login ticket to connect to a repository that differs from the ticket issuing repository if:

- The login ticket key (LTK) in the receiving repository is identical to the LTK in the repository in which the global ticket was generated
- The receiving repository trusts the repository in which the ticket was generated

For more information about how trusted repositories are defined and identified, see "Trusting and trusted repositories" on page 32.

2.9.2 Login ticket key

The LTK is a symmetric key, automatically installed in a repository when the repository is created. Each repository has one LTK. The LTK is stored in the `ticket_crypto_key` property of the docbase config object.

Login ticket keys are used with login tickets and application access tokens.

Login ticket keys are used to generate the Documentum CM Server signatures that are part of a login ticket key or application access token. If you want to use login tickets across repositories, the repository from which a ticket was issued and the repository receiving the ticket must have identical login ticket keys. When a Documentum CM Server receives a login ticket, it decodes the string and uses its login ticket key to verify the signature. If the LTK used to verify the signature is not identical to the key used to generate the signature, the verification fails.

Documentum CM Server supports two administration methods that allow you to export a login ticket key from one repository and import it into another repository. The methods are `EXPORT_TICKET_KEY` and `IMPORT_TICKET_KEY`. These methods are also available as Foundation Java API methods in the `IDfSession` interface.

It is also possible to reset a repository LTK if needed. Resetting a key removes the old key and generates a new key for the repository.

For more information about executing the `EXPORT_TICKET_KEY` and `IMPORT_TICKET_KEY` methods, and about resetting a login ticket key, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

2.9.3 Login ticket expiration

Login tickets are valid for given period of time, determined by configuration settings in the server config object or by an argument provided when the ticket is created. The configuration settings in the server config object define both a default validity period for tickets created by that server and a maximum validity period. The default period is defined in the `login_ticket_timeout` property. The maximum period is defined in the `max_login_ticket_timeout` property.

A validity period specified as an argument overrides the default defined in the server config object. However, if the method argument exceeds the maximum validity period in `max_login_ticket_timeout`, the maximum period is used.

For example, suppose you configure a server so that login tickets created by that server expire by default after 10 minutes and set the maximum validity period to 60 minutes. Now suppose that an application creates a login ticket while connected to that server and sets the ticket validity period to 20 minutes. The value set by the application overrides the default, and the ticket is valid for 20 minutes. If the application attempts to set the ticket validity period to 120 minutes, the 120 minutes is ignored and the login ticket is created with a validity period of 60 minutes.

If an application creates a ticket and does not specify a validity period, the default period is applied to the ticket.

When a login ticket is generated, both its creation time and expiration time are recorded as UTC time. This ensures that problems do not arise from tickets used across time zones.

When a ticket is sent to a server other than the server that generated the ticket, the receiving server tolerates up to a three-minute difference in time. That is, if the ticket is received within three minutes of its expiration time, the ticket is considered valid. This three-minute difference allows for minor differences in machine clock time across host machines. However, it is the responsibility of the system administrators to ensure that the machine clocks on host machines with applications and repositories be set as closely as possible to the correct time.

For more information about configuring the default and maximum validity periods in a repository, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

2.9.4 Revoking login tickets

You can set a cutoff date for login tickets on individual repositories. If you set a cutoff date for a repository, the repository servers consider any login tickets invalid that were generated prior to the specified date and time to be revoked. When a server receives a connection request with a revoked login ticket, it rejects the connection request.

The cutoff date is recorded in the `login_ticket_cutoff` property of the repository docbase config object.

This feature adds more flexibility to the use of login tickets by allowing you to create login tickets that may be valid in some repositories and invalid in other repositories. A ticket may be unexpired but still be invalid in a particular repository if that repository has `login_ticket_cutoff` set to a date and time prior to the ticket creation date.

2.9.5 Restricting superuser use

You can disallow use of a global login ticket by a superuser when connecting to a particular server. This is a security feature of login tickets. For example, suppose there is a userX in RepositoryA and a userX in RepositoryB and that the userX in RepositoryB is a superuser. Suppose also that the two repositories trust each other. An application connected to RepositoryA could generate a global login ticket for the userX (from RepositoryA) that allows that user to connect to RepositoryB. Because userX is a superuser in RepositoryB, when userX from RepositoryA connects, that person is granted Superuser privileges in RepositoryB.

To ensure that sort of security breach cannot occur, you can restrict superusers from using a global login ticket to connect to a server.

For more information about restricting superuser use of global login tickets, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

2.10 Application access control tokens

This section describes application access control tokens, an optional feature that gives you added control over access to repositories.

Application access control (AAC) tokens are encoded strings that may accompany connection requests from applications. The information in a token defines constraints on the connection request. If a Documentum CM Server is configured to use AAC tokens, any connection request received by that server from a non-superuser must be accompanied by a valid token and the connection request must comply with the constraints in the token.

If you configure a Documentum CM Server to use AAC tokens, you can control:

- Which applications can access the repository through that server
- Who can access the repository through that server

  You can allow any user to access the repository through that server or you can limit access to a particular user or to members of a particular group.
- Which client host machines can be used to access the repository through that server

These constraints can be combined. For example, you can configure a token that only allows members of a particular group using a particular application from a specified host to connect to a server.

Application access control tokens are ignored if the user requesting a connection is a superuser. A superuser can connect without a token to a server that requires a token. If a token is provided, it is ignored.

2.10.1 Using tokens

Tokens are enabled on a server-by-server basis. You can configure a repository with multiple servers so that some of its servers require a token and some do not. This provides flexibility in system design. For example, you can designate one Documentum CM Server assigned to a repository as the server to be used for connections coming from outside a firewall. By requiring that server to use tokens, you can further restrict what machines and applications are used to connect to the repository from outside the firewall.

When you create a token, you use arguments on the command line to define the constraints that you want to apply to the token. The constraints define who can use the token and in what circumstances. For example, if you identify a particular group in the arguments, only members of that group can use the token. Or, you can set an argument to constrain the token use to the host machine on which the token was generated. If you want to restrict the token to use by a particular application, you supply an application ID string when you generate the token, and any application using the token must provide a matching string in its connection request. All of the constraint parameters you specify when you create the token are encoded into the token.

When an application issues a connection request to a server that requires a token, the application may generate a token at runtime or it may rely on the client library to append an appropriate token to the request. The client library also appends a host machine identifier to the request.

Note: Only 5.3 Foundation Java API or later is capable of appending a token or machine identifier to a connection request. Configuring Foundation Java API to append a token is optional.

If you want to constrain the use to a particular host machine, you must also set the `dfc.machine.id` key in the `dfc.properties` file used by the client on that host machine.

If the receiving server does not require a token or the user is a superuser, the server ignores any token, application ID, and host machine ID accompanying the request and processes the request as usual.

If the receiving server requires a token, the server decodes the token and determines whether the constraints are satisfied. If the constraints are satisfied, the server allows the connection. If not, the server rejects the connection request.

For more information about enabling token use in a server, configuring Foundation Java API to append a token or machine identifier to a connection request, and implementing token use and enabling token retrieval, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

2.10.2 Token format and scope

The format of an AAC token is:

```
DM_TOKEN=ASCII-encoded string
```

The ASCII-encoded string is comprised of two parts: a set of values describing the token and a signature generated from those values. The values describing the token include such information as when the token was created, the repository in which it was created, and who created the token. For troubleshooting purposes, Foundation Java API has the `IDfClient.getApplicationTokenDiagnostics` method, which returns the encoded values in readable text format. The signature is generated using the repository login ticket key.

The scope of an application access control token identifies which Documentum CM Servers can accept the token. The scope of an AAC token can be either a single repository or global. The scope is defined when the token is generated.

If the scope of a token is a single repository, then the token is only accepted by Documentum CM Servers of that repository. The application using the token can send its connection request to any of the repository servers.

A global token can be used across repositories. An application can use a global token to connect to repository other than the repository in which the token was generated, if:

- The target repository is using the same LTK as the repository in which the global token was generated
- The target repository trusts the repository in which the token was generated

  Repositories that accept tokens generated in other repositories must trust these other repositories.

For more information, see “Login ticket key” on page 27. For more information about how trust is determined between repositories, see “Trusting and trusted repositories” on page 32.

2.10.3 Token generation and expiration

Application access control tokens can be generated at runtime or you can generate and store tokens for later retrieval by Foundation Java API. For runtime generation in an application, use the `getApplicationToken` method defined in the `IDfSession` interface.

To generate tokens for storage and later retrieval, use the `dmtkgen` utility. This option is useful if you want to place a token on a host machine outside a firewall so that users connecting from that machine are restricted to a particular application. It is also useful for backwards compatibility. You can use stored tokens retrieved by Foundation Java API to ensure that methods or applications written prior to version 5.3 can connect to servers that now require a token.

The `dmtkgen` utility generates an XML file that contains a token. The file is stored in a location identified by the `dfc.tokenstorage_dir` key in the `dfc.properties` file. Token use is enabled by `dfc.tokenstorage.enable` key. If use is enabled, a token can be retrieved and appended to a connection request by Foundation Java API when needed.

Application access control tokens are valid for a given period of time. The period may be defined when the token is generated. If not defined at that time, the period defaults to one year, expressed in minutes. Unlike login tickets, you cannot configure a default or maximum validity period for an application access token.

For more information about using `dmtkgen`, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

2.10.4 Internal methods, user methods, and tokens

The internal methods supporting replication and federations are not affected by enabling token use in any server. These methods are run under an account with the Superuser privileges, so the methods can connect to a server without a token even if that server requires a token.

Similarly if a user method (program or script defined in a `dm_method` object) runs under a superuser account, the method can connect to a server without a token even if that server requires a token. However, if the method does not run as a superuser and tries to connect without a token to a server that requires a token, the connection attempt fails.

You can avoid the failure by setting up and enabling token retrieval by Foundation Java API on the host on which the method is executed. Token retrieval allows Foundation Java API to append a token retrieved from storage to the connection request. The token must be generated by the `dmtkgen` utility and must be a valid token for the connection request.

2.11 Trusting and trusted repositories

A trusting repository is a repository that accepts login tickets or application access tokens, or both, that were generated by a Documentum CM Server from a different repository. A trusted repository is a repository that generates login tickets or application access tokens, or both, that are accepted by a different repository. The repositories whose servers generate the tickets or application access tokens or receive the tickets or tokens must be appropriately configured as trusted or trusting repositories.

All repositories run in either trusting or nontrusting mode. Whether a repository is running in trusting or nontrusting mode is defined in the `trust_by_default` property in the docbase config object.

If `trust_by_default` is set to `T`, the repository is running in trusting mode and trusts all other repositories. In trusting mode, the repository accepts any global login ticket or application access token generated with a LTK that matches its LTK, regardless of the ticket or token source repository. If the property is set to `F`, the repository is running in nontrusting mode. A nontrusting repository accepts global login tickets or application access tokens generated with a matching LTK if they come from repositories specifically named as trusted repositories. The list of trusted repository names is recorded in a repository `trusted_docbases` property in its docbase config object.

For example, suppose an installation has four repositories: RepositoryA, RepositoryK, RepositoryM, and RepositoryN. All four have identical LTKs. RepositoryA has `trust_by_default` set to `T`. Therefore, RepositoryA trusts and accepts login tickets or tokens from the three other repositories. RepositoryK has `trust_by_default` set to `F`. RepositoryK also has two repositories listed in its `trusted_docbases` property: RepositoryM and RepositoryN. RepositoryK rejects login tickets or tokens from RepositoryA because RepositoryA is not in the list of trusted repositories. It accepts tickets or tokens from RepositoryM and RepositoryN because they are listed in the `trusted_docbases` property.

2.12 Transaction management

This section describes transactions and how they are managed.

A transaction is one or more repository operations handled as an atomic unit. All operations in the transaction must succeed or none may succeed. A repository session can have only one open transaction at any particular time. A transaction is either internal or explicit.

2.12.1 Internal and explicit transactions

An internal transaction is a transaction managed by Documentum CM Server. The server opens transactions, commits changes, and performs rollbacks as necessary to maintain the integrity of the data in the repository. Typically, an internal transaction consists of only a few operations. For example, a save on a `dm_sysobject` is one transaction, consisting of minimally three operations: saving the `dm_sysobject_s` table, saving the `dm_sysobject_r` table, and saving the content file. If any of the save operations fail, the transaction fails and all changes are rolled back.

An explicit transaction is a transaction managed by a user or client application. The transaction is opened with a DQL `BEGINTRAN` statement or a `beginTransaction` method. It is closed when the transaction is explicitly committed to save the changes, or aborted to close the transaction without saving the changes. An explicit transaction can include as many operations as desired. However, none of the changes made in an explicit transaction are committed until the transaction is explicitly committed. If an operation fails, the transaction is automatically aborted and all changes made prior to the failure are lost.

2.12.2 Constraints on explicit transactions

There are constraints on the work you can perform in an explicit transaction:

- You cannot perform any operation on a remote object if the operation results in an update in the remote repository.

  Opening an explicit transaction starts the transaction only for the current repository. If you issue a method in the transaction that references a remote object, work performed in the remote repository by the method is not under the control of the explicit transaction. This means that if you abort the transaction, the work performed in the remote repository is not rolled back.
- You cannot perform any of the following methods that manage objects in a lifecycle: attach, promote, demote, suspend, and resume.
- You cannot issue a complete method for an activity if the activity is using XPath to route a case condition to define the transition to the next activity.
- You cannot execute an `IDfSysObject.assemble` method that includes the `interruptFreq` argument.
- You cannot use the Foundation Java API methods in the transaction if you opened the transaction with the DQL `BEGIN[TRAN]` statement.

  If you want to use the Foundation Java API methods in an explicit transaction, open the transaction with a Foundation Java API method.
- You cannot execute dump and load operations inside an explicit transaction.
- You cannot issue a `CREATE TYPE` statement in an explicit transaction.
- You cannot issue an `ALTER TYPE` statement in an explicit transaction, unless the `ALTER TYPE` statement lengthens a string property.

2.12.3 Database-level locking in explicit transactions

Database-level locking places a physical lock on an object in the RDBMS tables. Database-level locking is more severe than that provided by the checkout method and is only available in explicit transactions.

Applications may find it advantageous to use database-level locking in explicit transactions. If an application knows which objects it will operate on and in what order, the application can avoid deadlock by placing database locks on the objects in that order. You can also use database locks to ensure that version mismatch errors do not occur.

To put a database lock on an object, use the `lockEx(true)` method (in the `IDfPersistentObject` interface). A superuser can lock any object with a database-level lock. Other users must have at least Version permission on an object to place a database lock on the object.

After an object is physically locked, the application can modify the properties or content of the object. It is not necessary to issue a checkout method unless you want to version the object. If you want to version an object, you must also check out the object.

2.12.4 Managing deadlocks

Deadlock occurs when two connections are both trying to access the same information in the underlying database. When deadlock occurs, the RDBMS typically chooses one of the connections as a victim, drops any locks held by that connection, and rolls back any changes made in that connection transaction.

2.12.4.1 Handling deadlocks in internal transactions

Documentum CM Server manages internal transactions and database operations in a manner that reduces the chance of deadlock as much as possible. However, some situations may still cause deadlocks. For example, deadlocks can occur if:

- A query that turns off full-text search and tries to read data from a table through an index when another connection is locking the data while it tries to update the index. When full-text search is enabled, properties are indexed and the table is not queried.
- Two connections are waiting for locks being held by each other.

When deadlock occurs, Documentum CM Server executes internal deadlock retry logic. The deadlock retry logic tries to execute the operations in the victim transaction up to 10 times. If an error such as a version mismatch occurs during the retries, the retries are stopped and all errors are reported. If the retry succeeds, an informational message is reported.

2.12.4.2 Handling deadlocks in explicit transactions

Documentum CM Server deadlock retry logic is not available in explicit transactions. If an application runs under an explicit transaction or contains an explicit transaction, the application should contain deadlock retry logic.

Documentum CM Server provides a computed property that you can use in applications to test for deadlock. The property is `_isdeadlocked`. This is a Boolean property that returns `TRUE` if the repository session is deadlocked.

To test custom deadlock retry logic, Documentum CM Server provides an administration method called `SET_APIDEADLOCK`. This method plants a trigger on a particular operation. When the operation executes, the server simulates a deadlock, setting the `_isdeadlocked` computed property and rolling back any changes made prior to the method execution. Using `SET_APIDEADLOCK` allows you to test an application deadlock retry logic in a development environment. For more information about the `SET_APIDEADLOCK` method, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

Chapter 3

Caching

3.1 Object type caching

Documentum CM Server and Foundation Java API maintain caches of object type definitions. These caches help to ensure fast response times when users access objects. To ensure that the cached information about an object type is accurate, Foundation Java API and Documentum CM Server have mechanisms to verify the accuracy of a cached object type definition and update it if necessary. The mechanism varies depending on the object types.

3.1.1 Object types with names beginning with dm, dmr, and dmi

These object types are built-in types in a Documentum CM Server installation. Their type definitions are relatively static. There are few changes that can be made to the definition of a built-in type. For these types, the mechanism is an internal checking process that periodically checks all the object type definitions in the Documentum CM Server global cache. If any definitions are out-of-date, the process flushes the cache and reloads the type definitions into the global cache. Changes to these types are not visible to existing sessions because the Foundation Java API caches are not updated when the global cache is refreshed.

Stopping and restarting a session makes any changes in the global cache visible. If the session was a web-based client session, the web application server must be restarted.

The interval at which the process runs is configurable by changing the setting of the `database_refresh_interval` in the `server.ini` file.

3.1.2 Custom object types and types with names beginning with dmc

These object types are installed with the DAR files or scripts to support client applications. Their type definitions typically change more often, and the changes may need to be visible to users immediately. For example, a Collaboration Services user can change the structure of a datatable often, and each change modifies the underlying type definition. To meet that requirement, the mechanism that refreshes cached type definitions for these types is more dynamic than that for the built-in types.

For these types, Foundation Java API shared cache is updated regularly, at intervals defined by the `dfc.cache.type.currency_check_interval` key in the `dfc.properties` file. That key defaults to 300 seconds (5 minutes). It can be reset using Documentum Administrator.

Additionally, when requested in a fetch method, Foundation Java API checks the consistency of its cached version against the server global cache. If the versions in the caches are found to be mismatched, the object type definition is updated appropriately. If the server cache is more current, the Foundation Java API caches are updated. If Foundation Java API has a more current version, the server cache is updated.

This mechanism ensures that a user who makes the change sees that change immediately and other users in other sessions see it shortly thereafter. Stopping and restarting a session or the web application server is not required to see changes made to these objects.

For more information about setting the database refresh interval for the server global cache, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

3.2 Repository session caches

Repository session caches are created when a user or application opens a repository session. These caches exist only for the life of the repository session. The types of caches are:

- Object cache

  An in-memory object cache is maintained for each repository session for the duration of the repository session.

- Data dictionary caches

  In conjunction with the object cache, Foundation Java API maintains a data dictionary cache. The data dictionary cache is a shared cache, shared by all sessions in a multi-threaded application. When an object is fetched, the Foundation Java API also fetches and caches in memory the object associated data dictionary objects if they are not already in the cache.

3.3 Consistency checking

Consistency checking is the process that ensures that cached data accessed by a client is current and consistent with the data in the repository. How often the process is performed for any particular set of query results is determined by the consistency check rule defined in the method that references the data.

The consistency check rule can be a keyword, an integer value, or the name of a cache config object. Using a cache config object to group cached data has the following benefits:

- Validates cached data efficiently

  It is more efficient to validate a group of data than it is to validate each object or set of query results individually.

- Helps ensure that applications access current data

- Makes it easy to change the consistency check rule because the rule is defined in the cache config object rather than in application method calls
- Allows you to define a job to validate cached data automatically

Consistency checking is basically a two-part process:

1. Foundation Java API determines whether a consistency check is necessary.
2. Foundation Java API conducts the consistency check if needed.

The consistency checking process described in this section is applied to all objects in the in-memory cache, regardless of whether the object is persistently cached or not.

For more information about how the Foundation Java API determines whether a check is needed, see "Determining if a consistency check is needed" on page 39. For more information about how the check is conducted, see "Conducting consistency checks" on page 41.

3.3.1 Determining if a consistency check is needed

To determine whether a check is needed, the Foundation Java API uses the consistency check rule defined in the method that references the data. The rule may be expressed as either a keyword, an integer value, or the name of a cache config object.

3.3.1.1 Rules with a keyword or integer

If the rule was specified as a keyword or an integer value, Foundation Java API interprets the rule as a directive on when to perform a consistency check. The directive is one of the following:

- Perform a check every time the data is accessed

  This option means that the data is always checked against the repository. If the cached data is an object, the object is always checked against the object in the repository. If the cached data is a set of query results, the results are always regenerated. The keyword `check_always` defines this option.

- Never perform a consistency check

  This option directs Foundation Java API to always use the cached data. The cached data is never checked against the repository if it is present in the cache. If the data is not present in the cache, the data is obtained from the server. The keyword `check_never` defines this option.

- Perform a consistency check on the first access only

  This option directs Foundation Java API to perform a consistency check the first time the cached data is accessed in a session. If the data is accessed again during the session, a consistency check is not conducted. The keyword `check_first_access` defines this option.

- Perform a consistency check after a specified time interval

This option directs Foundation Java API to compare the specified interval to the timestamp on the cached data and perform a consistency check only if the interval has expired. The timestamp on the cached data is set when the data is placed in the cache. The interval is expressed in seconds and can be any value greater than 0.

3.3.1.2 Rules with a cache config object

If a consistency check rule names a cache config object, Foundation Java API uses information from the cache config object to determine whether to perform a consistency check on the cached data. The cache config information is obtained by invoking the `CHECK_CACHE_CONFIG` administration method and stored in memory with a timestamp that indicates when the information was obtained. The information includes the `r_last_changed_date` and the `client_check_interval` property values of the cache config object.

When a method defines a consistency check rule by naming a cache config object, Foundation Java API first checks whether it has information about the cache config object in its memory. If it does not, it issues a `CHECK_CACHE_CONFIG` administration method to obtain the information. If it has information about the cache config object, Foundation Java API must determine whether the information is current before using that information to decide whether to perform a consistency check on the cached data.

To determine whether the cache config information is current, Foundation Java API compares the stored `client_check_interval` value to the timestamp on the information. If the interval has expired, the information is considered out of date and Foundation Java API executes another `CHECK_CACHE_CONFIG` method to ask Documentum CM Server to provide current information about the cache config object. If the interval has not expired, Foundation Java API uses the information that it has in memory.

After Foundation Java API has current information about the cache config object, it determines whether the cached data is valid. To determine that, Foundation Java API compares the timestamp on the cached data against the `r_last_changed_date` property value in the cache config object. If the timestamp is later than the `r_last_changed_date` value, the cached data is considered usable and no consistency check is performed. If the timestamp is earlier than the `r_last_changed_date` value, a consistency check is performed on the data.

For more information about the `CHECK_CACHE_CONFIG` administration method, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

3.3.2 Conducting consistency checks

To perform a consistency check on a cached object, Foundation Java API uses the `i_vstamp` property value of the object. If Foundation Java API has determined that a consistency check is needed, it compares the `i_vstamp` value of the cached object to the `i_vstamp` value of the object in the repository. If the `i_vstamp` values are different, Foundation Java API refetches the object and resets the time stamp. If they are the same, Foundation Java API uses the cached copy.

Foundation Java API does not perform consistency checks on cached query results. If the cached results are out of date, Documentum CM Server reexecutes the query and replaces the cached results with the newly generated results.

If a fetch method does not include an explicit value for the argument defining a consistency check rule, the default is `check_always`. That means that Foundation Java API checks the `i_vstamp` value of the in-memory object against the `i_vstamp` value of the object in the repository. The default consistency rule is `check_never`. This means that Foundation Java API uses the cached query results.

Chapter 4

Data model

4.1 Objects and object types

The OpenText Documentum CM system is object-oriented. An object is an individual item in the repository. All the repository items manipulated by users are objects. Every document is an object, as are the cabinets and folders in which documents are stored. Even users are handled as objects. Each of the objects belongs to an object type.

An object type represents a class of objects. The definition of an object type consists of a set of properties, whose values describe individual objects of the type. Object types are similar to templates. When you create an object in a repository, you identify which type of object you want to create. Documentum CM Server uses the type definition as a template to create the object, and then sets the properties for the object to values specific to that object instance.

Most OpenText Documentum CM object types exist in a hierarchy. Within the hierarchy, an object type is a supertype or a subtype or both. A supertype is an object type that is the basis for another object type, called a subtype. The subtype inherits all the properties of the supertype. The subtype also has the properties defined specifically for it. For example, the `dm_folder` type is a subtype of `dm_sysobject`. It has all the properties defined for `dm_sysobject` plus two defined specifically for `dm_folder`.

A type can be both a supertype and a subtype. For example, `dm_folder` is a subtype of `dm_sysobject` and a supertype of `dm_cabinet`.

Most object types are persistent. When a user creates an object of a persistent type, the object is stored in the repository and persists across sessions. A document that a user creates and saves one day is stored in the repository and available in another session on another day. The definitions of persistent object types are stored in the repository as objects of type `dm_type` and `dmi_type_info`.

There are some object types that are not persistent. Objects of these types are created at runtime when they are needed. For example, collection objects and query result objects are not persistent. They are used at runtime to return the results of DQL statements. When the underlying RDBMS returns rows for a SELECT statement, Documentum CM Server places each returned row in a query result object and then associates the set of query result objects with a collection object. Neither the collection object nor the query result objects are stored in the repository. When you close the collection, after all query result objects are retrieved, both the collection and the query result objects are destroyed.

4.1.1 Object type categories

Object types are sorted into categories to facilitate their management by Documentum CM Server. The categories are:

- Standard object type

  Standard object types are types that do not fall into one of the remaining categories.
- Aspect property object type

  Aspect property object types are internal types used by Documentum CM Server and Foundation Java API to manage properties defined for aspects. These types are automatically created and managed internally when properties are added to aspects. They are not visible to users and user applications.
- Lightweight object type

  Lightweight object types are a special type used to minimize the storage footprint for multiple objects that share the same system information. A lightweight type is a subtype of its shareable type.
- Shareable object type

  Shareable object types are the parent types of lightweight object types. Only `dm_sysobject` and its subtypes can be defined as shareable. A single instance of a shareable type object is shared among many lightweight objects.

Lightweight and shareable object types are additional types added to Documentum CM Server to solve common problems with large content stores. Specifically, these types can increase the rate of object ingestion into a repository and can reduce the object storage requirements.

An object type category is stored in the `type_category` property in the `dm_type` object representing the object type.

For more information about how lightweight and shareable types are associated within the underlying database tables, see “Storing lightweight subtype instances” on page 53.

4.1.2 Lightweight object types

A lightweight type is a type whose implementation is optimized to reduce the storage space needed in the database for instances of the type. All lightweight SysObjects types (a SysObject is the parent type of the most commonly used objects in the OpenText Documentum CM system) are subtypes of a shareable type. When a lightweight SysObject is created, it references a shareable supertype object. As additional lightweight SysObjects are created, they can reference the same shareable object. That shareable object is called the lightweight SysObject parent, and the lightweight SysObject is the child. Each lightweight SysObject shares the information in its shareable parent object. In that way, instead of having multiple nearly identical rows in the SysObject tables to support all the instances of the lightweight type, a single parent object exists for multiple lightweight SysObjects.

You may see a lightweight SysObject referred to as a lightweight object, or sometimes abbreviated as LWSO. All of these terms are equivalent.

Lightweight objects are useful if you have a large number of properties that are identical for a group of objects. This redundant information can be shared among the LWSOs from a single copy of the shared parent object. For example, Enterprise A-Plus Financial Services receives many payment checks each day. They record the images of the checks and store the payment information in SysObjects. They will retain this information for several years and then delete it. For their purposes, all objects created on the same day can use a single ACL, retention information, creation date, version, and other properties. That information is held by the shared parent object. The LWSO has information about the specific transaction.

Using lightweight SysObjects can provide the following benefits:

- Lightweight types take up less space in the underlying database tables than a standard subtype.
- Importing lightweight objects into a repository is faster than importing standard SysObjects.

4.1.3 Shareable object types

A shareable type is a type whose instances can share its property values with instances of lightweight types. It is possible for multiple lightweight objects to share the property values of one shareable object. The shareable object that is sharing its properties with the lightweight object is called the parent object, and the lightweight object is called its child.

4.1.4 OpenText Documentum CM system object type names

The names of all object types that are installed with Documentum CM Server or by a OpenText Documentum CM client product start with the letters `dm`. There are four such prefixes:

- `dm`, which represents object types that are commonly used and visible to users and applications.
- `dmr`, which represents object types that are generally read only.
- `dmi`, which represents object types that are used internally by Documentum CM Server and OpenText Documentum CM client components.
- `dmc`, which represents object types installed to support a OpenText Documentum CM client application. They are typically installed by a script when Documentum CM Server is installed or when the client component is installed.

The use of `dm` as the first two characters in an object type name is reserved for OpenText Documentum CM products.

For more information about the rules for naming user-defined object types and properties, and a description of the `dm_lightweight` object type, see *OpenText*

Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD).

4.1.5 Content files and object types

The SysObject object type and all of its subtypes, except cabinets, folders, and their subtypes, have the ability to accept content. You can associate one or more content files with individual objects of the type.

The content associated with an object is either primary content or renditions of the primary content. The format of all primary content for any one object must have the same file format. The renditions can be in any format. A rendition of a document is a content file that differs from the source document content file only in its format.

If you want to create a document that has primary content in a variety of formats, you must use a virtual document. Virtual documents are a hierarchical structure of component documents that can be published as a single document. The component documents can have different file formats.

- Virtual documents: For more information, see "Virtual documents" on page 151.
- Adding content to objects: For more information, see "Adding content" on page 132.
- Renditions: For more information, see "Renditions" on page 111.

4.2 Properties

Properties are the fields that comprise an object definition. The values in those fields describe individual instances of the object type. When an object is created, its properties are set to values that describe that particular instance of the object type. For example, two properties of the document object type are title and subject. When you create a document, you provide values for the title and subject properties that are specific to that document.

4.2.1 Property characteristics

Properties have a number of characteristics that define how they are managed and handled by Documentum CM Server. These characteristics are set when the property is defined and cannot be changed after the property is created.

4.2.1.1 Persistent and non-persistent

The properties that make up a persistent object type definition are persistent. Their values for individual objects of the type are saved in the repository. These persistent properties and their values make up the object metadata.

An object type persistent properties include not only the properties defined for the type, but also those that the type inherits from it supertype. If the type is a lightweight object type, its persistent properties also include those it shares with its sharing type.

Many object types also have associated computed properties. Computed properties are non-persistent. Their values are computed at runtime when a user requests the property value and lost when the user closes the session.

Persistent properties have domains. A property domain identifies the property datatype and several other characteristics of the property, such as its default value or label text for it. You can query the data dictionary to retrieve the characteristics defined by a domain.

For more information about supertypes and inheritance, see “Objects and object types” on page 43. For more information about the persistent and computed properties, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.

4.2.1.2 Single-valued and repeating

All properties are either single-valued or repeating. A single-valued property stores one value or, if it stores multiple values, stores them in one comma-separated list. Querying a single-valued property returns the entire value, whether it is one value or a list of values.

A repeating property stores multiple values in an indexed list. Index positions within the list are specified in brackets at the end of the property name when referencing a specific value in a repeating property. An example is: `keywords[2]` or `authors[17]`. Full-text queries can also locate specific values in a repeating property. Special DQL functions exist to allow you to query values in a repeating property.

4.2.1.3 Datatype

All properties have a datatype that determines what kind of values can be stored in the property. For example, a property with an integer datatype can only store whole numbers. A property datatype is specified when the object type for which the property is defined is created.

For more information about valid datatypes and the limits and defaults for each datatype, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.

4.2.1.4 Read only or read and write

All properties are either read only or read and write. Read-only properties are those that only Documentum CM Server can write. Read and write properties can typically be operated on by users or applications. In general, the prefix, or lack of a prefix, on a property name indicates whether the property is read only or can also be written.

User-defined properties are read and write by default. Only superusers can add a read-only property to an object type.

4.2.1.5 Qualifiable and non-qualifiable

Persistent properties are either qualifiable or non-qualifiable.

A qualifiable property is represented by a column in the appropriate underlying database table for the type that contains the property. The majority of properties are qualifiable. By default, a property is created as a qualifiable property unless its definition explicitly declares it to be a non-qualifiable property.

A non-qualifiable property is stored in the `i_property_bag` property of the object. This is a special property that stores properties and their values in a serialized format. Non-qualifiable properties do not have their own columns in the underlying database tables that represent the object types for which they are defined. Consequently, the definition of a non-qualifiable property cannot include a Check constraint.

Both qualifiable and non-qualifiable properties can be full-text indexed, and both can be referenced in the selected values list of a query statement. Similar to qualifiable properties, selected non-qualifiable properties are returned by a query as a column in a query result object. However, non-qualifiable properties cannot be referenced in an expression in a qualification (such as in a WHERE clause) in a query unless the query is a full-text DQL query.

The `attr_restriction` property in the `dm_type` object identifies the type properties as either qualifiable or non-qualifiable.

4.2.1.6 Local and global

All persistent properties are either global or local. This characteristic is only significant if a repository participates in object replication or is part of a federation. A federation is a group of one or more repositories.

Object replication creates replica objects, copies of objects that have been replicated between repositories. When users change a global property in a replica, the change actually affects the source object property. Documentum CM Server automatically refreshes all the replicas of the object containing the property. When a repository participates in a federation, changes to global properties in users and groups are propagated to all member repositories if the change is made through the governing repository using Documentum Administrator.

A local property value can be different in each repository participating in the replication or federation. If a user changes a local property in a replicated object, the source object is not changed and neither are the other replicated objects.

Note: It is possible to configure four local properties of the `dm_user` object to make them behave as global properties.

For more information about creating global users and for configuring four local user properties to behave as global properties, see *OpenText Documentum Content Management - Server and Server Extensions Installation Guide (EDCSY250400-IGD)*.

4.2.1.7 Property identifiers

Every property has an identifier. These identifiers are used instead of property names to identify a property when the property is stored in a property bag. The property identifier is unique within an object type hierarchy. For example, all the properties of `dm_sysobject` and its subtypes have identifiers that are unique within the hierarchy that has `dm_sysobject` as its top-level supertype.

The identifier is an integer value stored in the `attr_identifier` property of each type `dm_type` object. When a property is stored in a property bag, its identifier is stored as a Base64-encoded string in place of the property name.

A property identifier cannot be changed.

For more information, see "Property bag" on page 49. For more information about property identifiers, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.

4.2.2 Property bag

The property bag is a special property used to store:

- Non-qualifiable properties and their values
- Aspect properties and their values

You can store both single-valued and repeating property values in a property bag.

4.2.2.1 Implementation

The property bag is implemented in a repository as the `i_property_bag` property. The `i_property_bag` property is part of the `dm_sysobject` type definition by default. Consequently, each subtype of `dm_sysobject` inherits this property. That means that you can define a subtype of the `dm_sysobject` or one of its subtypes that includes a non-qualifiable property without specifically naming the `i_property_bag` property in the subtype definition.

The `i_property_bag` property is not part of the definition of the `dm_lightweight` type. However, if you create a lightweight subtype whose definition contains a non-qualifiable property, Documentum CM Server automatically adds `i_property_bag` to the type definition. It is not necessary to explicitly name the property in the type definition.

Similarly, if you include a non-qualifiable property in the definition of an object type that has no supertype or whose supertype is not in the `dm_sysobject` hierarchy, the `i_property_bag` property is added automatically to the type.

The `i_property_bag` property is also used to store aspect properties if the properties are optimized for fetching. Consequently, the object type definitions of object instances associated with the aspect must include the `i_property_bag` property. In this situation, you must explicitly add the property bag to the object type before associating its instances with the aspect.

It is also possible to explicitly add the property bag to an object type using an `ALTER TYPE` statement.

The property bag cannot be removed after it is added to an object type.

The `i_property_bag` property is a string datatype of 2000 characters. If the names and values of properties stored in `i_property_bag` exceed that size, the overflow is stored in a second property, called `r_property_bag`. This is a repeating string property of 2000 characters.

Whenever the `i_property_bag` property is added to an object type definition, the `r_property_bag` property is also added.

For more information about aspects and aspect properties, see "Aspects" on page 73. For information about the reference description for the property bag property, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*. For information about how to alter a type to add a property bag, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

4.3 Repositories

A repository is where persistent objects managed by Documentum CM Server are stored. A repository stores the object metadata and, sometimes, content files. A OpenText Documentum CM system installation can have multiple repositories. Each repository is uniquely identified by a repository ID, and each object stored in the repository is identified by a unique object ID.

Repositories contain sets of tables in an underlying relational database installation. Two types of tables are implemented:

- Object type tables
- Object type index tables

4.3.1 Object type tables

The object type tables store metadata.

Each persistent object type, such as `dm_sysobject` or `dm_group`, is represented by two tables in the set of object type tables. One table stores the values for the single-valued properties for all instances of the object type. The other table stores the values for repeating properties for all instances of the object type.

4.3.1.1 Single-valued property tables

The tables that store the values for single-valued properties are identified by the object type name followed by _s (for example, `dm_sysobject_s` and `dm_group_s`). In the _s tables, each column represents one property and each row represents one instance of the object type. The column values in the row represent the single-valued property values for that object.

4.3.1.2 Repeating property tables

The tables that store values for repeating properties are identified by the object type name followed by _r (for example, `dm_sysobject_r` and `dm_group_r`). In these tables, each column represents one property.

In the _r tables, there is a separate row for each value in a repeating property. For example, suppose a subtype called recipe has one repeating property, ingredients. A recipe object that has five values in the ingredients property will have five rows in the `recipe_r` table, one row for each ingredient, as shown in the following table:

r_object_id	ingredients
. . .	4 eggs
. . .	1 lb. cream cheese
. . .	2 t vanilla
. . .	1 c sugar

r_object_id	ingredients
. . .	2 T grated orange peel

The `r_object_id` value for each row identifies the recipe that contains these five ingredients.

If a type has two or more repeating properties, the number of rows in the _r table for each object is equal to the number of values in the repeating property that has the most values. The columns for repeating properties having fewer values are filled in with NULLs.

For example, suppose the recipe type has four repeating properties: authors, ingredients, testers, and ratings. One particular recipe has one author, four ingredients, and three testers. For this recipe, the ingredients property has the largest number of values, so this recipe object has four rows in the `recipe_r` table:

. . .	authors	ingredients	testers	ratings
. . .	yvonned	1/4 lb. butter	winifredh	4
. . .	NULL	1/2 c bittersweet chocolate	johnp	6
. . .	NULL	1 c sugar	claricej	7
. . .	NULL	2/3 cup light cream	NULL	NULL

The server fills out the columns for repeating properties that contain a smaller number of values with NULLs.

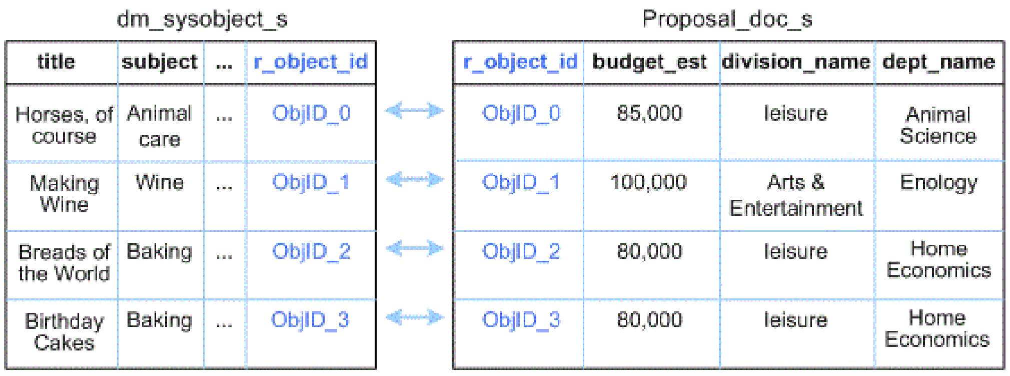
Even an object with no values assigned to any of its repeating properties has at least one row in its type _r table. The row contains a NULL value for each of the repeating properties. If the object is a SysObject or SysObject subtype, it has a minimum of two rows in its type _r table because its `r_version_label` property has at least one value- its implicit version label.

For information about how NULLs are handled in the OpenText Documentum CM system, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

4.3.1.3 How standard subtype instances are stored

If an object type is a subtype of a standard object type, the tables representing the object type store only the properties defined for the object type. The values for inherited properties are stored in rows in the tables of the supertype. In the _s tables, the `r_object_id` value serves to join the rows from the subtype _s table to the matching row in the supertype _s table. In the _r tables, the `r_object_id` and `i_position` values are used to join the rows.

For example, suppose you create a subtype of `dm_sysobject` called `proposal_doc`, with three properties: `budget_est`, `division_name`, and `dept_name`, all single-valued properties. Figure 4-1 illustrates the underlying table structure for this type and its instances. The values for the properties defined for the proposal doc type are stored in the `proposal_doc_s` table. Those properties that it inherits from its supertype, `dm_sysobject`, are stored in rows in the `dm_sysobject` object type tables. The rows are associated through the `r_object_id` column in each table.


dm_sysobject_s

title	subject	...	r_object_id
Horses, of course	Animal care	...	ObjID_0
Making Wine	Wine	...	ObjID_1
Breads of the World	Baking	...	ObjID_2
Birthday Cakes	Baking	...	ObjID_3

Proposal_doc_s

r_object_id	budget_est	division_name	dept_name
ObjID_0	85,000	leisure	Animal Science
ObjID_1	100,000	Arts & Entertainment	Enology
ObjID_2	80,000	leisure	Home Economics
ObjID_3	80,000	leisure	Home Economics

Figure 4-1: Standard subtype

4.3.1.4 Storing lightweight subtype instances

A lightweight type is a subtype of a shareable type, so the tables representing the lightweight type store only the properties defined for the lightweight type. The values for inherited properties are stored in rows in the tables of the shareable type (the supertype of the lightweight type). In standard objects, the `r_object_id` property is used to join the rows from the subtype to the matching rows in the supertype. However, since many lightweight objects can share the properties from their shareable parent object, the `r_object_id` values differ from the parent object `r_object_id` value. For lightweight objects, the `i_sharing_parent` property is used to join the rows. Therefore, many lightweight objects, each with its own `r_object_id`, can share the property values of a single shareable object.

When a lightweight object shares a parent object with other lightweight objects, the lightweight object is unmaterialized. All the unmaterialized lightweight objects share the properties of the shared parent, so, in effect, the lightweight objects all have identical values for the properties in the shared parent. This situation can change if some operation needs to change a parent property for one of (or a subset of) the lightweight objects. Since the parent is shared, the change in a property would affect all the children. If the change only affects one child, that child object has to have its own copy of the parent. When a lightweight object has its own private copy of a parent, the object is materialized. Documentum CM Server creates rows in the tables of the shared type for the object, copying the values of the shared properties into those rows. The lightweight object no longer shares the property values with the instance of the shared type, but with its own private copy of that shared object.

For example, if you checkout a lightweight object, it is materialized. A copy of the original parent is created with the same `r_object_id` value as the child and the lightweight object is updated to point to the new parent. Since the private parent has the same `r_object_id` as the lightweight child, a materialized lightweight object behaves similar to a standard object. As another example, if you delete an unmaterialized lightweight object, the shared parent is not deleted (whether or not there are any remaining lightweight children). If you delete a materialized lightweight object, the lightweight child and the private parent are deleted.

When, or if, a lightweight object instance is materialized depends on the object type definition. You can define a lightweight type such that instances are materialized automatically when certain operations occur, only on request, or never.

The following is an example of how lightweight objects are stored and how materialization changes the underlying database records. Note that this example only uses the _s tables to illustrate the implementation. The implementation is similar for _r tables.

Suppose the following shareable and lightweight object types exist in a repository:

- `customer_record`, with a SysObject supertype and the following properties:

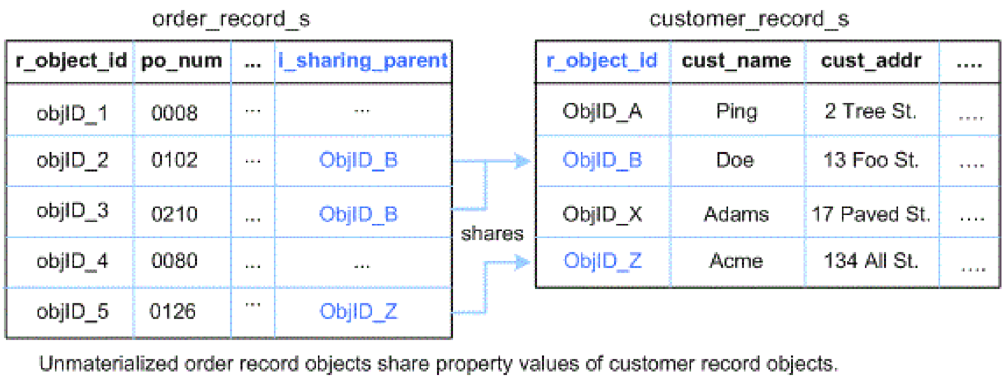
  ```
  cust_name string(32)
  cust_addr string(64)
  cust_city string(32)
  cust_state string(2)
  cust_phone string(24)
  cust_email string(100)
  ```

- `order_record`, with the following properties:

  ```
  po_number string(24)
  parts_ordered string(24)REPEATING
  delivery_date DATE
  billing_date DATE
  date_paid DATE
  ```

  This type shares with `customer_record` and is defined for automatic materialization.

Instances of the order record type will share the values of instances of the customer record object type. By default, the order record instances are unmaterialized. The following illustration describes how the unmaterialized lightweight instances are represented in the database tables:



r_object_id	po_num	...	i_sharing_parent
objID_1	0008	...	...
objID_2	0102	...	ObjID_B
objID_3	0210	...	ObjID_B
objID_4	0080	...	...
objID_5	0126	...	ObjID_Z

r_object_id	cust_name	cust_addr	....
ObjID_A	Ping	2 Tree St.	....
ObjID_B	Doe	13 Foo St.	....
ObjID_X	Adams	17 Paved St.	....
ObjID_Z	Acme	134 All St.	....

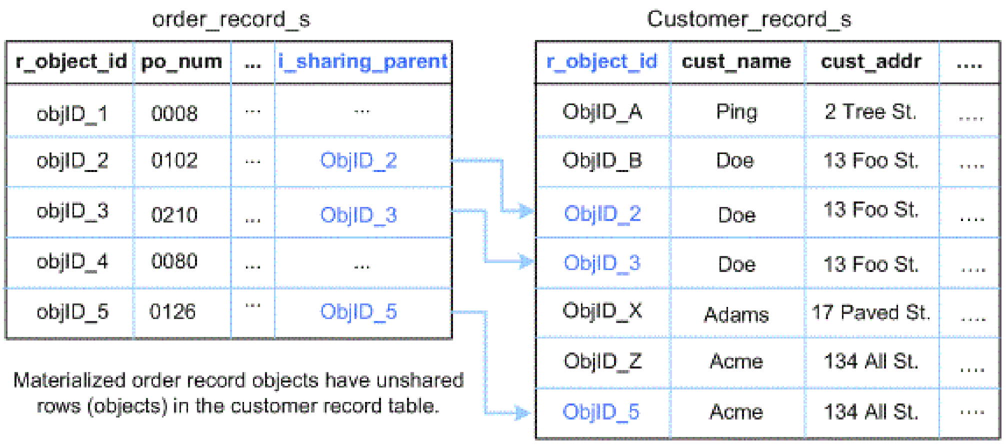
Unmaterialized order record objects share property values of customer record objects.

Figure 4-2: Default lightweight storage

The order record instances represented by objID_2 and objID_3 share the property values of the customer record instance represented by objID_B. Similarly, the order record object instance represented by objID_5 shares the property values of the customer record object instance represented by objID_Z. The `i_sharing_type` property for the parent, or shared, rows in `customer_record` are set to reflect the fact that those rows are shared.

There are no order record-specific rows created in `customer_record_s` for the unmaterialized order record objects.

The order record object type is defined for automatic materialization, and so certain operations on an instance will materialize the instance. This does not create a new order record instance, but instead creates a new row in the customer record table that is specific to the materialized order record instance. Figure 4-3 illustrates how a materialized instance is represented in the database tables.



r_object_id	po_num	...	i_sharing_parent
objID_1	0008	...	...
objID_2	0102	...	ObjID_2
objID_3	0210	...	ObjID_3
objID_4	0080	...	...
objID_5	0126	...	ObjID_5

r_object_id	cust_name	cust_addr	....
ObjID_A	Ping	2 Tree St.	....
ObjID_B	Doe	13 Foo St.	....
ObjID_2	Doe	13 Foo St.	....
ObjID_3	Doe	13 Foo St.	....
ObjID_X	Adams	17 Paved St.	....
ObjID_Z	Acme	134 All St.	....
ObjID_5	Acme	134 All St.	....

Materialized order record objects have unshared rows (objects) in the customer record table.

Figure 4-3: Materialized lightweight object storage

Materializing the order record instances created new rows in the `customer_record_s` table, one row for each order record object, and additional rows in each supertype table in the type hierarchy. The object ID of each customer record object representing a materialized order record object is set to the object ID of the order record object it represents, to associate the row with the order record object. Additionally, the `i_sharing_type` property of the previously shared customer record object is updated. In the order record objects, the `i_sharing_parent` property is reset to the object ID of the order record object itself.

For information about the identifiers recognized by Documentum CM Server, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.

4.3.1.5 Location and extent of object type tables

By default, all object types tables are created in the same tablespace with default extent sizes.

On some databases, you can change the defaults when you create the repository. By setting the `server.ini` file parameters before the initialization file is read during repository creation, you can define:

- The tablespaces in which to create the object-type tables
- The size of the extent allotted for system-defined object types

You can define tablespaces for the object type tables based on categories of size or for specific object types. For example, you can define separate tablespaces for the object types categorized as large and another space for those categorized as small. The category designations are based on the number of objects of the type expected to be included in the repository. Or, you can define a separate tablespace for the SysObject type and a different space for the user object type.

Additionally, you can change the size of the extent allotted to categories of object types or to specific object types.

For information about changing the default location and extents of object type tables and the locations of the index tables, see *OpenText Documentum Content Management - Server and Server Extensions Installation Guide (EDCSY250400-IGD)*.

4.3.2 Object type index tables

When a repository is created, the system creates a variety of indexes on the object type tables, including one on the `r_object_id` property for each _s object type table and one on the `r_object_id` and `i_position` for each _r object type table. The indexes are used for non-full-text DQL queries only, to enhance performance of property searches. Indexes are represented in the repository by objects of type `dmi_index`. The indexes are managed by the RDBMS.

By default, when you create a repository, the system puts the type index tables in the same tablespace as the object type tables. On certain operating systems, you can define an alternative location for the indexes during repository creation. Or, after the indexes are created, you can move them manually using the `MOVE_INDEX` administration method.

You can create additional indexes using the `MAKE_INDEX` administration method. Using `MAKE_INDEX` is recommended instead of creating indexes through the RDBMS server because Documentum CM Server uses the `dmi_index` table to determine which properties are indexed. The `MAKE_INDEX` method allows you to define the location of the new index.

You can remove user-defined indexes using the `DROP_INDEX` administration method. Dropping a system-defined index is not recommended.

The administration methods are available through Documentum Administrator, the DQL `EXECUTE` statement, or the `IDfSession.apply` method.

4.3.3 Content storage areas

The content files associated with SysObjects are part of a repository. They are stored in a file system directory, in a Centera host system, on an external storage device, or in the repository through a blob store or turbo storage area. All the files are represented in the repository by a content object, which itself identifies the storage area of the file. Content files in turbo or blob storage are stored directly in the repository. Content in turbo storage is stored in a property of the content object and subcontent objects. Content stored in blob storage is stored in a separate database table referenced by a blob store object.

For information about the description of the storage area options, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

4.4 Registered tables

In addition to the object type and type index tables, Documentum CM Server recognizes registered tables.

Registered tables are RDBMS tables that are not part of the repository but are known to Documentum CM Server. They are created by the DQL `REGISTER` statement and automatically linked to the System cabinet in the repository. They are represented in the repository by objects of type `dm_registered`.

After an RDBMS table is registered with the server, you can use DQL statements to query the information in the table or to add information to the table.

A number of views are created in a Documentum CM Server repository automatically. Some of these views are for internal use only, but some are available to provide information to users. The views that are available for viewing by users are defined as registered tables. To obtain a list of these views, you can run the following DQL query as a user with at least Sysadmin privileges:

```
SELECT "object_name", "table_name", "r_object_id" FROM "dm_registered"
```

For information about the `REGISTER` statement and querying registered tables, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

4.5 Data dictionary

The data dictionary is a collection of information about object types and their properties. The information is stored in internal data types and made visible to users and applications through the process of publishing the data.

The data dictionary is primarily for the use of client applications. Documentum CM Server stores and maintains the data dictionary information but only uses a small part-the default property values and the `ignore_immutable` values. The remainder of the information is for the use of client applications and users.

Applications can use data dictionary information to enforce business rules or provide assistance for users. For example, you can define a unique key constraint for an object type and applications can use that constraint to validate data entered by users. Or, you can define value assistance for a property. Value assistance returns a list of possible values that an application can then display to users as a list of choices for a dialog box field. You can also store error messages, help text, and labels for properties and object types in the data dictionary. All of this information is available to client applications.

4.5.1 Localization support

The data dictionary is the mechanism you can use to localize Documentum CM Server. The data dictionary supports multiple locales. A data dictionary locale represents a specific geographic region or linguistic group. For example, suppose your company has sites in Germany and England. Using the multi-locale support, you can store labels for object types and properties in German and English. Then, applications can query for the user current locale and display the appropriate labels on dialog boxes.

OpenText Documentum CM provides a default set of data dictionary information for each of the following locales:

- English
- French
- Italian
- Spanish
- German
- Japanese
- Korean

By default, when Documentum CM Server is installed, the data dictionary file for one of the locales is installed also. The procedure determines which of the default locales is most appropriate and installs that locale. The locale is identified in the `dd_locales` property of the `dm_docbase_config` object.

The data dictionary support for multiple locales lets you store a variety of text strings in the languages associated with the installed locales. For each locale, you can store labels for object types and properties, some help text, and error messages.

4.5.2 Modifying data dictionary

There are two kinds of modifications you can make to the data dictionary. You can:

- Install additional locales from the set of default locales provided with Documentum CM Server or install custom locales
- Modify the information in an installed locale by adding to the information, deleting the information, or changing the information

Some data dictionary information can be set using a text file that is read into the dictionary. You can also set data dictionary information when an object type is created or afterwards, using the `ALTER TYPE` statement.

For information about adding to or modifying the data dictionary, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

4.5.3 Publishing data dictionary

Data dictionary information is stored in repository objects that are not visible or available to users or applications. To make the data dictionary information available, it must be published. Publishing the data dictionary copies the information in the internal objects into three kinds of visible objects:

- `dd type info` objects

  A `dd type info` object contains the information specific to the object type in a specific locale.

- `dd attr info` objects

  A `dd attr info` object contains information specific to the property in a specific locale.

- `dd common` objects

  A `dd common info` object contains the information that applies to both the property and type level across all locales for a given object type or property.

For example, if a site has German and English locales installed, there will be two `dd type info` objects for each object type-one for the German locale and one for the English locale. Similarly, there will be two `dd attr info` objects for each property-one for the German locale and one for the English locale. However, there will be only one `dd common info` object for each object type and property because that object stores the information that is common across all locales.

Applications query the `dd common info`, `dd type info`, and `dd attr info` objects to retrieve and use data dictionary information. For information about publishing the data dictionary, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

4.5.4 Retrieving data dictionary information

You can retrieve data dictionary information using DQL queries or a Foundation Java API method.

Using DQL lets you obtain multiple data dictionary values in one query. However, the queries are run against the current `dmi_dd_type_info`, `dmi_dd_attr_info`, and `dmi_dd_common_info` objects. Consequently, a DQL query may not return the most current data dictionary information if there are unpublished changes in the information.

Neither DQL nor Foundation Java API queries return data dictionary information about new object types or added properties until that information is published, through an explicit `publishDataDictionary` method (in the `IDfSession` interface) or through the scheduled execution of the Data Dictionary Publisher job.

4.5.4.1 Using DQL

To retrieve data dictionary information using DQL, use a query against the object types that contain the published information. These types are `dd common info`, `dd type info`, and `dd attr info`. For example, the following query returns the labels for `dm_document` properties in the English locale:

```
SELECT "label_text" FROM "dmi_dd_attr_info"
WHERE "type_name"='dm_document' AND "nls_key"='en'
```

If you want to retrieve information for the locale that is the best match for the current client session locale, use the `DM_SESSION_DD_LOCALE` keyword in the query. For example:

```
SELECT "label_text" FROM "dmi_dd_attr_info"
WHERE "type_name"='dm_document' AND "nls_key"=DM_SESSION_DD_LOCALE
```

To ensure the query returns current data dictionary information, examine the `resync_needed` property. If that property is `TRUE`, the information is not current and you can republish before executing the query.

For information about the full description of the `DM_SESSION_DD_LOCALE` keyword, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

4.5.4.2 Using Foundation Java API

In Foundation Java API, data dictionary information is accessed through the `IDfSession.getTypeDescription` method. The method returns an `IDfTypedObject` object that contains the data dictionary information about an object type or a property.

The associated *Javadocs* contain information about using the `IDfSession.getTypeDescription` method.

4.6 Data dictionary contents

This section describes the kinds of information the data dictionary can contain.

4.6.1 Constraints

A constraint is a restriction applied to one or more property values for an instance of an object type. Documentum CM Server does not enforce constraints. The client application must enforce the constraint, using the constraint data dictionary definition. You can provide an error message as part of the constraint definition for the client application to display or log when the constraint is violated.

You can define a Check constraint in the data dictionary. Check constraints are most often used to provide data validation. You provide an expression or routine in the constraint definition that the client application can run to validate a given property value.

You can define a check constraint at either the object type or property level. If the constraint expression or routine references multiple properties, you must define the constraint at the type level. If it references a single property, you can define the constraint at either the property or type level.

You can define check constraints that apply only when objects of the type are in a particular lifecycle state.

4.6.2 Lifecycle states and default lifecycles for object types

You can define data dictionary information that applies to objects only when the objects are in a particular lifecycle state. As a document progresses through its lifecycle, the business requirements for the document are likely to change. For example, different version labels may be required at different states in the cycle. To control version labels, you can define value assistance to provide users with a list of valid version labels at each state of a document lifecycle. Or, you can define check constraints for each state, to ensure that users have entered the correct version label.

You can identify a default lifecycle for an object type and store that information in the data dictionary. If an object type has a default lifecycle, when a user creates an object of that type, the user can use the keyword `default` to identify the lifecycle when attaching the object to the lifecycle. There is no need to know the lifecycle object ID or name.

Note: Defining a default lifecycle for an object type does not mean that the default is attached to all instances of the type automatically. Users or applications must explicitly attach the default. Defining a default lifecycle for an object type provides an easy way for users to identify the default lifecycle for any particular type and a way to enforce business rules concerning the appropriate lifecycle for any particular object type. Also, it allows you to write an application that will not require revision if the default changes for an object type.

Defining a default lifecycle for an object type is performed using the `ALTER TYPE` statement.

The lifecycle defined as the default for an object type must be a lifecycle for which the type is defined as valid. Valid types for a lifecycle are defined by two properties in the `dm_policy` object that defines the lifecycle in the repository. The properties are `included_type` and `include_subtypes`. A type is valid for a lifecycle if:

- The type is named in `included_type`, or
- The `included_type` property references one of the type supertypes and `include_subtypes` is `TRUE`

4.6.3 Component specifications

Components are user-written routines. Component specifications designate a component as a valid routine to execute against instances of an object type. Components are represented in the repository by `dm_qual_comp` objects. They are identified in the data dictionary by their classifiers and the object ID of their associated `qual comp` objects.

A classifier is constructed of the `qual comp` `class_name` property and a acronym that represents the component build technology. For example, given a component whose `class_name` is `checkin` and whose build technology is Active X, its classifier is `checkin.ACX`.

You can specify only one component of each class for an object type.

4.6.4 Default values for properties

Documentum CM Server assigns the property a default property value when new objects of the type are created, unless the user explicitly sets the property value.

4.6.5 Value assistance

Value assistance provides a list of valid values for a property. A value assistance specification defines a literal list, a query, or a routine to list possible values for a property. Value assistance is typically used to provide a list of values for a property associated with a field in a dialog box.

4.6.6 Mapping information

Mapping information consists of a list of values that are mapped to another list of values. Mapping is generally used for repeating integer properties, to define understandable text for each integer value. Client applications can then display the text to users instead of the integer values.

For example, suppose an application includes a field that allows users to choose between four resort sites: Malibu, French Riviera, Cancun, and Florida Keys. In the repository, these sites may be identified by `integers-Malibu=1`, `French Riviera=2`, `Cancun=3`, and `Florida Keys=4`. Rather than display 1, 2, 3, and 4 to users, you can define mapping information in the data dictionary so that users see the text names of the resort areas, and their choices are mapped to the integer values for use the by application.

Chapter 5

Object type and instance manipulations and customizations

5.1 Object type manipulations

The OpenText Documentum CM object model is extensible. This extensibility provides users with the customized object types and properties needed to meet the particular requirements of their jobs. Documentum CM Server allows you to create new object types, alter some existing types, and drop custom types.

5.1.1 Creating new object types

You must have Create Type, Superuser, or Sysadmin privileges to create a new object type. With the appropriate user privileges, you can create a new type that is unrelated to any existing type in the repository, or you can create a subtype of any existing type that allows subtyping.

New object types are created using the `CREATE TYPE` statement. For information about `CREATE TYPE` and the object types that are supported, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

5.1.2 Altering object types

An object type definition includes its structure (the properties defined for the type) and several default values, such as the default storage area for content associated with objects of the type or the default ACL associated with the object type.

For system-defined object types, you cannot change the structure. You can only change the default values of some properties. If the object type is a custom type, you can change the structure and the default values. You can add properties, drop properties, or change the length definition of character string properties in custom object types.

Default aspects can be added to both system-defined object types and custom object types. An aspect is a code module associated with object instances. If you add a default aspect to an object type, that aspect is associated with each new instance of the type or its subtypes.

Object types are altered using the `ALTER TYPE` statement. You must be either the type owner or a superuser to alter a type.

The changes apply to the object type, the subtypes of the type, and all objects of the type and its subtypes.

For information about aspects and default aspects, see “Aspects” on page 73. For information about `ALTER TYPE` and the possible alterations that can be made to object types, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

5.1.3 Dropping object types

Dropping an object type removes its definition from the repository. It also removes any data dictionary objects for the type. Only user-defined types can be dropped from a repository. To drop a type, you must be the type owner or a superuser.

Documentum CM Server imposes the following restrictions on dropping types:

- No objects of the type can exist in the repository.
- The type cannot have any subtypes.

To drop a type, use the `DROP TYPE` statement. For more information about the `DROP TYPE` statement, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

5.2 Object instance manipulations

Documentum CM Server allows users and applications to create, modify, and destroy object instances, provided the user or application has the appropriate privileges or permissions. Objects can be created, manipulated, or destroyed using Foundation Java API methods or DQL statements.

5.2.1 Object creation

The ability to create objects is controlled by user privilege levels. Anyone can create documents and folders. To create a cabinet, a user must have the Create Cabinet privilege. To create users, the user must have the Sysadmin (System Administrator) privilege or the Superuser privilege. To create a group, a user must have Create Group, Sysadmin, or Superuser privileges. For information about user privilege levels and object-level permissions, see “User privileges” on page 86. For information about how to assign privileges, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

In Foundation Java API, the interface for each class of objects has a method that allows you to instantiate a new instance of the object.

In DQL, you use the `CREATE OBJECT` method to create a new instance of an object. For information about DQL and the reference information for the DQL statements, including `CREATE OBJECT`, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

5.2.2 Object modification

There are a variety of changes that users and applications can make to existing objects. The most common changes are:

- changing property values
- adding, modifying, or removing content
- changing the object access permissions
- associating the object with a workflow or lifecycle

In Foundation Java API, the methods that change property values are part of the interface that handles the particular object type. For example, to set the subject property of a document, you use a method in the `IDfSysObject` interface.

In Foundation Java API, methods are part of the interface for individual classes. Each interface has methods that are defined for the class plus the methods inherited from its superclass. The methods associated with a class can be applied to objects of the class. For information about the Foundation Java API and its classes and interfaces, see *OpenText Documentum Content Management - Foundation Java API Development Guide (EDCPKCL250400-DGD)* or the associated *Javadocs*.

The DQL is a superset of SQL. It allows you to query the repository tables and manipulate the objects in the repository. DQL has several statements that allow you to create objects. There are also DQL statements you can use to update objects by changing property values or adding content.

Creating or updating an object using DQL instead of the Foundation Java API is generally faster because DQL uses one statement to create or modify and then save the object. Using Foundation Java API methods, you must issue several methods-one to create or fetch the object, several to set its properties, and a method to save it.

5.2.3 Object destruction

Destroying an object removes it from the repository.

You must either be the owner of an object or you must have Delete permission on the object to destroy it. If the object is a cabinet, you must also have the Create Cabinet privilege.

Any SysObject or subtype must meet the following conditions before you can destroy it:

- The object cannot be locked.
- The object cannot be part of a frozen virtual document or snapshot.
- If the object is a cabinet, it must be empty.
- If the object is stored with a specified retention period, the retention period must have expired.

Destroying an object removes the object from the repository and also removes any relation objects that reference the object. Relation objects are objects that define a relationship between two objects. Only the explicit version is removed. Destroying an object does not remove other versions of the object. To remove multiple versions of an object, use a prune method. For more information about how the prune method behaves, see “Removing versions” on page 120. For information about relationships, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.

By default, destroying an object does not remove the object content file or content object that associated the content with the destroyed object. If the content was not shared with another document, the content file and content object are orphaned. To remove orphaned content files and orphaned content objects, run the `dmclean` and `dmfilescan` utilities as jobs, or manually. For information about the `dmclean` and `dmfilescan` jobs and how to execute the utilities manually, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

However, if the content file is stored in a storage area with digital shredding enabled and the content is not shared with another object, destroying the object also removes the content object from the repository and shreds the content file.

When the object you destroy is the original version, Documentum CM Server does not actually remove the object from the repository. Instead, it sets the object `i_is_deleted` property to `TRUE` and removes all associated objects, such as relation objects, from the repository. The server also removes the object from all cabinets or folders and places it in the Temp cabinet. If the object is carrying the symbolic label `CURRENT`, it moves that label to the version in the tree that has the highest `r_modify_date` property value. This is the version that has been modified most recently.

Note: If the object you want to destroy is a group, you can also use the DQL `DROP GROUP` statement.

5.3 Changing the object type of an object

The OpenText Documentum CM system gives you the ability to change the object type of an object, with some constraints. This feature is useful in repositories that have a lot of user-defined types and subtypes. For example, suppose your repository contains two user-defined document subtypes: working and published. The published type is a subtype of the working type with several additional properties. As a document moves through the writing, editing, and review cycle, it is a working document. However, as soon as it is published, you want to change its type to published. Documentum CM Server supports a type change of this kind. To make the change, you use the DQL `CHANGE...OBJECT[S]` statement. For information about the syntax and use of the `CHANGE OBJECT` statement, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

The change is subject to the following restrictions:

- The new type must have the same type identifier as the current type.

  A type identifier is a two-digit number that appears as the first two digits of an object ID. For example, the type identifier for all documents and document subtypes is 09. Consequently, the object ID for every document begins with 09.
- The new type must be either a subtype or supertype of the current type.

  This means that type changes cannot be lateral changes in the object hierarchy. For example, if two object types, A and B, are both direct subtypes of mybasetype, you cannot change an object of type A directly to type B.
- The object that you want to change cannot be immutable (unchangeable).

For more information about immutability and which objects are changeable, see “Changeable versions” on page 121.

The following illustration describes an example of a type hierarchy. In this example, you can change subtype_A to either baseSubtype1 or mybasetype. Similarly, you can change baseSubtype1 to either subtype_A or mybasetype, or mybasetype to either baseSubtype1 or baseSubtype2. However, you cannot change baseSubtype1 to baseSubtype2 or Subtype_B to Subtype_C because these types are peers on the hierarchy. Lateral changes are not allowed. Only vertical changes within the hierarchy are allowed.

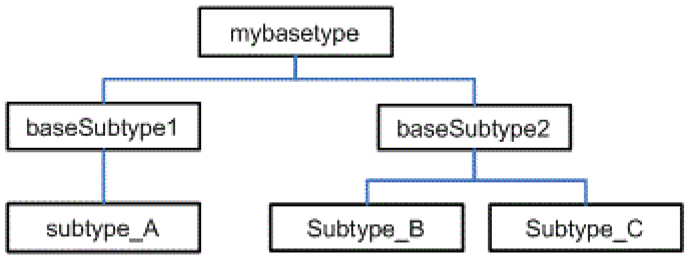


Figure 5-1: Change object sample hierarchy

5.4 Business object framework

The Business Object Framework (BOF) is a structured environment for developing content applications. The BOF is a feature of Foundation Java API that allows you to write code modules to customize or add behaviors in Foundation Java API. The customizations may be applied at the service, object type, or object level.

Using the business object framework to create customized modules provides the following benefits:

- The customizations are independent of the client applications, removing the need to code the customization into the client applications.
- The customizations can be used to extend core Documentum CM Server and Foundation Java API functionality.

- The customizations execute well in an application server environment.

To allow you to easily test modules in BOF development mode, Foundation Java API and Documentum CM Server support a development registry. This is a file that lists implementation classes to use during development. It loads the classes from the local classpath rather than being downloaded from a repository. For more information about using the BOF development mode, see *OpenText Documentum Content Management - Foundation Java API Development Guide (EDCPKCL250400-DGD)*.

5.4.1 BOF module

A BOF module is unit of executable business logic and its supporting material, such as third-party software or documentation. Foundation Java API supports four types of modules:

- Service-based modules (SBOs)
- Type-based modules (TBOs)
- Aspects
- Simple modules

An SBO provides functionality that is not specific to a particular object type or repository. For example, you might write an SBO that customizes the inbox.

A TBO provides functionality that is specific to an object type. For example, a TBO might be used to validate the title, subject, and keywords properties of a custom document subtype.

An aspect provides functionality that is applicable to specific objects. For example, you can use an aspect to set the value of a one property based on the value of another property.

A simple module is similar to an SBO, but provides functionality that is specific to a repository. For example, a simple module would be used if you wanted to customize a behavior that is different across repository versions.

A BOF module is comprised of the Java archive (JAR) files that contain the implementation classes and the interface classes for the behavior the module implements, and any interface classes on which the module depends. The module may also include Java libraries and documentation.

5.4.1.1 Module packaging and deployment

After you have created the files that comprise a module, you use Documentum Composer to package the module into a DAR file and install the DAR file to the appropriate repositories.

SBOs are installed in the repository that is the global registry. Simple modules, TBOs, and aspects are installed in each repository that contains the object type or objects whose behavior you want to modify.

Installing a BOF module creates a number of repository objects. The top-level object is a `dmc_module` object. Module objects are subtypes of `dm_folder`. They serve as a container for the BOF module. The properties of a module object provide information about the BOF module it represents. For example, they identify the module type (SBO, TBO, aspect, or simple), its implementation class, the interfaces it implements, and any modules on which the module depends.

The module folder object is placed in the repository in /System/Modules, under the appropriate subfolder. For example, if the module represents an TBO and its name is `MyTBO`, it is found in `/System/Modules/TBO/MyTBO`.

Each JAR file in the module is represented by a `dmc_jar` object. A JAR object has properties that identify the Java version level required by the classes in the module and whether the JAR file contains implementation or interface classes, or both.

The JAR objects representing the module implementation and interface classes are linked directly to the `dmc_module` folder. The jar objects representing the JAR files for supporting software are linked to folders represented by `dmc_java_library` objects. The `Java` library objects are then linked to the top-level module folder. The following illustration describes these relationships:

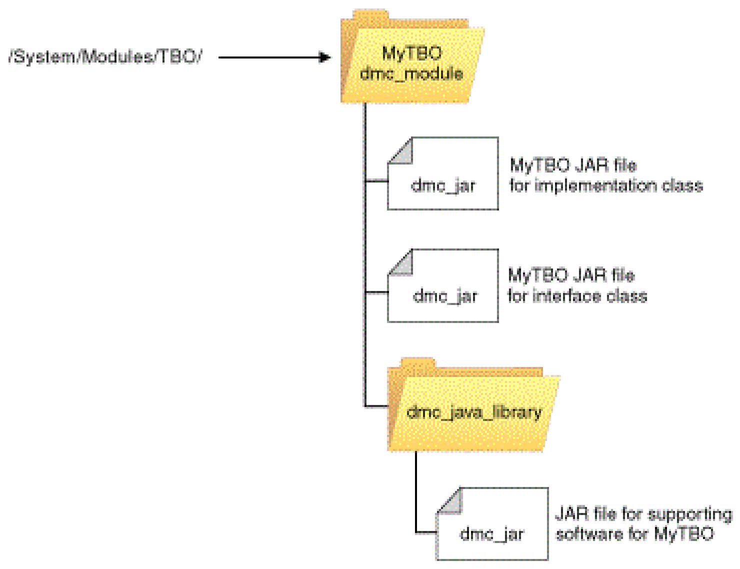


Figure 5-2: BOF module storage

The properties of a Java library object allow you to specify whether you want to sandbox the libraries linked to that folder. Sandboxing refers to loading the library into memory in a manner that makes it inaccessible to any application other than the application that loaded it. Foundation Java API achieves sandboxing by using a standard BOF class loader and separate class loaders for each module. The class loaders try to load classes first, before delegating to the usual hierarchy of Java class loaders.

In addition to installing the modules in a repository, you must also install the JAR file for a module interface classes on each client machine running Foundation Java API, and the file must be specified in the client `CLASSPATH` environment variable.

BOF modules are delivered dynamically to client applications when the module is needed. The delivery mechanism relies on local caching of modules, on client machines. Foundation Java API does not load TBOs, aspects, or simple modules into the cache until an application tries to use them. After a module is loaded, Foundation Java API checks for updates to the modules in the local cache whenever an application tries to use a module or after the interval specified by the `dfc.bof.cache.currency_check_interval` property in the `dfc.properties` file. The default interval value is 30 seconds. If a module has changed, only the changed parts are updated in the cache.

The location of the local cache is specified in the `dfc.properties` file, in the `dfc.data.cache_dir` property. The default value is the cache subdirectory of the directory specified in the `dfc.data.dir` property. All applications that use a particular Foundation Java API installation share the cache.

For information about how to create the BOF modules of the various types and how to set up and enable the BOF development mode, see *OpenText Documentum Content*

Management - Foundation Java API Development Guide (EDCPKCL250400-DGD). For information about instructions for packaging and deploying modules and information about deploying the interface classes to a client machine, see the *OpenText Documentum Composer* documentation.

5.4.2 Service-based objects

A service-based object (SBO) is a module that implements a service for multiple object types. For example, if you want to implement a service that automatically handles property validation for a variety of document subtypes, you would use an SBO. You can also use SBOs to implement utility functions to be called by TBOs or to retrieve items from external sources, for example, email messages.

An SBO associates an interface with an implementation class. SBOs are stored in the global registry, in a folder under /System/Modules/SBO. The name of the folder is the name of the SBO. The name of the SBO is typically the name of the interface.

5.4.3 Type-based objects

A type-based object (TBO) associates a custom object type that extends a OpenText Documentum CM system object type with an implementation class that extends the appropriate Foundation Java API class. For example, suppose you want to add some validation behavior to a specific document subtype. You would create a TBO for that subtype with an implementation class that extends `IDfDocument`, adding the validation behavior.

TBOs are specific to an object type, and so they are stored in each repository that contains the specified object type. They are stored in a folder under the `/System/Modules/TBO`. The folder name is the name of the TBO, which is typically the name of the object type for which it was created.

5.4.4 Aspects

An aspect is a BOF module that customizes behavior or records metadata or both for an instance of an object type.

You can attach an aspect to any object of type `dm_sysobject` or its subtypes. You can also attach an aspect to custom-type objects if the type has no supertype and you have issued an `ALTER TYPE` statement to modify the type to allow aspects.

An object can have multiple aspects attached, but can not have multiple instances of one aspect attached. That is, given object X and aspects a1, a2, and a3, you can attach a1, a2, and a3 to object X, but you cannot attach any of the aspects to object X more than once.

To attach instance-specific metadata to an object, you can define properties for an aspect.

For more information about the syntax and use of the `ALTER TYPE` statement, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

Note: Replication of objects with aspects is not supported.

5.4.4.1 Aspect properties

After you create an aspect, you can define properties for the aspect using Documentum Composer or the `ALTER ASPECT` statement. Properties of an aspect can be dropped or modified after they are added. Changes to an aspect that add, drop, or modify a property affect objects to which the aspect is currently attached.

Note: You cannot define properties for aspects whose names contain a dot (.). For example, if the aspect name is `com.mycompany.policy`, you can not define properties for that aspect.

Aspect properties are not fulltext-indexed by default. If you want to include the values in the index, you must use explicitly identify which properties you want indexed. You can use Documentum Composer or `ALTER ASPECT` to do this. For more information about the syntax and use of the `ALTER ASPECT` statement, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

5.4.4.2 Implementation of aspect properties

Aspect properties are stored in internal object types. When you define properties for an aspect, Documentum CM Server creates an internal object type that records the names and definitions of those properties. The name of the internal type is derived from the type object ID and is in the format: `dmi_`*`type_id`*. Documentum CM Server creates and manages these internal object types. The implementation of these types ensures that the properties they represent appear to client applications as standard properties of the object type to which the aspect is attached.

At the time you add properties to an aspect, you can choose to optimize performance for fetching or querying those properties by including the `OPTIMIZEFETCH` keyword in the `ALTER ASPECT` statement. That keyword directs Documentum CM Server to store all the aspect properties and their values in the property bag of any object to which the aspect is attached, if the object has a property bag.

5.4.4.3 Default aspects

Default aspects are aspects that are defined for a particular object type using the `ALTER TYPE` statement. If an object type has a default aspect, each time a user creates an instance of the object type or a subtype of the type, the aspects are attached to that instance.

An object type may have multiple default aspects. An object type inherits all the default aspects defined for its supertypes, and may also have one or more default aspects defined directly for itself. All of a type default aspects are applied to any instances of the type.

When you add a default aspect to a type, the newly added aspect is only associated with new instances of the type or subtype created after the addition. Existing instances of the type or its subtypes are not affected.

If you remove a default aspect from an object type, existing instances of the type or its subtypes are not affected. The aspect remains attached to the existing instances.

The `default_aspects` property in an object type `dmi_type_info` object records those default aspects defined directly for the object type. At runtime, when a type is referenced by a client application, Foundation Java API stores the type inherited and directly defined default aspects in memory. The in-memory cache is refreshed whenever the type definition in memory is refreshed.

5.4.5 Simple modules

A simple module customizes or adds a behavior that is specific to a repository version. For example, you may want to customize a workflow or lifecycle behavior that is different for different repository versions. A simple module is similar to an SBO, but does not implement the `IDfService` interface. Instead, it implements the `IDfModule` interface.

Simple modules associate an interface with an implementation class. They are stored in each repository to which they apply, and are stored in `/System/Modules`. The folder name is the name of the module.

Chapter 6

Security services

6.1 Overview

The security features supported by Documentum CM Server maintain system security and the integrity of the repository. They also provide accountability for user actions. Documentum CM Server supports:

- Standard security features
- Trusted Content Services security features

6.1.1 Standard security features

The following table lists the supported standard security features:

Feature	Description
User authentication	User authentication is the verification that the user is a valid repository user. User authentication occurs automatically, regardless of whether repository security is active. For more information, see "User authentication" on page 84.
Password encryption	Password encryption protects passwords stored in a file. Documentum CM Server automatically encrypts the passwords it uses to connect to third-party products, such as RDBMS, and the passwords used by internal jobs to connect to repositories. Documentum CM Server also supports encryption of other passwords through methods and a utility. For more information, see "Password encryption" on page 85.
Application-level control of SysObjects	Application-level control of SysObjects is an optional feature that you can use in client applications to ensure that only approved applications can handle particular documents or objects. For more information, see "Application-level control of SysObjects" on page 85.

Feature	Description
User privileges	User privileges define what special functions, if any, a user can perform in a repository. For example, a user with Create Cabinet user privileges can create cabinets in the repository. For more information, see "User privileges" on page 86.
Object-level permissions	Object-level permissions define which users and groups can access a SysObject and which level of access those users have. For more information, see "Object-level permissions" on page 88.
Table permits	Table permits are a set of permits applied only to registered tables, RDBMS tables that have been registered with Documentum CM Server. For more information, see "Table permits" on page 90.
Dynamic groups	Dynamic groups are groups whose membership can be controlled at runtime.
Access Control Lists (ACLs)	Object-level permissions are assigned using ACLs. Every SysObject in the repository has an ACL. The entries in the ACL define the access to the object. For more information, see "ACLs" on page 91.
Folder security	Folder security is an adjunct to repository security. For more information, see "Folder security" on page 91.
Auditing and tracing facilities	Auditing and tracing are optional features that you can use to monitor the activity in your repository. For more information, see "Auditing and tracing" on page 96.
Support for simple electronic signoffs and digital signatures	Documentum CM Server supports three options for electronic signatures. Support for simple signoffs, which use the IDfPersistentObject.signoff method, and for digital signatures, which is implemented using third-party software in a client application, are provided as standard features of Documentum CM Server. Support for the third option, using the IDfSysObject.addESignature method, is only available with Trusted Content Services, and is not available on the Linux operating system. For more information about all three options supporting signature requirements, see "Signature requirement support" on page 97.

Feature	Description
Secure Sockets Layer (SSL) communications between Documentum CM Server and the client library (DMCL) on client hosts	When you install Documentum CM Server, the installation procedure creates two service names for Documentum CM Server. One represents a native, nonsecure port and the other a secure port. You can then configure the server and clients, through the server config object and dmcl.ini files, to use the secure port.
Privileged DFC	This feature allows Foundation Java API to run under a privileged role, which gives escalated permissions or privileges for a specific operation. For more information, see “Privileged DFC” on page 104.

For information about users and groups, including dynamic groups, see “Users and groups” on page 80. For more information about setting the connection mode for servers and configuring clients to request a native or secure connection, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

6.1.2 Trusted Content Services security features

The following table lists the supported security features:

Feature	Description
Encrypted file store storage areas	Using encrypted file stores provides a way to ensure that content stored in a file store is not readable by users accessing it from the operating system. Encryption can be used on content in any format except rich media stored in a file store storage area. The storage area can be a standalone storage area or it can be a component of a distributed store. For more information, see “Encrypted file store storage areas” on page 107.
Digital shredding of content files	Digital shredding provides a final, complete way of removing content from a storage area by ensuring that deleted content files can not be recovered by any means. For more information, see “Digital shredding” on page 108.

Feature	Description
Electronic signature support using the `IDfSysObject.addESignature` method	The `addESignature` method is used to implement an electronic signature requirement through Documentum CM Server. The method creates a formal signature page and adds that page as primary content (or a rendition) to the signed document. The signature operation is audited, and each time a new signature is added, the previous signature is verified first. For more information, see "Signature requirement support" on page 97.

6.2 Repository security

The repository security setting controls whether object-level permissions, table permits, and folder security are enforced. The setting is recorded in the repository in the `security_mode` property in the docbase config object. The property is set to ACL, which turns on enforcement when a repository is created. Unless you have explicitly turned security off by setting `security_mode` to none, object-level permissions and table permits are always enforced.

6.3 Users and groups

Users and groups are the foundation of many of the security features. For example, users must be active users in a repository, satisfy user authentication and, after they establish repository sessions, must have appropriate object-level permissions to access documents and other SysObjects in the repository.

This section provides an overview of how users and groups are implemented.

6.3.1 Users

This section introduces repository users.

A repository user is an actual person or a virtual user who is defined as a user in the repository. A virtual user is a repository user who does not exist as an actual person.

Repository users have two states, active and inactive. An active user can connect to the repository and work. An inactive user is not allowed to connect to the repository.

6.3.1.1 Repository implementation of users

Users are represented in the repository as `dm_user` objects. The user object can represent an actual individual or a virtual person. The ability to define a virtual user as a repository user is a useful capability. For example, suppose you want an application to process certain user requests and want to dedicate an inbox to those requests. You can create a virtual user and register that user to receive events arising from the requests. The application can then read that user inbox to obtain and process the requests.

The properties of a user object record information that allows Documentum CM Server to manage the user access to the repository and to communicate with the user when necessary. For example, the properties define how the user is authenticated when the user requests repository access. They also record the user state (active or inactive) and the user email address allowing Documentum CM Server to send automated emails when needed, and the user home repository (if any).

For information about the properties defined for the `dm_user` object type, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.

6.3.1.2 Local and global users

In a federated distributed environment, a user is either a local user or a global user. A local user is managed from the context of the repository in which the user is defined. A global user is a user defined in all repositories participating in the federation and managed from the federation governing repository.

For more information about users in general and instructions about creating local users, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*. For more information about creating and managing global users, see *OpenText Documentum Content Management - Server and Server Extensions Installation Guide (EDCSY250400-IGD)*.

6.3.2 Groups

Groups are sets of users or groups or a mixture of both. They are used to assign permissions or client application roles to multiple users. There are several classes of groups in a repository. A group class is recorded in its `group_class` property. For example, if `group_class` is `group`, the group is a standard group, used to assign permissions to users and other groups.

A group, similar to an individual user, can own objects, including other groups. A member of a group that owns an object or group can manipulate the object just as an individual owner. The group member can modify or delete the object.

Additionally, a group can be a dynamic group. Membership in a dynamic group is determined at runtime. Dynamic groups provide a layer of security by allowing you to control dynamically who Documentum CM Server treats as a member of a group.

Several types of groups are:

- Standard groups:

  A standard group consists of a set of users. The users can be individual users or other groups or both. A standard group is used to assign object-level permissions to all members of the group. For example, you might set up a group called `engr` and assign Version permission to the `engr` group in an ACL applied to all engineering documents. All members of the `engr` group then have Version permission on the engineering documents.

  Standard groups can be public or private. When a group is created by a user with Sysadmin or Superuser privileges, the group is public by default. If a user with Create Group privileges creates the group, it is private by default. You can override these defaults after a group is created using the `ALTER GROUP` statement. For information about how to use the `ALTER GROUP` statement, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

- Role groups:

  A role group contains a set of users or other groups or both that are assigned a particular role within a client application domain. A role group is created by setting the `group_class` property to role and the `group_name` property to the role name.

- Module role groups:

  A module role group is a role group that is used by an installed BOF module. It represents a role assigned to a module of code, rather than a particular user or group. Module role groups are used internally. The `group_class` value for these groups is module role.

- Privileged groups:

  A privileged group is a group whose members are allowed to perform privileged operations even though the members do not have the privileges as individuals. A privileged group has a `group_class` value of privilege group.

- Domain groups:

  A domain group represents a particular client application domain. A domain group contains a set of role groups corresponding to the roles recognized by the client application.

- Dynamic groups:

  A dynamic group is a group, of any group class, with a list of potential members. A setting in the group definition defines whether the potential members are treated as members of the group or not when a repository session is started. Depending on that setting, an application can issue a session call to add or remove a user from the group when the session starts.

  A non-dynamic group cannot have a dynamic group as a member. A dynamic group can include other dynamic groups as members or non-dynamic groups as members. However, if a non-dynamic group is a member, the members of the non-dynamic group are treated as potential members of the dynamic group.

- Local and global groups:

  In a federated distributed environment, a group is either a local group or a global group. A local group is managed from the context of the repository in which the group is defined. A global group is a group defined in all repositories participating in the federation and managed from the federation governing repository. For information about creating local groups, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*. For information about creating and managing global groups, see *OpenText Documentum Content Management - Server and Server Extensions Installation Guide (EDCSY250400-IGD)*.

Role and domain groups are used by client applications to implement roles within an application. The two kinds of groups are used together to achieve role-based functionality. Documentum CM Server does not enforce client application roles.

For example, suppose you write a client application called `report_generator` that recognizes three roles: readers (users who read reports), writers (users who write and generate reports), and administrators (users who administer the application). To support the roles, you create three role groups, one for each role. The `group_class` is set to role for these groups and the group names are the names of the roles: readers, writers, and administrators. Then, create a domain group by creating a group whose `group_class` is domain and whose group name is the name of the domain. In this case, the domain name is `report_generator`. The three role groups are the members of the `report_generator` domain group.

When a user starts the `report_generator` application, the application examines its associated domain group and determines the role group to which the user belongs. The application then performs only the actions allowed for members of that role group. For example, the application customizes the menus presented to the user depending on the role to which the user is assigned.

Note: Documentum CM Server does not enforce client application roles. It is the responsibility of the client application to determine if there are role groups defined for the application and apply and enforce any customizations based on those roles.

For more information about groups, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

6.4 User authentication

User authentication is the procedure by which Documentum CM Server ensures that a particular user is an active and valid user in a repository.

Documentum CM Server authenticates the user whenever a user or application attempts to open a repository connection or reestablish a timed-out connection. The server checks that the user is a valid, active repository user. If not, the connection is not allowed. If the user is a valid, active repository user, Documentum CM Server authenticates the user name and password.

Users are also authenticated when they:

- Assume an existing connection
- Change their password
- Perform an operation that requires authentication before proceeding
- Sign-off an object electronically

Documentum CM Server supports a variety of mechanisms for user authentication, including authentication against the operating system, against OpenText Directory Services (OTDS), using a plug-in module, or using a password stored in the repository.

There are several ways to configure user authentication, depending on your choice of authentication mechanism. For example, if you are authenticating against the operating system, you can write and install your own password checking program. If you use a plug-in module, you can use the module provided with Documentum CM Server or write and install a custom module.

To protect the repository, you can enable a feature that limits the number of failed authentication attempts. If the feature is enabled and a user exceeds the limit, the user account is deactivated in the repository.

For more information about user authentication options and procedures for implementing them, and setting up authentication failure limits, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

Documentum CM Server tracks software usage by recording login times. The Documentum CM Server global registry contains a record of the first and the latest login time for each user of each application that connects to Documentum CM Server. Documentum CM Server periodically generates basic reports to indicate usage. These reports are available to the Documentum CM Server administrator. For more information about usage tracking reports, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

6.5 Password encryption

Password encryption is the automatic process used by Documentum CM Server to protect certain passwords.

The passwords used by Documentum CM Server to connect to third-party products, such as RDBMS, as well as those used by many internal jobs to connect to a repository, are stored in files in the installation. To protect these passwords, Documentum CM Server automatically encrypts them. Decrypting the passwords occurs automatically also. When an encrypted password is passed as an argument to a method, the Foundation Java API decrypts the password before passing the arguments to Documentum CM Server.

OpenText Documentum CM is integrated with HashiCorp Vault to store and retrieve all secrets automatically from Vault. You must configure Documentum Secret Integration Service (DSIS) to use Vault and make sure that DSIS is running. For more information about configuring DSIS, see *OpenText Documentum Content Management - Server and Server Extensions Installation Guide (EDCSY250400-IGD)*.

Client applications can use password encryption for their own password by using the Foundation Java API method `IDfClient.encryptPassword`. The method allows you to use encryption in your applications and scripts. Use `encryptPassword` to encrypt passwords used to connect to a repository. All the methods that accept a repository password accept a password encrypted using the `encryptPassword` method. The Foundation Java API will automatically perform the decryption.

Passwords are encrypted using the Administration Encryption Key (AEK). The AEK is installed during Documentum CM Server installation. After encrypting a password, Documentum CM Server also encodes the encrypted string using Base64 before storing the result in the appropriate password file. The final string is longer than the clear text source password.

For information about administering password encryption, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*. The associated *Javadocs* contain more information about `encryptPassword`.

6.6 Application-level control of SysObjects

In some business environments, such as regulated environments, it is essential that some documents and other SysObjects be manipulated only by approved client applications. Application-level control of SysObjects is a Documentum CM Server feature that allows client applications to assert ownership of particular objects and, consequently, prohibit users from modifying or manipulating those objects through other applications.

Application-level control is independent of repository security. Even if repository security is turned off, client applications can still enforce application-level control of objects. Application-level control, if implemented, is enforced on all users except superusers. Application-level control is implemented through application codes.

Each application that requires control over the objects it manipulates has an application code. The codes are used to identify which application has control of an object and to identify which controlled objects can be accessed from a particular client.

An application sets an object `a_controlling_app` property to its application code to identify the object as belonging to the application. When set, the property can only be modified by that application or another that knows the application code.

To identify to the system which objects it can modify, an application sets the `dfc.application_code` key in the client config object or the `application_code` property in the session config object when the application is started. Setting the property in the client config object, rather than the session config object, provides performance benefits, but affects all sessions started through that Foundation Java API instance. The key and the property are repeating. On start-up, an application can add multiple entries for the key or set the property to multiple application codes if users are allowed to modify objects controlled by multiple applications through that particular application.

When a non-superuser accesses an object, Documentum CM Server examines the object `a_controlling_app` property. If the property has no value, the user access is determined solely by ACL permissions. If the property has a value, Documentum CM Server compares the value to the values in the session `application_code` property. If a match is found, the user is allowed to access the object at the level permitted by the object ACL. If a match is not found, Documentum CM Server examines the `default_app_permit` property in the docbase config object. The user is granted access to the object at the level defined in that property (Read permission by default) or at the level defined by the object ACL, whichever is the more restrictive. Additionally, if a match is not found, the user is never allowed extended permissions on the object, regardless of the permission provided by the default repository setting or the ACL.

6.7 User privileges

Documentum CM Server supports a set of user privileges that determine what special operations a user can perform in the repository. There are two types of user privileges: basic and extended. The basic privileges define the operations that a user can perform on SysObjects in the repository. The extended privileges define the security-related operations the user can perform.

User privileges are always enforced whether repository security is turned on or not.

6.7.1 Basic user privileges

The following table lists the basic user privileges:

Level	Name	Description
0	None	User has no special privileges.
1	Create Type	User can create object types.
2	Create Cabinet	User can create cabinets.
4	Create Group	User can create groups.
8	Sysadmin	User has System Administration privileges.
16	Superuser	User has Superuser privileges.

The basic user privileges are additive, not hierarchical. For example, granting Create Group to a user does not give the user Create Cabinet or Create Type privileges. If you want a user to have both privileges, you must explicitly give the user both privileges.

Typically, the majority of users in a repository have None as their privilege level. Some users, depending on their job function, will have one or more of the higher privileges. A few users will have either Sysadmin or Superuser privileges.

User privileges do not override object-level permissions when repository security is turned on. However, a superuser always has at least Read permission on any object and can change the object-level permissions assigned to any object.

Applications and methods that are executed with Documentum CM Server as the server always have Superuser privileges.

6.7.2 Extended user privileges

The following table lists the extended user privileges:

Level	Name	Description
8	Config Audit	User can execute the methods to start and stop auditing.
16	Purge Audit	User can remove audit trail entries from the repository.
32	View Audit	User can view audit trail entries.

The extended user privileges are not hierarchical. For example, granting a user Purge Audit privilege does not confer Config Audit privilege also.

Repository owners, superusers, and users with the View Audit permission can view all audit trail entries. Other users in a repository can view only those audit trail entries that record information about objects other than ACLs, groups, and users.

Only repository owners and superusers can grant and revoke extended user privileges, but they can not grant or revoke these privileges for themselves.

For information about assigning privileges, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

6.8 Object-level permissions

Object-level permissions are access permissions assigned to every SysObject (and SysObject subtype) in the repository. They are defined as entries in ACL objects. The entries in the ACL identify users and groups and define their object-level permissions to the object with which the ACL is associated.

Each SysObject (or SysObject subtype) object has an associated ACL. For most sysObject subtypes, the permissions control the access to the object. For `dm_folder`, however, the permissions are not used to control access unless folder security is enabled. In such cases, the permissions are used to control specific sorts of access, such as the ability to link a document to the folder.

For more information, see “ACLs” on page 91. For more information, see “Folder security” on page 91. The associated *Javadocs* for the `IDfSysObject.link` and `IDfSysObject.unlink` methods contain a description of privileges necessary to link or unlink an object.

There are two kinds of object-level permissions: base permissions and extended permissions.

6.8.1 Base object-level permissions

The following table lists the base permissions:

Level	Permission	Description
1	None	No access is permitted.
2	Browse	The user can look at property values, but not at associated content.
3	Read	The user can read content, but not update it.
4	Relate	The user can attach an annotation to the object.
5	Version	The user can version the object, but cannot overwrite the existing version.

Level	Permission	Description
6	Write	The user can write and update the object. Write permission confers the ability to overwrite the existing version.
7	Delete	The user can delete the object.

These permissions are hierarchical. For example, a user with Version permission also has the access accompanying Read and Browse permissions. Or, a user with Write permission also has the access accompanying Version permission.

6.8.2 Extended object-level permissions

The following table lists the extended permissions:

Permission	Description
Change Location	In conjunction with the appropriate base object-level permissions, allows the user to move an object from one folder to another. All users having at least Browse permission on an object are granted Change Location permission by default for that object. **Note:** Browse permission is not adequate to move an object.
Change Ownership	The user can change the owner of the object.
Change Permission	The user can change the basic permissions of the object.
Change State	The user can change the document lifecycle state of the object.
Delete Object	The user can delete the object. The delete object extended permission is not equivalent to the base Delete permission. Delete Object extended permission does not grant Browse, Read, Relate, Version, or Write permission.
Execute Procedure	The user can run the external procedure associated with the object. All users having at least Browse permission on an object are granted Execute Procedure permission by default for that object.

Permission	Description
Change Folder Links	Allows a user to link an object to a folder or unlink an object from a folder. The permission must be defined in the ACL associated with the folder.

The extended permissions are not hierarchical. You must assign each explicitly.

6.8.3 Default permissions

Object owners, because they have Delete permission on the objects they own by default, also have Change Location and Execute Procedure permissions on those objects. By default, superusers have Read permission and all extended permissions except Delete Object and change folder links on any object.

6.9 Table permits

The table permits control access to the RDBMS tables represented by registered tables in the repository. Table permits are only enforced when repository security is on. To access an RDBMS table using DQL, you must have:

- At least Browse access for the `dm_registered` object representing the RDBMS table
- The appropriate table permit for the operation that you want to perform

Note: Superusers can access all RDBMS tables in the database using a `SELECT` statement regardless of whether the table is registered or not.

The following table lists the levels of table permits:

Level	Permit	Description
0	None	No access is permitted
1	Select	The user can retrieve data from the table.
2	Update	The user can update existing data in the table.
4	Insert	The user can insert new data into the table.
8	Delete	The user can delete rows from the table.

The permits are identified in the `dm_registered` object that represents the table, in the `owner_table_permit`, `group_table_permit`, and `world_table_permit` properties.

The permits are not hierarchical. For example, assigning the permit to insert does not confer the permit to update. To assign more than one permit, you add the integers representing the permits you want to assign, and set the appropriate property to the total. For example, if you want to assign both insert and update privileges as the group table permit, set the `group_table_permit` property to `6`, the sum of the integer values for the update and insert privileges.

6.10 Folder security

Folder security is a supplemental level of repository security. When folder security is turned on, for some operations the server checks and applies permissions defined in the ACL associated with the folder in which an object is stored or on the primary folder of the object. These checks are in addition to the standard object-level permission checks associated with the object ACL. In new repositories, folder security is turned on by default.

Folder security does not prevent users from working with objects in a folder. It provides an extra layer of security for operations that involve linking or unlinking, such as creating a new object, moving an object, deleting an object, and copying an object.

Folder security is turned on and off at the repository level, using the `folder_security` property in the docbase config object.

For more information about folder security, including a list of the extra checks it imposes, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

6.11 ACLs

An Access Control List (ACL) is the mechanism that Documentum CM Server uses to impose object-level permissions on SysObjects. An ACL has one or more entries that identify a user or group and the object-level permissions accorded that user or group by the ACL. Another name for an ACL is a permission set. An ACL is a set of permissions that apply to an object.

Each SysObject has an ACL. The ACL assigned to a SysObject is used to control access to that object. For folders, the assigned ACL serves additional functions. If folder security is enabled, the ACL assigned to the folder sets the folder security permissions. If the default ACL for the Documentum CM Server is configured as `Folder`, then newly created objects in the folder are assigned the folder ACL.

An ACL is represented in the repository as an object of type `dm_acl`. ACL entries are recorded in repeating properties in the object. Each ACL is uniquely identified within the repository by its name and domain. The domain represents the owner of the ACL. When an ACL is assigned to an object, the object `acl_name` and `acl_domain` properties are set to the name and domain of the ACL.

After an ACL is assigned to an object, the ACL can be changed. You can modify the ACL itself or you can remove it and assign a different ACL to the object.

ACLs are typically created and managed using Documentum Administrator. However, you can also create and manage them through Foundation Java API or DQL.

6.11.1 ACL entries

Documentum CM Server enforces all ACL entries that determine which users and groups can access the object and the level of access for each. There are several types of ACL entries:

- `AccessPermit` and `ExtendedPermit`
- `AccessRestriction` and `ExtendedRestriction`
- `RequiredGroup` and `RequiredGroupSet`
- `ApplicationPermit` and `ApplicationRestriction`

`AccessPermit` and `ExtendedPermit` entries grant the base and extended permissions. Documentum CM Server supports creating, modifying, or deleting `AccessPermit` and `ExtendedPermit` entries.

The remaining entry types provide extended capabilities for defining access. For example, an `AccessRestriction` entry restricts a user or group access to a specified level even if that user or group is granted a higher level by another entry. You can create, modify, or delete any entry other than an `AccessPermit` or `ExtendedPermit` entry.

For more information about descriptions of the type of entries you can place in an ACL and instructions for creating ACLs, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*. For more information about the options for assigning ACLs to objects, see “Assigning ACLs” on page 143.

6.11.2 Categories of ACLs

ACLs are either external or internal ACLs:

- External ACLs are created explicitly by users. The name of an external ACL is determined by the user. External ACLs are managed by users, either the user who creates them or superusers.
- Internal ACLs are created by Documentum CM Server. Internal ACLs are created in a variety of situations. For example, if a user creates a document and grants access to the document to HenryJ, Documentum CM Server assigns an internal ACL to the document. The internal ACL is derived from the default ACL with the addition of the permission granted to HenryJ. The names of internal ACL begin with `dm_`. Internal ACLs are managed by Documentum CM Server.

The external and internal ACLs are further characterized as public or private ACLs:

- Public ACLs are available for use by any user in the repository. Public ACLs created by the repository owner are called system ACLs. System ACLs can only be managed by the repository owner. Other public ACLs can be managed by their owners or a user with Sysadmin or Superuser privileges.

- Private ACLs are created and owned by a user other than the repository owner. However, unlike public ACLs, private ACLs are available for use only by their owners, and only their owners or a superuser can manage them.

6.11.3 Template ACLs

A template ACL is an ACL that can be used in many contexts. Template ACLs use aliases in place of user or group names in the entries. The aliases are resolved when the ACL is assigned to an object. A template ACL allows you to create one ACL that you can use in a variety of contexts and applications and ensure that the permissions are given to the appropriate users and groups. For more information, see "Aliases" on page 229.

6.11.4 Mandatory Access Control

The required group implies that at run time, the currently connected user has to belong to all the required groups, before the permission of the user are evaluated based on the permission set on the object. If the user belongs to all the required groups, the required group set evaluates if the user is a member of at least one of the groups listed in the required group set. The user is then evaluated if there are currently any restrictions imposed on the user. This is the core function of Mandatory Access Control (MACL).

MACL has the following ACL entries:

- Access Restrictions
- Required Groups
- Required Group Set
- Application Permit
- Application Restrictions

6.11.4.1 Access Restrictions

You can see how Access Restrictions work to deny the specified permission.

An Access Restriction entry:

- Indicates that the specified user (or group) can not have a particular permission.
- Similar in behavior to Windows allow/deny paradigm.
- Does not imply any permission but simply denies a permit.

6.11.4.2 Required Groups

- A Required Group entry specifies that the user must belong in that group to satisfy any security check.
- If an ACL has multiple Required Group entries, then the user must belong to all of the groups.
- The accessor_name for a Required Group must be a group name rather than a user name.

6.11.4.3 Required Group Set

- If an ACL contains a Required Group Set entry or entries then a user must belong to at least one of the groups in a Required Group Set in order to satisfy any security check in the schema.
- Only one set of required groups may be added to an ACL.

6.11.4.4 Application Permissions and Restrictions

- Provides the ability to extend our security model by storing application specific permissions (or restrictions) in an ACL.
- Freeform string(128) field
- Documentum CM Server does not understand or use application permissions, it only stores them.
- Examples are print permission and export permission.

6.11.4.5 Dynamic Groups

- Dynamic Groups are organized and managed similar to other groups but for security checks and queries, a member is not necessarily treated as being in or out of this group simply by persistent group membership.
- A Dynamic Group has a membership (similar to a normal group) but that membership defines potential members of the group.
- In order for the Documentum CM Server to treat the member as part of the group, the user must request to be included through a session call.
- A Dynamic Group can declare its membership by default so that its users is treated as in the group unless they ask to be removed.

For example:

- Security Labels: Top Secret group requires member to be inside the firewall.

  If application detects that user is inside the firewall, application adds the user to the dynamic group for that session.
- Roles-based security can be implemented by Dynamic Groups

  Document permissions can be granted to Roles and application can decide when to place the session user under that role, giving the user access to that role’s data.

6.11.4.5.1 Dynamic Groups auditing

- Ability to audit when a user is added or removed from a dynamic group at runtime.
- Two auditable events: `dm_add_dynamic_group` and `dm_remove_dynamic_group`
- Two attributes to `dm_audittrail_group`:
  - `is_dynamic`: Boolean, non-repeating
  - `is_dynamic_default`: Boolean, non-repeating

6.11.4.6 Extended Permission

- Extended permission: `DELETE_OBJECT`
- Two ways to grant delete permission:
  - Normal hierarchical permissions (`DELETE`)
  - Extended Permission (`DELETE_OBJECT`)
- Adding `DELETE` through the standard hierarchical permissions also grants all lower permissions.
- Adding `DELETE_OBJECT` through the extended permissions only grants delete without any other permissions.
- At least `BROWSE` hierarchical permission is also needed.

6.11.4.7 Configuration for MACL

- MACL functionality is enabled when you set the value of `macl_security_disabled` to `FALSE`.
- After an ACL is created with MACL rules, the rules are evaluated depending on the value of the `macl_security_disabled` property.
- Internal applications have special mechanism to create ACLs with MACL rules regardless of Trusted Content Services.
- `DELETE_OBJECT` extended permission can be used without Trusted Content Services.

6.12 Auditing and tracing

Auditing and tracing are two security tools that you can use to track operations in the repository.

6.12.1 Auditing

Auditing is the process of recording the occurrence of system and application events in the repository. Events are operations performed on objects in a repository or something that happens in an application. System events are events that Documentum CM Server recognizes and can audit. Application events are user-defined events. They are not recognized by Documentum CM Server and must be audited by an application.

Documentum CM Server audits a large set of events by default. For example, all successful `addESignature` events and failed attempts to execute `addESignature` events are audited. Similarly, all executions of methods that register or unregister events for auditing are themselves audited.

You can also audit many other operations. For example, you can audit:

- All occurrences of an event on a particular object or object type
- All occurrences of a particular event, regardless of the object on which it occurs
- All workflow-related events
- All occurrences of a particular workflow event for all workflows started from a given process definition
- All executions of a particular job

There are several methods in the `IDfAuditTrailManager` interface that can be used to request auditing. For example, the `registerEventForType` method starts auditing a particular event for all objects of a specified type. Typically, you must identify the event you want to audit and the target of the audit. The event can be either a system event or an application (user-defined) event. The target can be a particular object, all objects of a particular object type, or objects that satisfy a particular query.

The audit request is stored in the repository in registry objects. Each registry object represents one audit request.

Issuing an audit request for a system event initiates auditing for the event. If the event is an application event, the application is responsible for checking the registry objects to determine whether auditing is requested for the event and, if so, create the audit trail entry.

Users must have Config Audit privileges to issue an audit request.

The records of audited events are stored in the repository as entries in an audit trail. The entries are objects of `dm_audittrail`, `dm_audittrail_acl`, or `dm_audittrail_group`. Each entry records the information about one occurrence of an event. The information is specific to the event and can include information about property values in the audited object.

For information about auditing, including a list of those events that are audited by default, how to initiate auditing, and what information is stored in an audit trail record, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

6.12.2 Tracing

Tracing is an feature that logs information about operations that occur in Documentum CM Server and Foundation Java API. The information that is logged depends on which tracing functionality is turned on.

Documentum CM Server and Foundation Java API support multiple tracing facilities. On Documentum CM Server, you can turn on tracing for a variety of server features, such as storage area operation, and operations on SysObjects. The jobs in the administration tool suite also generate trace files for their operations.

Foundation Java API has a robust tracing facility that allows you to trace method operations and RPC calls. The facility allows you to configure many options for the generated trace files. For example, you can trace by user or thread, specify stack depth to be traced, and define the format of the trace file.

For information about the `SET_OPTIONS` and `MODIFY_TRACE` administration methods, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*. For information about all the jobs in the administration tool suite, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

6.13 Signature requirement support

Many business processes have signature requirements for one or more steps in a process. Similarly, some lifecycle states can require a signature before an object can move to the next state. For example, a budget request can need an approval signature before the money is disbursed. Users can be required to sign standard operating procedures (SOPs) to indicate that they have read the procedures. Or, a document can require an approval signature before the document is published on a website.

Documentum CM Server supports signature requirements with the following options:

- "Electronic signatures" on page 98
- "Digital signatures" on page 103
- "Simple sign-offs" on page 103

Electronic signatures are generated and managed by Documentum CM Server. The feature is supported by two methods: `IDfSysObject.addESignature` and

`IDfSysObject.verifyESignature`. Use this option if you require a rigorous signature implementation to meet regulatory requirements. You must have the appropriate license to use this option.

Digital signatures are electronic signatures in formats such as PKCS #7, XML signature, or PDF signature. Digital signatures are generated by third-party products called when an `addDigitalSignature` method is executed. Use this option if you want to implement strict signature support in a client application.

Simple sign-offs are the least rigorous way to supply an electronic signature. Simple sign-offs are implemented using the `IDfPersistentObject.signoff` method. This method authenticates a user signing off a document and creates an audit trail entry for the `dm_signoff` event.

6.13.1 Electronic signatures

An electronic signature is a signature recorded in formal signature page generated by Documentum CM Server and stored as part of the content of the object. Electronic signatures are generated when an application issues an `IDfSysObject.addESignature` method.

Electronic signatures are the most rigorous signature requirement that Documentum CM Server supports.

6.13.1.1 Overview of Implementation

Electronic signatures are generated by Documentum CM Server when an application or user issues an `addESignature` method. Signatures generated by `addESignature` are recorded in a formal signature page and added to the content of the signed object. The method is audited automatically, and the resulting audit trail entry is signed by Documentum CM Server. The auditing feature cannot be turned off. If an object requires multiple signatures, before allowing the addition of a signature, Documentum CM Server verifies the preceding signature. Documentum CM Server also authenticates the user signing the object.

All the work of generating the signature page and handling the content is performed by Documentum CM Server. The client application is only responsible for recognizing the signature event and issuing the `addESignature` method. A typical sequence of operations in an application using the feature is:

1. A signature event occurs and is recognized by the application as a signature event.

   A signature event is an event that requires an electronic signature on the object that participated in the event. For example, a document check-in or lifecycle promotion might be a signature event.

2. In response, the application asks the user to enter a password and, optionally, choose or enter a justification for the signature.

3. After the user enters a justification, the application can call the `createAudit` method to create an audit trail entry for the event.

This step is optional, but auditing the event that triggered the signature is common.

4. The application calls `addESignature` to generate the electronic signature.

After `addESignature` is called, Documentum CM Server performs all the operations required to generate the signature page, create the audit trail entries, and store the signature page in the repository with the object. You can add multiple signatures to any particular version of a document. The maximum number of allowed signatures on a document version is configurable.

Electronic signatures require a template signature page and a method (stored in a `dm_method` object) to generate signature pages using the template. The OpenText Documentum CM system provides a default signature page template and signature generation method that can be used on documents in PDF format or documents that have a PDF rendition. You can customize the electronic signature support in a variety of ways. For example, you can customize the default template signature page, create your own template signature page, or provide a custom signature creation method for use with a custom template.

6.13.1.2 addESignature method

When an application or user issues an `IDfSysObject.addESignature` method, Documentum CM Server performs the following operations:

1. Authenticates the user and verifies that the user has at least Relate permission on the document to be signed.

   If a user name is passed in the `addESignature` method arguments, that user must be the same as the session user issuing the `addESignature` method.
2. Verifies that the document is not checked out.

   A checked out document cannot be signed by `addESignature`.
3. Verifies that the `pre_signature` hash argument, if any, in the method, matches a hash of the content in the repository.
4. If the content has been previously signed, the server:

   - Retrieves all the audit trail entries for the previous `dm_addesignature` events on this content.
   - Verifies that the most recent audit trail entry is signed (by Documentum CM Server) and that the signature is valid.
   - Verifies that the entries have consecutive signature numbers.
   - Verifies that the hash in the audit trail entry matches the hash of the document content.
5. Copies the content to be signed to a temporary directory location and calls the signature creation method. The signature creation method:

   - Generates the signature page using the signature page template and adds the page to the content.

- Replaces the content in the temporary location with the signed content.

6. If the signature creation method returns successfully, the server replaces the original content in the repository with the signed copy.

   If the signature is the first signature applied to that particular version of the document, Documentum CM Server appends the original, unsigned content to the document as a rendition with the page modifier set to `dm_sig_source`.

7. Creates the audit trail entry recording the `dm_addesignature` event.

   The entry also includes a hash of the newly signed content.

You can trace the operations of `addESignature` and the called signature creation method.

The OpenText Documentum CM system provides a default signature page template and a default signature creation method with Documentum CM Server so you can use the electronic signature feature with no additional configuration. The only requirement for using the default functionality is that documents to be signed must be in PDF format or have a PDF rendition associated with their first primary content page.

6.13.1.3 Default signature page template

The default signature page template is a PDF document generated from a Microsoft Word document. Both the PDF template and the source Microsoft Word document are installed when Documentum CM Server is installed. They are installed in `%DM_HOME%\bin` (`$DM_HOME/bin`). The PDF file is named `sigpage.pdf` and the Microsoft Word file is named `sigpage.doc`.

In the repository, the Microsoft Word document that is the source of the PDF template is an object of type `dm_esign_template`. It is named Default Signature Page Template and is stored in `Integration/Esignature/Templates`.

The PDF template document is stored as a rendition of the Microsoft Word document. The page modifier for the PDF rendition is `dm_sig_template`.

The default template allows up to six signatures on each version of a document signed using that template.

6.13.1.4 Default signature creation method

The default signature creation method is a Java method named `esign_pdf`. The method uses the OpenPDF library to generate signature pages. The signature creation method generates a signature page, adds that page as the first page to the original PDF, and the signed PDF is added as a rendition to the primary content.

6.13.1.5 Default content handling

If you are using the default signature creation method, the content to be signed must be in PDF format. The content can be the first primary content page of the document or it can be a rendition of the first content page.

When the method creates the signature page, it appends or prepends the signature page to the PDF content. Whether the signature page is added at the front or back of the content to be signed is configurable. After the method completes successfully, Documentum CM Server adds the content to the document:

- If the signature is the first signature on that document version, the server replaces the original PDF content with the signed content and appends the original PDF content to the document as a rendition with the page modifier `dm_sig_source`.
- If the signature is a subsequent addition, the server simply replaces the previously signed PDF content with the newly signed content.

6.13.1.6 Audit trail entries

Documentum CM Server automatically creates an audit trail entry each time an `addESignature` method is successfully executed. The entry records information about the object being signed, including its name, object ID, version label, and object type. The ID of the session in which it was signed is also recorded. This can be used in connection with the information in the `dm_connect` event for the session to determine what machine was used when the object was signed.

Documentum CM Server uses the generic string properties in the audit trail entry to record information about the signature. The following table lists the use of those properties for a `dm_addesignature` event:

Property	Information stored
`string_1`	Name of the user who signed the object
`string_2`	The justification for the signature
`string_3`	The signature number, the name of the method used to generate the signature, and a hash of the content prior to signing. The hash value is the value provided in the `pre_signatureHash` argument of the `addESignature` method. The information is formatted in the following manner: `sig_number/method_name/pre_signature hash argument`

Property	Information stored
`string_4`	Hash of the primary content page 0. The information also records the hash algorithm and the format of the content. The information is formatted in the following manner: `hash_algorithm/format_name/hash`
`string_5`	Hash of the signed content. The information also records the hash algorithm and the format of the content. The information is formatted in the following manner: `hash_algorithm/format_name/hash` If the signed content was added to the document as primary content, then the value in `string_5` is the same as the `string_4` value.

6.13.1.7 Customizing signatures

If you are using the default electronic signature functionality, signing content in PDF format, you can customize the signature page template. You can add information to the signature page, remove information, or just change its look by changing the arrangement, size, and font of the elements on the page. You can also change whether the signature creation method adds the signature page at the front or back of the content to be signed.

If you want to embed a signature in content that is not in PDF format, you must use a custom signature creation method. You can also create a custom signature page template for use by the custom signature creation method, although using a template is not required.

For more information about customizing electronic signatures and tracing the use of electronic signatures, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

6.13.1.8 Signature verification

Electronic signatures added by `addEsignature` are verified by the `verifyESignature` method. The method finds the audit trail entry that records the latest `dm_addesignature` event for the document and performs the following checks:

- Calls the `IDfAuditTrailManager.verifyAudit` method to verify the Documentum CM Server signature on the audit trail entry.
- Checks that the hash values of the source content and signed content stored in the audit trail entry match those of the source and signed content in the repository.
- Checks that the signatures on the document are consecutively numbered.

Only the most recent signature is verified. If the most recent signature is valid, previous signatures are guaranteed to be valid.

6.13.2 Digital signatures

Digital signatures are electronic signatures, in formats such as PKCS #7, XML Signature, or PDF Signature, that are generated and managed by client applications. The client application is responsible for ensuring that users provide the signature and for storing the signature in the repository. The signature can be stored as primary content or renditions. For example, if the application is implementing digital signatures based on Microsoft Office XP, the signatures are typically embedded in the content files and the files are stored in the repository as primary content files for the documents. If Adobe PDF signatures are used, the signature is also embedded in the content file, but the file is typically stored as a rendition of the document, rather than primary content.

Documentum CM Server supports digital signatures with a property on SysObjects and the `addDigitalSignature` method. The property is a Boolean property called `a_is_signed` to indicate whether the object is signed. The `addDigitalSignature` method generates an audit trail entry recording the signing. The event name for the audit trail entry is `dm_adddigsignature`. The information in the entry records who signed the document, when it was signed, and a reason for signing, if one was provided.

It is possible to require Documentum CM Server to sign the generated audit trail entries. Because the `addDigitalSignature` method is audited by default, there is no explicit registry object for the event. However, if you want Documentum CM Server to sign audit trail entries for `dm_adddigsignature` events, you can issue an explicit method requesting auditing for the event.

For more information about Documentum CM Server signatures on audit trail entries, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*. The associated *Javadocs* provide information about methods to request auditing for the `dm_adddigsignature` event, in the `IDfAuditTrailManager` interface.

6.13.3 Simple sign-offs

Simple sign-offs authenticate the user signing off the object and record information about the sign-off in an audit trial entry. A simple sign-off is useful in situations in which the sign-off requirement is not rigorous. For example, you may want to use a simple sign-off when team members are required to sign a proposal to indicate approval before the proposal is sent to upper management. Simple sign-offs are the least rigorous way to satisfy a signature requirement.

Simple sign-offs are implemented using a `IDfPersistentObject.signoff` method. The method accepts a user authentication name and password as arguments. When the method is executed, Documentum CM Server calls a signature validation program to authenticate the user. If authentication succeeds, Documentum CM Server generates an audit trail entry recording the sign-off. The entry records what was signed, who signed it, and some information about the context of the signing. Using sign-off does not generate an actual electronic signature. The audit trail entry is the only record of the sign-off.

You can use a simple sign-off on any SysObject or SysObject subtype. A user must have at least Read permission on an object to perform a simple sign-off on the object.

You can customize a simple sign-off by creating a custom signature validation program.

For information about creating a custom signature validation program, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*. The associated *Javadocs* provide information on the `IDfPersistentObject.signoff` usage notes.

6.14 Privileged DFC

Privileged DFC is the term used to refer to Foundation Java API instances that are recognized by Documentum CM Servers as privileged to invoke escalated privileges or permissions for a particular operation. In some circumstances, an application may need to perform an operation that requires higher permissions or a privilege than is accorded to the user running the application. In such circumstances, a privileged DFC can request the use of a privileged role to perform the operation. The operation is encapsulated in a privileged module invoked by the Foundation Java API instance.

Supporting privileged DFC is a set of privileged group, privileged roles, and the ability to define type-based objects and simple modules as privileged modules, as follows:

- Privileged groups are groups whose members are granted a particular permission or privileged automatically. You can add or remove users from these groups.
- Privileged roles are groups defined as role groups that can be used by Foundation Java API to give Foundation Java API an escalated permission or privilege required to execute a privileged module. Only Foundation Java API can add or remove members in those groups.
- Privileged modules are modules that use one or more escalated permissions or privileges to execute.

By default, each Foundation Java API is installed with the ability to request escalated privileges enabled. However, to use the feature, the Foundation Java API must have a registration in the global registry. That registration information must be defined in each repository in which the Foundation Java API will exercise those privileges.

Note: In some workstation environments, it may also be necessary to manually modify the Java security policy files to use privileged DFC. For more information about privileged DFC, see *OpenText Documentum Content*

Management - Server Administration and Configuration Guide (EDCCS250400-AGD).

You can disable the use of escalated privileges by a Foundation Java API instance. This is controlled by the `dfc.privilege.enable` key in the `dfc.properties` file.

The `dfc.name` property in the `dfc.properties` file controls the name of the Foundation Java API instance.

For more information about configuring privileged DFC, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD).*

6.14.1 Privileged DFC registrations

Three objects are used to register a Foundation Java API instance for privileged roles:

- Client registration object
- Public key certificate object
- Client rights object

Each installed Foundation Java API has an identity, with a unique identifier extracted from the PKI credentials. The first time an installed Foundation Java API is initialized, it creates its PKI credentials and publishes its identity to the global registry known to the Foundation Java API. In response, a client registration object and a public key certificate object are created in the global registry. The client registration object records the Foundation Java API instance identity. The public key certificate object records the certificate used to verify that identity.

The PKI credentials for a Foundation Java API are stored by default in a file named `dfc.keystore` in the same directory as the `dfc.properties` file. You can change the file location and name if you want, by setting the `dfc.security.keystore.file` key in the `dfc.properties` file.

The first time a Foundation Java API instance is initialized, it creates its own PKI credentials and publishes its identity to the global registry. For subsequent startups, the Foundation Java API instance checks for the presence of its credentials. If they are not found or are not accessible-for instance, when a password has changed Foundation Java API recreates the credentials and republishes its identity to the global registry if privileged DFC is enabled in the `dfc.properties` file. Republishing the credentials causes the creation of another client registration object and public key certificate object for the Foundation Java API instance. Deleting `dfc.keystore` causes the Foundation Java API instance to register again, and the first registration becomes invalid. Recreating the Foundation Java API credentials also invalidates the existing client rights, and client rights objects must be created again for each repository. For more information about creating client rights objects, see *OpenText Documentum Content Management - Administrator User Guide (EDCAC250400-UGD).*

If Foundation Java API finds its credentials, the Foundation Java API may or may not check to determine if its identity is established in the global registry. Whether that check occurs is controlled by the `dfc.verify_registration` key in the `dfc.properties` file. That key is false by default, which means that on subsequent initializations, Foundation Java API does not check its identity in the global registry if Foundation Java API finds its credentials.

A client rights object records the privileged roles that a Foundation Java API instance can invoke. It also records the directory in which a copy of the instance public key certificate is located. Client rights objects are created manually, using Documentum Administrator, after installing the Foundation Java API instance. A client rights object must be created in each repository in which the Foundation Java API instance exercises those roles. Creating the client rights object automatically creates the public key certificate object in the repository.

Client registration objects, client rights objects, and public key certificate objects in the global registry and other repositories are persistent. Stopping the Foundation Java API instance does not remove those objects. The objects must be removed manually if the Foundation Java API instance associated with them is removed or if its identity changes.

If the client registration object for a Foundation Java API instance is removed from the global registry, you can not register that Foundation Java API as a privileged DFC in another repository. Existing registrations in repositories continue to be valid, but you can not register the Foundation Java API in a new repository.

If the client rights objects are deleted from a repository but the Foundation Java API instance is not removed, errors are generated when the Foundation Java API attempts to exercise an escalated privilege or invoke a privileged module.

6.14.2 Recognizing a privileged DFC instance

At runtime, Documentum CM Server must have a way to determine whether a particular Foundation Java API instance is a privileged DFC and, if so, what privileged roles that Foundation Java API can use. To identify itself as a privileged DFC when a Foundation Java API instance wants to use a privileged role, the request is sent with digitally signed information that identifies the instance. Documentum CM Server uses this information to retrieve the client rights object and public key certificate for the instance. Using that information, Documentum CM Server verifies that the Foundation Java API instance has the rights to use that role to perform the requested operation.

6.14.3 Using approved Foundation Java API instances only

It is possible to configure a repository to accept connection requests only from Foundation Java API instances that are successfully authenticated through their client registration objects. If you configure a repository in that manner, its Documentum CM Servers accept connection requests only from Foundation Java API instances that have a valid client rights object in the repository. This behavior is controlled by the `approved_clients_only` property in the docbase config object.

A repository default behavior is to accept connection requests from all Foundation Java API instances, regardless of whether or not they have a client rights object in the repository.

6.15 Encrypted file store storage areas

Encrypted file store storage areas is available with Trusted Content Services.

An encrypted file store storage area is a file store storage area that contains encrypted content files. With Trusted Content Services, you can designate any file store storage area as an encrypted file store. The file store can be a standalone storage area or it can be a component of a distributed store.

Note: If a distributed storage area has multiple file store components, the components can be a mix of encrypted and unencrypted.

A file store storage area is designated as encrypted or unencrypted when you create the storage area. You cannot change the encryption designation after you create the area.

When you store content in an encrypted file store storage area, the encryption occurs automatically. Content is encrypted by Documentum CM Server when the file is saved to the storage area. The encryption is performed using a file store encryption key. Each encrypted storage area has its own file store key. The key is encrypted and stored in the `crypto_key` property of the storage area object (`dm_filestore` object). It is encrypted using the repository encryption key.

Similarly, decryption occurs automatically when the content is fetched from the storage area.

Encrypted content can be full-text indexed. However, the index itself is not encrypted. If you are storing nonindexable content in an encrypted storage area and indexing renditions of the content, the renditions are not encrypted unless you designate their storage area as an encrypted storage area.

You can use dump and load operations on encrypted file stores if you include the content files in the dump file.

Note: The encryption key is 192 bits in length and is used with the Triple DES-EDE-CBC algorithm. For encryption algorithm AES, the supported key lengths are 128-bit, 192-bit or 256-bit.

For more information about the repository encryption key and about dump and load operations, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

6.16 Digital shredding

Digital shredding is available with Trusted Content Services. Using the feature ensures that content in shredding-enabled storage areas is removed from the storage area in a way that makes recovery virtually impossible. When a user removes a document whose content is stored in a shredding-enabled file store storage area, the orphan content object is immediately removed from the repository and the content file is immediately shredded.

Digital shredding uses the capabilities of the underlying operating system to perform the shredding. The shredding algorithm is in compliance with DOD 5220.22-M (NISPOM, National Security Industrial Security Program Operating Manual), option d. This algorithm overwrites all addressable locations with a character, then its complement, and then a random character.

Digital shredding is supported for file store areas if they are standalone storage areas. You can also enable shredding for file store storage areas that are the targets of linked store storage areas. Shredding is not supported for these storage areas if they are components of a distributed storage area.

Digital shredding is not supported for distributed storage areas, nor for the underlying components. It is also not supported for blob, turbo, and external storage areas.

Chapter 7

Content management services

7.1 Document objects

Documents have an important role in most enterprises. They are a repository for knowledge. Almost every operation or procedure uses documents in some way. In the OpenText Documentum CM system, documents are represented by `dm_document` objects, a subtype of `dm_sysobject`.

SysObjects are the supertype, directly or indirectly, of all object types in the hierarchy that can have content. SysObject properties store information about the object version, the content file associated with the object, security permissions on the object, and other important information.

The SysObject subtype most commonly associated with content is `dm_document`.

You can use a document object to represent an entire document or only a portion of a document. For example, a document can contain text, graphics, or tables.

A document object can be either a simple document or a virtual document.

- simple document

  A simple document is a document with one or more primary content files. Each primary content file associated with a document is represented by a content object in the repository. All content objects in a simple document have the same file format.

- virtual document

  A virtual document is a container for other document objects, structured in an ordered hierarchy. The documents contained in a virtual document hierarchy can be simple documents or other virtual documents. A virtual document can have any number of component documents, nested to any level.

  Using virtual documents allows you to combine documents with a variety of formats into one document. You can also use the same document in more than one parent document. For example, you can place a graphic in a simple document and then add that document as a component to multiple virtual documents.

  For more information, see “Virtual documents“ on page 151.

7.2 Document content

Document content is the text, graphics, video clips, and so forth that make up the content of a document. All content in a repository is represented by content objects. All content associated with a document is either primary content or a rendition.

7.2.1 Content objects

A content object is the connection between a document object and the file that actually stores the document content. A content object is an object of type `dmr_content`. Every content file in the repository, whether in a repository storage area or external storage, has an associated content object. The properties of a content object record important information about the file, such as the documents to which the content file belongs, the format of the file, and the storage location of the file.

Documentum CM Server creates and manages content objects. The server automatically creates a content object when you add a file to a document if that file is not already represented by a content object in the repository. If the file already has a content object in the repository, the server updates the `parent_id` property in the content object. The `parent_id` property records the object IDs of all documents to which the content belongs.

Typically, there is only one content object for each content file in the repository. However, if you have Content Storage Services, you can configure the use of content duplication checking and prevention. This feature is used primarily to ensure that numerous copies of duplicate content, such as an email attachment, are not saved into the storage area. Instead, one copy is saved and multiple content objects are created, one for each recipient.

7.2.2 Primary content

Primary content refers to the content that is added to the first content file added to a document. It defines the document primary format. Any other content added in that same format is also called primary content.

Each primary content file in a document has a page number. The page number is recorded in the page attribute of the file's content object. This is a repeating attribute. If the content file is part of multiple documents, the attribute has a value for each document. The file can be a different page in each document.

7.2.3 Renditions

A rendition is a representation of a document that differs from the original document only in its format or some aspect of the format. The first time you add a content file to a document, you specify the content file format. This format represents the primary format of the document. You can create renditions of that content using converters supported by Documentum CM Server or through OpenText™ Documentum™ Content Management Transformation Services - Media, that handles rich media formats such as JPEG, audio, and video formats.

Page numbers are used to identify the primary content that is the source of a rendition.

Converters allows you to:

- Transform one file format to another file format.
- Transform one graphic image format to another graphic image format.

Some of the converters are supplied with Documentum CM Server, while others must be purchased separately. You can use a converter that you have written, or one that is not on the current list of supported converters.

When you ask for a rendition that uses one of the converters, Documentum CM Server saves and manages the rendition automatically.

OpenText Documentum CM provides a suite of additional products that perform specific transformations. For example, OpenText Documentum Content Management (CM) Transformation Services - Media creates two renditions each time a user creates and saves a document with a rich media format:

- A thumbnail rendition
- A default rendition that is specific to the primary content format

Additionally, Transformation Services - Media supports the use of the `TRANSCODE_CONTENT` administration method to request additional renditions.

7.2.3.1 Rendition formats and characteristics

A rendition format indicates what type of application can read or write the rendition. For example, if the specified format is maker, the file can be read or written by Adobe FrameMaker, a desktop publishing application.

A rendition format can be the same format as the primary content page with which the rendition is associated. However, in such cases, you must assign a page modifier to the rendition, to distinguish it from the primary content page file. You can also create multiple renditions in the same format for a particular primary content page. Page modifiers are also used in that situation to distinguish among the renditions. Page modifiers are user-defined strings, assigned when the rendition is added to the primary content.

Documentum CM Server is installed with a wide range of formats. Installing Transformation Services - Media provides an additional set of rich media formats. You can modify or delete the installed formats or add new formats. For information about obtaining a list of formats and how to modify or add a format, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

Each time you add a content file to an object, Documentum CM Server records the content's format in a set of properties in the content object for the file. This internal information includes:

- Resolution characteristics
- Encapsulation characteristics
- Transformation loss characteristics

This information, put together, gives a full format specification for the rendition. It describes the format's screen resolution, any encoding the data has undergone, and the transformation path taken to achieve that format.

- For more information about the supported format conversions on the Windows operating system, see “Supported conversions on Windows” on page 113.
- For more information about the supported format conversions on the Linux operating system, see “Supported conversions on Linux” on page 114.
- For information about `TRANSCODE_CONTENT`, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

7.2.3.2 Generated renditions

- Automatic renditions

  When an application requests a rendition, the application specifies the rendition of the file. If the requested rendition exists in the repository, Documentum CM Server will deliver it to the application. If there is no rendition in that format, but Documentum CM Server can create one, it will do so and deliver the automatically generated rendition to the user.

  For example, suppose you want to view a document whose content is in plain ASCII text. However, you want to see the document with line breaks, for easier viewing. To do so, the application issues a `getFile` and specifies that it wants the content file in `crtext` format. This format uses carriage returns to end lines. Documentum CM Server will automatically generate the `crtext` rendition of the content file and deliver that to the application.

The Documentum CM Server transformation engine always uses the best transformation path available. When you specify a new format for a file, the server reads the descriptions of available conversion programs from the `convert.tbl` file. The information in this table describes each converter, the formats that it accepts, the formats that it can output, the transformation loss expected, and the rendition characteristics that it affects. The server uses these descriptions to decide the best transformation path between the current file format and the requested format.

However, note that the rendition that you create may differ in resolution or quality from the original. For example, suppose you want to display a GIF file with a resolution of 300 pixels per inch and 24-bits of color on a low-resolution (72 pixels per inch) black and white monitor. Transforming the GIF file to display on the monitor results in a loss of resolution.

- User-generated renditions

  At times you may want to use a rendition that cannot be generated by Documentum CM Server. In such cases, you can create the file outside of OpenText Documentum CM and add it to the document using an `addRendition` method in the `IDfSysObject` interface.

  To remove a rendition, use a `removeRendition` method. You must have at least Write permission on the document to remove a rendition of a document.

7.2.3.3 Supported conversions on Windows

Documentum CM Server supports conversions between the three types of ASCII text files. The following table lists the acceptable ASCII text input formats and the obtainable output formats:

Input format	Description of input format	Output formats
`crtext`	ASCII text file with carriage return line feed (for Microsoft Windows clients)	`text` `mactext`
`text`	ASCII text file (for Linux clients)	`crtext` `mactext`
`mactext`	ASCII text file (for Apple Macintosh clients)	`text` `crtext`

7.2.3.4 Supported conversions on Linux

On Linux, Documentum CM Server supports format conversion by using the converters in the `$DM_HOME/convert` directory. This directory contains the following subdirectories:

- `filtrix`
- `pmbplus`
- `pdf2text`
- `psify`
- `scripts`
- `soundkit`
- `troff`

Additionally, Documentum CM Server uses Linux utilities to perform conversions.

You can also purchase and install document converters. OpenText Documentum CM provides demonstration versions of Filtrix converters, which transform structured documents from one word processing format to another. The Filtrix converters are located in the `$DM_HOME/convert/filtrix` directory. To make these converters fully operational, you must contact Blueberry Software, Inc., and purchase a separate license.

You can also purchase and install Frame converters from Adobe Systems Inc. If you install the Frame converters in the Documentum CM Server bin path, the converters are incorporated automatically when you start the OpenText Documentum CM system. The server assumes that the conversion package is found in the Linux bin path of the server account and that this account has the `FMHOME` environment variable set to the FrameMaker home.

7.2.3.4.1 PBM image converters

To transform images, the server uses the PBMPLUS package available in the public domain. PBMPLUS is a toolkit that converts images from one format to another. This package has four parts:

- PBM - For bitmaps (1 bit per pixel)
- PGM - For gray-scale images
- PPM - For full-color images
- PNM - For content-independent manipulations on any of the other three formats and external formats that have multiple types

The parts are upwardly compatible. PGM reads both PBM and PGM and writes PGM. PPM reads PBM, PGM, and PPM, and writes PPM. PNM reads all three and, in most cases, writes the same type as it read. That is, if it reads PPM, it writes PPM.

If PNM does convert a format to a higher format, it issues a message to inform you of the conversion.

The PBMPLUS package is located in the `$DM_HOME/convert/pbmplus` directory. The source code for these converters is found in the `$DM_HOME/unsupported/pbmplus` directory.

The following table lists the acceptable input formats for PBMPLUS:

Input format	Description
`gem`	Digital Research image file
`gif`	General Interchange Format
`macp`	Apple MacPaint file
`pcx`	PCPaint file (Microsoft Windows)
`pict`	Apple Macintosh standard graphics file
`rast`	SUN raster image file
`tiff`	TIFF graphic file
`xbm`	xbitmap file (x.11 Windowing system definition)

The following table lists the acceptable output formats for the PBMPLUS package:

Output format	Description
`gem`	Digital Research image file
`gif`	General Interchange Format
`macp`	Apple MacPaint file
`pcx`	PCPaint file (Microsoft Windows)
`lj`	HP LaserJet
`ps`	PostScript file
`pict`	Apple Macintosh standard graphics file
`rast`	SUN raster image file
tiff	TIFF graphic file
xbm	xbitmap file (X.11 Windowing system definition)

7.2.3.4.2 Miscellaneous converters

Documentum CM Server also uses Linux utilities to provide some miscellaneous conversion capabilities. These utilities include tools for converting to and from Windows DOS format, for converting text into PostScript, and for converting `troff` and `man` pages into `text`. They also include tools for compressing and encoding files.

The following table lists the acceptable input formats for Linux conversion utilities:

Input format	Description
`crtext`	ASCII text file with carriage return line feed (for PCs)
`man`	Online Linux manual
`ps`	PostScript file
`text`	ASCII text file
`troff`	Linux text file

The following table lists the acceptable output formats for Linux conversion utilities:

Output format	Description
`crtext`	ASCII text file with carriage return line feed (for PCs)
`ps`	PostScript file
`text`	text file

7.2.3.5 Connecting source documents and renditions

A rendition can be connected to its source document through a content object or a relation object.

Renditions created by Documentum CM Server or AutoRenderPro™ are always connected through a content object. For these renditions, the rendition property in the content object is set to indicate that the content file represented by the content object is a rendition. The page property in the content object identifies the primary content page with which the rendition is associated.

Renditions created by the media server can be connected to their source either through a content object or using a relation object. The object used depends on how the source content file is transformed. If the rendition is connected using a relation object, the rendition is stored in the repository as a document whose content is the rendition content file. The document is connected to its source through the relation object.

7.2.4 Translations

Documentum CM Server contains support for managing translations of original documents using relationships.

For more information about setting up translation relationships, see “Managing translations” on page 148.

7.2.5 Content checking and duplication

For information about the content checking and duplication feature and about the features supported by Content Storage Services, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

7.3 Versioning

Documentum CM Server provides comprehensive versioning services for all SysObjects except folders and cabinets and their subtypes. Folder and cabinet SysObject subtypes cannot be versioned.

Versioning is an automated process that creates a historical record of a document. Each time you check in or branch a document or other SysObject, Documentum CM Server creates a new version of the object without overwriting the previous version. All the versions of a particular document are stored in a virtual hierarchy called a version tree. Each version on the tree has a numeric version label and, optionally, one or more symbolic version labels.

7.3.1 Version labels

Version labels are recorded in the `r_version_label` property defined for the `dm_sysobject` object type. This is a repeating property. The first index position (`r_version_label[0]`) is reserved for an object numeric version label. The remaining positions are used for storing symbolic labels.

Version labels are used to uniquely identify a version within a version tree. There are several kinds of labels.

- Numeric version labels

  The numeric version label is a number that uniquely identifies the version within the version tree. The numeric version label is generally assigned by the server and is always stored in the first position of the `r_version_label` attribute (`r_version_label[0]`). By default, the first time you save an object, the server sets the numeric version label to 1.0. Each time you check out the object and check it back in, the server creates a new version of the object and increments the numeric version label (1.1, 1.2, 1.3, and so forth). The older versions of the object are not overwritten. If you want to jump the version level up to 2.0 (or 3.0 or 4.0), you must do so explicitly while checking in or saving the document.

> **Note:** If you set the numeric version label manually the first time you check in an object, you can set it to any number you wish, in the format `n.n`, where `n` is zero or any integer value.

- Symbolic version labels

  A symbolic version label is either system- or user-defined. Using symbolic version labels lets you provide labels that are meaningful to applications and the work environment.

  Symbolic labels are stored starting in the second position (`r_version_label[1]`) in the `r_version_label` property. To define a symbolic label, define it in the argument list when you check in or save the document.

  An alternative way to define a symbolic label is to use an `IDfSysObject.mark` method. A mark method assigns one or more symbolic labels to any version of a document. For example, you can use a mark method, in conjunction with an unmark method, to move a symbolic label from one document version to another.

  A document can have any number of symbolic version labels. Symbolic labels are case sensitive and must be unique within a version tree.

- The CURRENT label

  The symbolic label CURRENT is the only symbolic label that the server can assign to a document automatically. When you check in a document, the server assigns CURRENT to the new version, unless you specify a label. If you specify a label (either symbolic or implicit), then you must also explicitly assign the label CURRENT to the document if you want the new version to carry the CURRENT label. For example, the following `checkin` call assigns the labels `inprint` and CURRENT to the new version of the document being checked in:

  ```
  IDfId newSysObjId = sysObj.checkin(false, "CURRENT,inprint");
  ```

  If you remove a version that carries the CURRENT label, the server automatically reassigns the label to the parent of the removed version.

Because both numeric and symbolic version labels are used to access a version of a document, Documentum CM Server ensures that the labels are unique across all versions of the document. The server enforces unique numeric version labels by always generating an incremental and unique sequence number for the labels.

Documentum CM Server also enforces unique symbolic labels. If a symbolic version label specified with a checkin, save, or mark method matches a symbolic label already assigned to another version of the same object, then the existing label is removed and the label is applied to the version indicated by the checkin, save, or mark method.

> **Note:** Symbolic labels are case sensitive. Two symbolic labels are not considered the same if their cases differ, even if the word is the same. For example, the labels `working` and `Working` are not the same.

7.3.2 Version trees

A version tree refers to an original document and all of its versions. The tree begins with the original object and contains all versions of the object derived from the original.

To identify which version tree a document belongs to, the server uses the document `i_chronicle_id` property value. This property contains the object ID of the original version of the document root of the version tree. Each time you create a new version, the server copies the `i_chronicle_id` value to the new document object. If a document is the original object, the values of `r_object_id` and `i_chronicle_id` are the same.

To identify the place of a document on a version tree, the server uses the document numeric version label.

7.3.3 Branching

A version tree is often a linear sequence of versions arising from one document. However, you can also create branches. Figure 7-1 illustrates a version tree that contains branches.

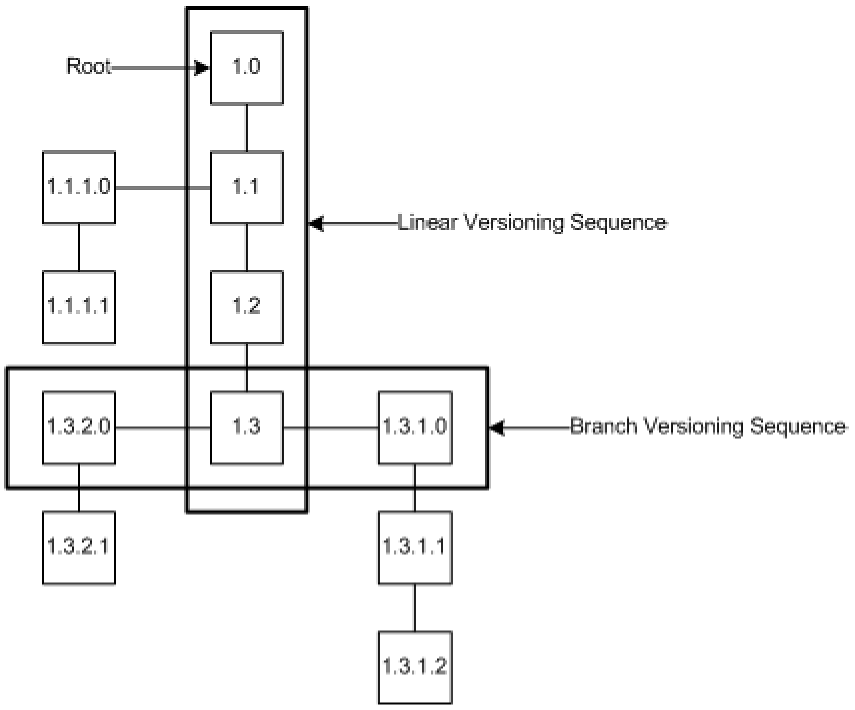


Figure 7-1: Branching

The numeric version labels on versions in branches always have two more digits than the version at the origin of the branch. For example, looking at the preceding figure, version 1.3 is the origin of two branches. These branches begin with the numeric version labels 1.3.1.0 and 1.3.2.0. If a branch off version 1.3.1.2 were created, the number of its first version would be 1.3.1.2.1.0.

Branching takes place automatically when you check out and then check back in an older version of a document because the subsequent linear versions of the document already exist and the server cannot overwrite a previously existing version. You can also create a branch by using the `IDfSysObject.branch` method instead of the `checkout` method when you get the document from the repository.

When you use a `branch` method, the server copies the specified document and gives the copy a branched version number. The method returns the `IDfID` object representing the new version. The parent of the new branch is marked immutable (unchangeable).

After you branch a document version, you can make changes to it and then check it in or save it. If you use a `checkin` method, you create a subsequent version of your branched document. If you use a `save` method, you overwrite the version created by the `branch` method.

A `branch` method is particularly helpful if you want to check out a locked document.

7.3.4 Removing versions

Documentum CM Server provides two ways to remove a version of a document. If you want to remove only one version, use a `IDfPersistentObject.destroy` method. If you want to remove more than one version, use a `IDfSysObject.prune` method.

With a prune method, you can prune an entire version tree or only a portion of the tree. By default, prune removes any version that does not belong to a virtual document and does not have a symbolic label.

To prune an entire version tree, identify the first version of the object in the method arguments. The object ID of the first version of an object is found in the `i_chronicle_id` property of each subsequent version. Query this property if you need to obtain the object ID of the first version of an object.

To prune only part of the version tree, specify the object ID of the version at the beginning of the portion you want to prune. For example, to prune the entire tree, specify the object ID for version 1.0. To prune only version 1.3 and its branches, specify the object ID for version 1.3.

You can also use an optional argument to direct the method to remove versions that have symbolic labels. If the operation removes the version that carries the symbolic label CURRENT, the label is automatically reassigned to the parent of the removed version.

When you prune, the system does not renumber the versions that remain on the tree. The system simply sets the `i_antecedent_id` property of any remaining version to the appropriate parent.

For example, look at the following figure. Suppose the version tree shown in the **Before pruning** area is pruned, beginning the pruning with version 1.2 and that versions with symbolic labels are not removed. The result of this operation is shown in the **After pruning** area. Notice that the remaining versions have not been renumbered.

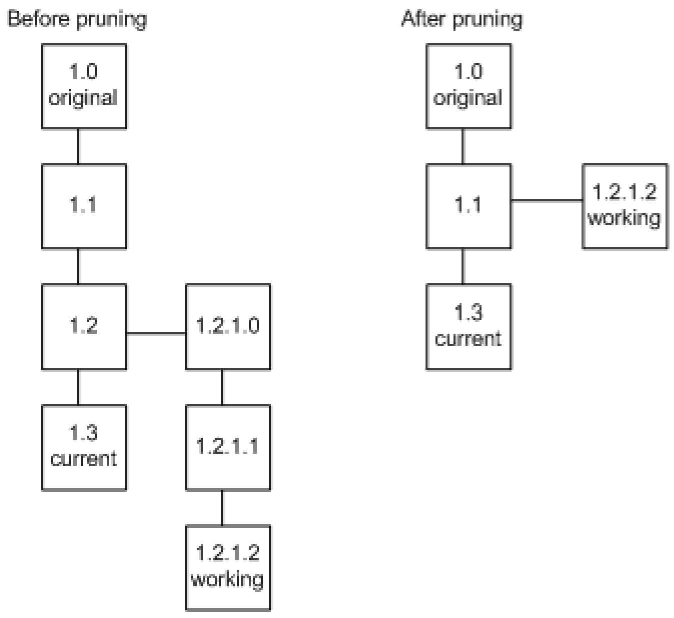


Figure 7-2: Before and after pruning

7.3.5 Changeable versions

You can modify the most recent version on any branch of a version tree. For instance, in the Figure 7-1, you can modify the following versions:

- 1.3
- 1.3.1.2
- 1.3.2.1
- 1.1.1.1

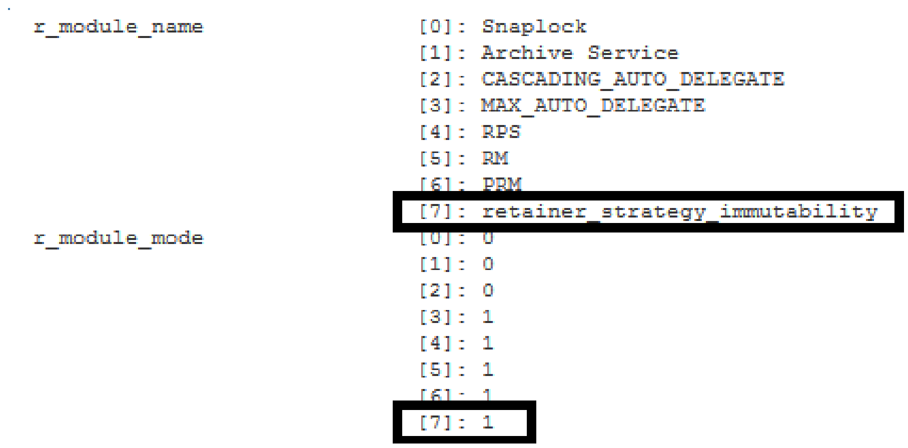
The other versions are immutable. However, you can create new, branched versions of immutable versions.

7.4 Immutability

Immutability is a characteristic that defines an object as unchangeable. An object is marked immutable if one of the following occurs:

- The object is versioned or branched.
- An `IDfSysObject.freeze` method is executed against the object.
- The object is associated with a retention policy that designates controlled documents as immutable.

In releases prior to 7.0, you can only mark documents immutable. Starting with OpenText Documentum CM 7.0, you can apply immutability rules to folders. To enable this feature in a repository, you must append a key-value pair (`retainer_strategy_immutability, 1`) to the `r_module_name` and `r_module_mode` properties of the `dm_docbase_config` object respectively. In these two properties, the indexes that the key (`retainer_strategy_immutability`) and value (`1`) use must match. In the following example, both properties use the index `[7]` to hold the key-value pair:



```
r_module_name                [0]: Snaplock
                             [1]: Archive Service
                             [2]: CASCADING_AUTO_DELEGATE
                             [3]: MAX_AUTO_DELEGATE
                             [4]: RPS
                             [5]: RM
                             [6]: PRM
                             [7]: retainer_strategy_immutability
r_module_mode                [0]: 0
                             [1]: 0
                             [2]: 0
                             [3]: 1
                             [4]: 1
                             [5]: 1
                             [6]: 1
                             [7]: 1
```

7.4.1 Effects of a checkin or branch method

When a user creates a new version of a document (or any SysObject or SysObject subtype), Documentum CM Server sets the `r_immutable_flag` property to `TRUE` in the old version. Users can no longer change the old version content or most of its property values.

7.4.2 Effects of a freeze method

Use a freeze method when you want to mark an object as immutable without creating a version of the object. When you freeze an object, users can no longer change its content, its primary storage location, or many of its properties. The content, primary storage location, and the frozen properties remain unchangeable until you explicitly unfreeze the object.

Note: A freeze method cannot be used to stop workflows. If you want to suspend a workflow, use an `IDfWorkflow.haltAll` method.

When you freeze an object, the server sets the following properties of the object to `TRUE`:

- `r_immutable_flag`

  This property indicates whether the object is changeable. If set to `TRUE`, you cannot change the object content, primary storage location, or most of its properties.
- `r_frozen_flag`

  This property indicates whether the `r_immutable_flag` property was set to `TRUE` by an explicit freeze method call.

If the object is a virtual document, the method sets additional properties and offers the option of freezing the components of any snapshot associated with the object.

To unfreeze an object, use an `IDfSysObject.unfreeze` method. Unfreezing an object resets the `r_frozen_flag` attribute to `FALSE`. If the object has not been previously versioned, then unfreezing it also resets the `r_immutable_flag` to `FALSE`. The method has an argument that, if set to `TRUE`, unfreezes the components of a snapshot associated with the object.

For more information about the additional attributes that are set when a virtual document is frozen, see "Freezing a document" on page 163.

For more information about how unfreezing affects a virtual document, see "Unfreezing a document" on page 164.

7.4.3 Effects of a retention policy

When a document is associated with a retention policy that is defined to make all documents it controls immutable, the document `r_immutable_flag` property is set to `TRUE`.

7.4.4 Attributes that remain changeable

Some properties are changeable even when an object `r_immutable_flag` property is set to `TRUE`. Users or applications can change the following properties:

- `r_version_label` (only symbolic labels, not the numeric label)
- `i_folder_id` (the object can be linked or unlinked to folders and cabinets)
- `acl_domain`, `acl_name`, `owner_name`, `group_name`, `owner_permit`, `group_permit`, `world_permit` (the security attributes)
- `a_special_app`
- `a_compound_architecture`
- `a_full_text` (requires Sysadmin or Superuser privileges)
- `a_storage_type`

The server can change the following attributes:

- `a_archive`
- `i_isdeleted`
- `i_vstamp`
- `r_access_date`
- `r_alias_set_id`
- `r_aspect_name`
- `r_current_state`
- `r_frozen_flag`
- `r_frzn_assembly_cnt`
- `r_policy_id`
- `r_immutable_flag`
- `i_reference_cnt`
- `r_policy_id`
- `r_resume_state`

A data dictionary attribute defined for the `dm_dd_info` type provides additional control over immutability for objects of type `dm_sysobject` or any subtypes of SysObject. The attribute is called `ignore_immutable`. When set to `TRUE` for a SysObject-type attribute, the attribute is changeable even if the `r_immutable_flag` for the containing object instance is set to `TRUE`.

For information about using the `ALTER TYPE` statement to set or change data dictionary attributes, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

7.5 Concurrent access control

In a multiuser environment, a document management system must provide some means to ensure the integrity of documents by controlling concurrent access to documents. Documentum CM Server provides three locking strategies for SysObjects:

- Database-level locking
- Repository-level locking
- Optimistic locking

7.5.1 Database-level locking

Database-level locking places a physical lock on an object in the RDBMS tables. Access to the object is denied to all other users or database connections.

Database locking is only available in an explicit transaction—a transaction opened with an explicit method or statement issued by a user or application. For example, the DQL `BEGIN TRAN` statement starts an explicit transaction. The database lock is released when the explicit transaction is committed or aborted.

A system administrator or superuser can lock any object with a database-level lock. Other users must have at least Write permission on an object to place a database lock on the object. Database locks are set using the `IDfPersistentObject.lock` method.

Database locks provide a way to ensure that deadlock does not occur in explicit transactions and that save operations do not fail due to version mismatch errors.

If you use database locks, using repository locks is not required unless you want to version an object. If you do want to version a modified object, you must place a repository-level lock on the object also.

7.5.2 Repository-level locking

Repository-level locking occurs when a user or application checks out a document or object. When a checkout occurs, Documentum CM Server sets the object `r_lock_owner`, `r_lock_date`, and `r_lock_machine` properties. Until the lock owner releases the object, the server denies access to any user other than the owner.

Use repository-level locking in conjunction with database-level locking in explicit transactions if you want to version an object. If you are not using an explicit transaction, use repository-level locking whenever you want to ensure that your changes can be saved.

To use a checkout method, you must have at least Version permission for the object or have Superuser privileges.

Repository locks are released by check-in methods (`IDfSysObject.checkin` or `IDfSysObject.checkinEx`). A check-in method creates a new version of the object, removes the lock on the old version, and gives you the option to place a lock on the new version.

If you use a save method to save your changes, you can choose to keep or relinquish the repository lock on the object. Save methods, which overwrite the current version of an object with the changes you made, have an argument that allows you to direct the server to hold the repository lock.

A `cancelCheckOut` method also removes repository locks. This method cancels a checkout. Any changes you made to the document are not saved to the repository.

7.5.3 Optimistic locking

Optimistic locking occurs when you use a fetch method to access a document or object. It is called optimistic because it does not actually place a lock on the object. Instead, it relies on version stamp checking when you issue the save to ensure that data integrity is not lost. If you fetch an object and change it, there is no guarantee your changes will be saved.

When you fetch an object, the server notes the value in the object `i_vstamp` attribute. This value indicates the number of committed transactions that have modified the object. When you are finished working and save the object, the server checks the current value of the object `i_vstamp` property against the value that it noted when you fetched the object. If someone else fetched (or checked out) and saved the object while you were working, the two values will not match and the server does not allow you to save the object.

Additionally, you cannot save a fetched object if someone else checks out the object while you are working on it. The checkout places a repository lock on the object.

For these reasons, optimistic locking is best used when:

- There are a small number of users on the system, creating little or no contention for desired objects.
- There are only a small number of non-content related changes to be made to the object.

For more information, see “Object-level permissions” on page 88.

For more information, see “User privileges” on page 86.

7.6 Document retention and deletion

A document remains in the repository until an authorized user (the owner or another privileged user) deletes the document. However, if business or compliance rules require the document to be retained for a specific length of time, you can ensure that it is not deleted within that period by applying retention to the document. If a document (or any other content-containing SysObject) is under retention control, it may only be deleted under special conditions.

Documentum CM Server supports two ways to apply retention:

- Retention policies

  Retention policies are part of the larger retention services provided by Retention Policy Services. These services allow you to manage the entire life of a document, including its disposition after the retention period expires. Consequently, documents associated with an active retention policy are not automatically deleted when the retention period expires. Instead, they are held in the repository until you impose a formal disposition or use a privileged delete to remove them.

  Using retention policies requires Retention Policy Services. With Retention Policy Services, you can define and apply retention policies through Retention Policy Services Administrator (an administration tool that is similar to, but separate from, Documentum Administrator). Retention policies can be applied to documents in any storage area type.

  Using retention policies is the recommended way to manage document retention.

- Content-addressed storage area retention periods

  If you are using content-addressed storage areas, you can configure the storage area to enforce a retention period on all content files stored in that storage area. The period is either explicitly specified by the user when saving the associated document or applied as a default by the Centera host system.

7.6.1 Retention policies

A retention policy defines how long an object must be kept in the repository. The retention period can be defined as an interval or a date. For example, a policy might specify a retention interval of five years. If so, then any object to which the policy is applied is held for five years from the date on which the policy is applied. If a date is set as the retention period, then any object to which the policy is applied is held until the specified date.

A retention policy is defined as either a fixed or conditional policy. If the retention policy is a fixed policy, the defined retention period is applied to the object when the policy is attached to the object. For example, suppose a fixed retention policy defines a retention period of five years. If you attach that policy to an object, the object is held in the repository for five years from the date on which the policy was applied.

If the retention policy is a conditional policy, the retention period is not applied to the object until the event occurs. Until that time, the object is held under an infinite retention (that is, the object is retained indefinitely). After the event occurs, the retention period defined in the policy is applied to the object. For example, suppose a conditional retention policy requires employment records to be held for 10 years after an employee leaves a company. This conditional policy is attached to all employment records. The records of any employee are retained indefinitely until the employee leaves the company. At that time, the conditional policy takes effect and the employee records are marked for retention for 10 years from the date of termination.

You can apply multiple retention policies to an object. In general, the policies can be applied at any time to the object.

The date an object retention expires is recorded in an object `i_retain_until` property. However, if there are conditional retention policies attached to the object, the value in that property is null until the condition is triggered. If there are multiple conditional retention policies attached to the object, the property is updated as each condition is triggered if the triggered policy retention period is further in the future than the current value of `i_retain_until`. However, Documentum CM Server ignores the value, considering the object under infinite retention, until all conditions are triggered.

A policy can be created for a single object, a virtual document, or a container such as a folder. If the policy is created for a container, all the objects in the container are under the control of the policy.

An object can be assigned to a retention policy by any user with Read permission on the object or any user who is a member of either the `dm_retention_managers` group or the `dm_retention_users` group. These groups are created when Documentum CM Server is installed. They have no default members.

Policies apply only to the specific version of the document or object to which they are applied. If the document is versioned or copied, the new versions or copies are not controlled by the policy unless the policy is explicitly applied to them. Similarly, if a document under the control of a retention policy is replicated, the replica is not controlled by the policy. Replicas may not be associated with a retention policy.

7.6.2 Storage-based retention periods

Storage-based retention periods are applied to content files stored in a content-addressed storage area. The period may be specified by the user or application that saves the document containing the file or it may be assigned based on a default period defined in the storage area.

7.6.3 Behavior if both a retention policy and storage-based retention apply

If a retention policy is assigned to a document and the document content is stored in a content-addressed storage area, the retention period furthest in the future is applied to the document. The retention value associated with the file content address is set to the date furthest in the future.

Similarly, the property `i_retain_until` is set to the date furthest in the future. For example, suppose a document created on April 1, 2005 is stored in a content-addressed storage area and assigned to a retention policy. The retention policy specifies that it must be held for five years. The expiration date for the policy is May 31, 2010. The content-addressed storage area has a default retention period of eight years. The expiration date for the storage-based retention period is May 31, 2013. Documentum CM Server will not allow the document to be deleted (without using a forced deletion) until May 31, 2013. The `i_retain_until` property is set to May 31, 2013.

If the retention policy is a conditional retention policy, the property value is ignored until the event occurs and the condition is triggered. At that time, the property is set to the retention value defined by the conditional policy. If multiple conditional retention policies apply, the property is updated as each is triggered if the triggered policy retention period is further in the future than the value already recorded in `i_retain_until`. However, Documentum CM Server ignores the value in `i_retain_until` all the policies are triggered. Until all conditional policies are triggered, the object is held in infinite retention.

7.6.4 Deleting documents under retention

Deleting documents associated with an active retention policy or with unexpired retention periods in a content-addressed storage area requires special operations:

- To delete a document associated with an active retention policy, you must perform a privileged deletion.
- To delete a document with an unexpired retention period stored in a content-addressed storage area, you must perform a forced deletion.

If a document is controlled by a retention policy and its content is stored in retention-enabled content-addressed storage area, you may be required to use both a privileged deletion and a forced deletion to remove the document.

- Privileged deletions

  Use a privileged deletion to remove documents associated with an active retention policy. Privileged deletions succeed if the document is not subject to any holds imposed through the Retention Policy Manager. You must be a member of the `dm_retention_managers` group and have Superuser privileges to perform a privileged deletion.
- Forced deletions

Forced deletions remove content with unexpired retention periods from retention-enabled content-addressed storage areas. You must be a superuser or a member of the `dm_retention_managers` group to perform a forced deletion.

The force delete request must be accompanied by a Centera profile that gives the requesting user the Centera privileges needed to perform a privileged deletion on the Centera host system. The Centera profile must be defined prior to the request. For information about defining a profile, contact the Centera system administrator at your site.

A forced deletion removes the document from the repository. If the content is not associated with any other documents, a forced deletion also removes the content object and associated content file immediately. If the content file is associated with other SysObjects, the content object is simply updated to remove the reference to the deleted document. The content file is not removed from the storage area.

Similarly, if the content file is referenced by more than one content object, the file is not removed from the storage area. Only the document and the content object that connects that document to the content file are removed.

7.6.5 Deleting versions and renditions

As documents are checked out and in to the repository, the version tree for the document grows. If you want to remove older versions of a document, and those versions do not have an unexpired retention period, you can use a destroy or prune method, or the Version Management administration tool.

You can remove unneeded renditions using the Rendition Management administration tool.

7.6.6 Retention in distributed environments

In a distributed environment, document content can be saved to a repository from a remote location, through OpenText™ Documentum™ Content Management Branch Office Caching Services, using either a synchronous or an asynchronous write operation. In a synchronous write operation, the content is saved to the appropriate storage area immediately. In an asynchronous write operation, the content is parked on OpenText Documentum Content Management (CM) Branch Office Caching Services and written to the storage area at a later time. In both options, the document metadata is saved to the repository immediately.

Regardless of whether the content is written synchronously or asychronously, if the document is under retention, retention is enforced as soon as the document metadata is saved to the repository.

- For information, see “Retention policies” on page 127. For more information about retention policies and their use, see *OpenText Documentum Content Management - Administrator User Guide (EDCAC250400-UGD)*.
- *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)* contains information about:

  - Content-addressed storage areas
  - How retention periods in a content-addressed storage area are defined and how they are implemented internally
  - Enabling the use of a Centera profile for a forced deletion
  - The Version Management tool
  - The Rendition Management tool
- For more information about using destroy or prune, see "Removing versions" on page 120.

7.7 Documents and lifecycles

Lifecycles represent the stages in the life of a document. A lifecycle consists of a linear sequence of states. Each state has associated entry criteria and actions that must be performed before an object can enter the state.

After you create a document, you can attach it to any lifecycle that is valid for the document object type. Only a user with the Change State extended permission can move the document from one state to another.

7.8 Documents and full-text indexing

All objects of type SysObject or SysObject subtypes are full-text indexed. The values of the object properties and content, if it has content, are indexed. Properties defined for any aspects associated with an object are not indexed unless those properties are defined for indexing in the aspect definition.

You can turn off indexing of object content or properties in several ways:

- Set the property `a_full_text` of an object type to false. The properties are indexed but not the content. You must have Sysadmin or Superuser privileges to change the value to `F`.
- Set enable indexing to `false` in Documentum Administrator to turn off indexing events for specific object types. Properties are indexed.
- Turn off indexing for specific formats by setting the `can_index` property to `false`. Properties are indexed.
- Use xPlore index agent filters to filter out content and metadata for specific types or repository paths.

7.9 Creating document objects

The most commonly created SysObject is a document or a document subtype. End users typically use a OpenText Documentum CM client application, such as Documentum Webtop, to create documents or to import documents created in an external editor. For information about using a OpenText Documentum CM client product to create documents, or other SysObjects, refer to the documentation for that component. For information about creating documents or other SysObjects programmatically, see *OpenText Documentum Content Management - Foundation Java API Development Guide (EDCPKCL250400-DGD)*. This section provides some important conceptual information about certain properties and the management of new SysObjects.

The `owner_name` property identifies the user or group who owns an object.

By default, an object is owned by the user who creates the object. However, you can assign ownership to another user or a group by setting the `owner_name` property. To change the object owner, you must be a superuser, the current owner of the object, or a user with Change Owner permission.

The `default_folder` property records the name of the primary location for an object. The primary location is the repository cabinet or folder in which the server stores a new object the first time the object is saved into the repository. Although this location is sometimes referred to as the primary cabinet for the object, it can be either a cabinet or a folder.

The home cabinet of a user is the default primary location for a new document (or any other SysObject) a user creates. It is possible to specify a different location programmatically by setting the `default_folder` property or linking the object to a different location.

After you define a primary location for a object, it is not necessary to define the location again each time you save the object.

7.9.1 Adding content

When users create a SysObject using a OpenText Documentum CM client, adding content is a seamless operation. Creating a new document using Documentum Webtop typically invokes an editor that allows users to create the content for the document as part of creating the object. If you are creating the object using an application or DQL, you must create the content before creating the object and then add the content to the object. You can add the content before or after you save the object.

Content can be a file or a block of data in memory. The method used to add the content to the object depends on whether the content is a file or data block.

The first content file added to an object determines the primary format for the object. The format is set and recorded in the `a_content_type` property of the object.

Thereafter, all content added to the object as primary content must have the same format as that first primary content file.

> **Note:** If you discover that the `a_content_type` property is set incorrectly for an object, it is not necessary to re-add the content. You can check out the object, reset the property, and save (or check in) the object.
>
> After you create content, you can add more content by appending a new file to the end of the object, or you can insert the file into the list.
>
> The content can be a file or a block of data, but it must reside on the same machine as the client application.

Renditions are typically copies of the primary content in a different format. You can add as many renditions of primary content as needed.

You cannot use DQL to add a file created on a Macintosh machine to an object. You must use a Foundation Java API method. Older Macintosh-created files have two parts: a data fork (the actual text of the file) and a resource fork. The Foundation Java API, in the `IDfSysObject` interface, includes methods that allow you to specify both the content file and its resource fork when adding content to a document.

7.9.2 Storing content

OpenText Documentum CM supports a variety of storage area options for storing content files. The files can be stored in a file system, in content-addressable storage, on external storage devices, or even within the RDBMS, as metadata. For the majority of documents, the storage location of their content files is typically determined by site administration policies and rules. These rules are enforced by using content assignment policies or by the default storage algorithm. End users and applications create documents and save or import them into the repository without concern for where they are stored. Exceptions to business rules can be assigned to a specific storage area on a one-by-one basis as they are saved or imported into the repository.

This section provides an overview of the ways in which the storage location for a content file is determined.

7.9.2.1 Content assignment policies

> **Note:** Content assignment policies is a feature of Content Storage Services.

Content assignment policies let you fully automate assigning content to file stores and content-addressed storage areas.

A content assignment policy contains one or more rules, expressed as conditions such as `content_size>10,000 (bytes)` or `format='gif'`. Each rule is associated with a file store or content addressed storage area. When a policy is applied to a document, the document is tested against each rule. When the document satisfies a rule, its content is stored in the storage area associated with the rule and the remaining rules are ignored.

Content assignment polices can only assign content to file store storage areas or content-addressed storage areas. Policies are enforced by Foundation Java API-based client applications (5.2.5 SP2 and later), and are applied to all new content files, whether created by a save or import into the repository or a checkin operation.

7.9.2.2 Default storage allocation

The default storage algorithm uses values in a document associated object, format object, or type definition to determine where to assign the content for storage.

The default storage algorithm is used when:

- Storage policies are not enabled
- Storage policies are enabled but a policy does not exist for an object type or for any of the type supertypes
- A content file does not satisfy any of the conditions in the applicable policy
- Content is saved with a retention date

7.9.2.3 Explicitly assigning a storage area

You can override a storage policy or the default storage algorithm by explicitly setting the `a_storage_type` attribute for an object before you save the object to the repository.

7.9.3 Setting content properties and metadata for content-addressed storage

Content-addressed storage areas allow you to store metadata, including a value for a retention period, with each piece of content in the system. Each of the storage system metadata fields that you want to set when content is stored is identified in the ca store object and in the content object representing the content file.

When a content file is saved to content-addressed storage, the metadata values are stored first in the content object and then copied into the storage area. Only those metadata fields that are defined in both the content object and the ca store object are copied to the storage area.

In the content object, the properties that record the metadata are:

- `content_attr_name`
- `content_attr_value`
- `content_attr_data_type`
- `content_attr_num_value`

- `content_attr_date_value`

These are repeating properties. When a `setContentAttrs` method or a `SET_CONTENT_ATTRS` administration method is issued, the name and value pairs identified in the parameter argument are stored in these content properties. The name is placed in the `content_attr_name` property and the value is stored in the property corresponding to the field datatype. For example, suppose a `setContentAttrs` method `identified` `title=DailyEmail` as a name and value pair. The method would append `title` to the list of field names in `content_attr_name` and store `DailyEmail` in `content_attr_value` in the corresponding index position. If title is already listed in `content_attr_name`, the value currently stored in `content_attr_value` is overwritten.

In a ca store object, the properties that identify the metadata are:

- `a_content_attr_name`

  This is a list of the metadata fields in the storage area to be set.

- `a_retention_attr_name`

  This identifies the metadata field that contains the retention period value.

When `setContentAttrs` runs, the metadata name and value pairs are stored first in the content object properties. Then, the plug-in library is called to copy them from the content object to the storage system metadata fields. Only those fields that are identified in both `content_attr_name` in the content object and in either `a_content_attr_name` or `a_retention_attr_name` in the storage object are copied to the storage area.

If `a_retention_attr_required` is set to `T (TRUE)` in the ca store object, the user or application must specify a retention period for the content when saving the content. That is accomplished by including the metadata field identified in the `a_retention_attr_name` property of the storage object in the list of name and value pairs when setting the content properties.

If `a_retention_attr_required is` set to `F (FALSE)`, then the content is saved using the default retention period, if one is defined for the storage area. However, the user or application can overwrite the default by including the metadata field identified in the `a_retention_attr_name` property of the storage object when setting the content properties.

The value for the metadata field identified in `a_retention_attr_name` can be a date, a number, or a string. For example, suppose the field name is `retain_date` and content must be retained in storage until January 1, 2016. The `setContentAttrs` parameter argument would include the following name and value pair:

```
'retain_date=DATE(01/01/2016)'
```

You can specify the date value using any valid input format that does not require a pattern specification. Do not enclose the date value in single quotes.

To specify a number as the value, use the following format:

```
'retain_date=FLOAT(number_of_seconds)'
```

For example, the following sets the retention period to 1 day (24 hours):

```
'retain_date=FLOAT(86400)'
```

To specify a string value, use the following format:

```
'retain_date="number_of_seconds"'
```

The string value must be numeric characters that Documentum CM Server can interpret as a number of seconds. If you include characters that cannot be translated to a number of seconds, Documentum CM Server sets the retention period to 0 by default, but does not report an error.

When using administration methods to set the metadata, use a `SET_CONTENT_ATTRS` method to set the content object attributes and a `PUSH_CONTENT_ATTRS` method to copy the metadata to the storage system.

The `Setcontentattrs` method must be run after the content is added to the SysObject and before the object is saved to the repository. The `SET_CONTENT_ATTRS` and `PUSH_CONTENT_ATTRS` methods must be executed after the object is saved to the repository.

7.9.4 Setting content properties and metadata for S3-compatible storage

S3-compatible storage areas allow you to store metadata with each piece of content in the system. Each of the storage system metadata fields that you want to set when content is stored is identified in the content object representing the content file.

When a content file is saved to S3-compatible storage, the metadata values are stored first in the content object and then copied into the storage area.

In the content object, the properties that record the metadata are:

- `content_attr_name`
- `content_attr_value`
- `content_attr_data_type`
- `content_attr_num_value`
- `content_attr_date_value`

These are repeating properties. When a `setContentAttrs` method or a `SET_CONTENT_ATTRS` administration method is issued, the name and value pairs identified in the parameter argument are stored in these content properties. The name is placed in the `content_attr_name` property and the value is stored in the property corresponding to the field datatype. For example, suppose a `setContentAttrs` method identified `title=DailyEmail` as a name and value pair. The method would append `title` to the list of field names in `content_attr_name` and store `DailyEmail` in `content_attr_value` in the corresponding index position. If title is already listed in `content_attr_name`, the value currently stored in `content_attr_value` is overwritten.

When the `setContentAttrs` method is executed, the metadata name and value pairs are stored first in the content object properties. After the `PUSH_CONTENT_ATTRS` method reads the content object attributes, it collects all user metadata available in the corresponding content object and prepends `X-AMZ-metadata-` with the collected metadata. Documentum CM Server then generates the PUT request containing headers with `X-AMZ-metadata-*` and sends the request to S3-compatible store.

When using administration methods to set the metadata, use a `SET_CONTENT_ATTRS` method to set the content object attributes and a `PUSH_CONTENT_ATTRS` method to copy the metadata to the storage system.

The `Setcontentattrs` method must be executed after the content is added to the SysObject and before the object is saved to the repository. The `SET_CONTENT_ATTRS` and `PUSH_CONTENT_ATTRS` methods must be executed after the object is saved to the repository.

7.9.5 Document objects and Access Control Lists

An Access Control List (ACL), specifies access permissions for an object. The standard entries can give a user or group any of the basic access permissions, extended permissions or both. You can also add entries that restrict access for specific users or groups and entries that specify permissions recognized only by specific applications.

Each object of type SysObject or SysObject subtype has one ACL that controls access to that object. The server automatically assigns a default ACL to a new SysObject if you do not explicitly assign an ACL to the object when you create it. If a new object is stored in a room (a secure area in a repository) and is governed by that room, the ACL assigned to the object is the default ACL for that room.

The ACL associated with an object is identified by two properties of the SysObject: `acl_name` and `acl_domain`. The `acl_name` is the name of the ACL and `acl_domain` records the owner of the ACL.

- For more information about creating and adding renditions, see “Renditions” on page 111.
- For more information about setting permissions for a SysObject, see “Assigning ACLs” on page 143.
- *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)* contains information about:
  - The implementation and use of the options for determining where content is stored
  - The behavior and implementation of content assignment policies and creating them

- How the default storage algorithm behaves
- Configuring a storage area to require a retention period for content stored in that area

7.10 Modifying document objects

This section contains information about modifying existing documents and other SysObjects.

The ability to modify a SysObject is controlled by object-level permissions and whether the object is under the control of a retention policy. Additionally, the ability to modify an object can be affected by application-defined roles.

Object-level permissions are defined in ACLs. Each SysObject has an associated ACL object that defines the access permissions for that object. The entries in the ACL define who can access the object and the operations allowed for those having access. Users with Superuser privileges can always access a SysObject because a superuser always has at least Read permission on SysObjects and has the ability to modify ACLs.

If the object is under the control of a retention policy, users cannot overwrite the content regardless of their permissions. Documents controlled by a retention policy may only be versioned or copied. Additionally, some retention policies set documents under their control as immutable. In that case, users can change only some of the document attributes.

Application-level control of access to a particular object is accomplished using role groups. These groups control which applications users can use to modify a document.

You cannot modify the content of objects that are included in a frozen (unchangeable) snapshot or that have the `r_immutable_flag` attribute set to `TRUE`. Similarly, most attributes of such objects are also unchangeable.

7.10.1 Accessing a document in the repository

Before a user or application can modify a SysObject, the object must be obtained from the repository. There are three options for obtaining an object from the repository:

- A `lock` method
- A `checkOut` method
- A `fetch` method

These methods retrieve the object metadata from the repository. Retrieving the object content requires a separate method. However, you must execute a lock, checkOut, or fetch before retrieving the content files.

A `lockEx(true)` method provides database-level locking. A physical lock is placed on the object at the RDBMS level. You can use database-level locking only if the user or application is in an explicit transaction. If you want to version the object, you must also issue a `checkOut` method after the object is locked.

Checking out a document places a repository lock on the object. A repository lock ensures that while you are working on a document, no other user can make changes to that document. Checking out a document also offers you two alternatives for saving the document when you are done. You need `Version` or `Write` permission to check out a document.

Use a `fetch` method when you want to read but not change an object. The method does not place either a repository or database lock on the object. Instead, the method uses optimistic locking. Optimistic locking does not restrict access to the object, and only guarantees that one user cannot overwrite the changes made by another. Consequently, it is possible to fetch a document, make changes, and not be able to save those changes. In a multiuser environment, it is generally best to use the fetch method only to read documents or if the changes you want to make will take a very short time.

To use fetch, you need at least `Read` permission to the document. With `Write` permission, you can use a `fetch` method in combination with a save method to change and save a document version.

After you have checked out or fetched the document, you can change the attributes of the document object or add, replace, or remove primary content. To change the object current primary content, retrieve the content file first.

7.10.2 Modifying single-valued attributes

If you modify the value of a single-valued attribute, the new value overwrites the old value.

In Foundation Java API, most attributes have a specific set method that sets the attribute. For example, if you wanted to set the subject attribute of a document, you call the `setSubject` method. There is also a generic set method that you can use to set any attribute.

7.10.3 Modifying repeating attributes

You can modify a repeating attribute by adding additional values, replacing current values, or removing values.

When you add a value, you can append it to the end of the values in the repeating property or you can replace an existing value. If you remove a value, all the values at higher index positions within the property are adjusted to eliminate the space left by the deleted value. For example, suppose a keywords property has four values:

```
keywords[0]=engineering
keywords[1]=productX
keywords[2]=metal
keywords[3]=piping
```

If you removed `productX`, the values for metal and piping are moved up and the keywords property now contains the following:

```
keywords[0]=engineering
keywords[1]=metal
keywords[2]=piping
```

7.10.3.1 Performance tip for repeating attributes

The time it takes the server to append or insert a value for a repeating property increases in direct proportion to the number of values in the property. Consequently, if you want to define a repeating property for a type and you expect that property to hold hundreds or thousands of values, it is recommended that you create an RDBMS table to hold the values instead and then register the table. When you query the type, you can issue a `SELECT` statement that joins the type and the table.

7.10.4 Adding content

There are several methods in the `IDfSysObject` interface for adding content.

7.10.4.1 Adding additional primary content

A document can have multiple primary content files, but all the files must have the same format. When you add an additional primary content files, you specify the file page number, rather than its format.

The page number must be the next number in the object sequence of page numbers. Page numbers begin with zero and increment by one. For example, if a document has three primary content files, they are numbered 0, 1, and 2. If you add another primary content file, you must assign it page number 3.

If you fail to include a page number, the server assumes the default page number, which is 0. Instead of adding the file to the existing content list, it replaces the content file previously in the 0 position.

7.10.4.2 Replacing an existing content file

To replace a primary content file, use an `insertFile` or `insertContent` method. Alternative acceptable methods are `setFileEx` or `setContentEx`. The new file must have the same format as the other primary content files in the object.

Whichever method you use, you must identify the page number of the file you want to replace in the method call. For example, suppose you want to replace the current table of contents file in a document referenced as `mySysObject` and the current table of contents file is page number 2. The following call replaces that file in the object `mySysObject`:

```
mySysObject.insertFile("toc_new",2)
```

7.10.5 Removing content from a document

To remove a content file from a document, use a `removeContent` method. You must specify the page number of the content you want to remove. If you remove a content file from the middle of a multi-paged document, the remaining pages are automatically renumbered.

You cannot remove a content page if the content has a rendition with the keep flag set to true and the page is not the last remaining page in the document.

7.10.6 Sharing a content file

Multiple objects can share one content file. You can bind a single content file to any number of objects. Content files are shared using a `bindFile` method. After a content file is saved as a primary content file for a particular object, you can use a `bindFile` method to add the content file as primary content to any number of other objects. The content file can have different page numbers in each object.

However, all objects that share the content must have the same value in their `a_content_type` attributes. If an object to which you are binding the content has no current primary content, the `bindFile` method sets the target document `a_content_type` attribute to the format of the content file.

Regardless of how many objects share the content file, the file has one content object in the repository. The documents that share the content file are recorded in the `parent_id` attribute of the content object.

7.10.7 Writing changes to the repository

The following methods write changes to the repository:

- `checkin`
- `checkinEx`
- `save`
- `saveLock`

7.10.7.1 Checkin and checkinEx methods

Use `checkin` or `checkinEx` to create a new version of a object. You must have at least `Version` permission for the object. The methods work only on checked-out documents.

The `checkinEx` method is specifically for use in applications. It has four arguments an application can use for its specific needs. Refer to the *Javadocs* for details.

Both methods return the object ID of the new version.

7.10.7.2 Save and saveLock methods

Use a `save` or `saveLock` method when you want to overwrite the version that you checked out or fetched. To use either, you must have at least `Write` permission on the object. A `save` method works on either checked-out or fetched objects. A `saveLock` method works only on checked-out objects.

If the document has been signed using `addESignature`, using save to overwrite the signed version invalidates the signatures and will prohibit the addition of signatures on future versions.

- For information about the list of the changeable properties in immutable objects, see "Attributes that remain changeable" on page 124.
- For information, see "Application-level control of SysObjects" on page 85.
- For information about the types of locks and locking strategies, see "Concurrent access control" on page 125.
- For information about the `CREATE OBJECT` and `UPDATE OBJECT` statements, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.

7.11 Managing permissions

Access permissions for an object of type SysObject or its subtypes are controlled by the ACL associated with the object. Each object has one associated ACL. An ACL is assigned to each SysObject when the SysObject is created. That ACL can be modified or replaced as needed, as the object moves through its lifecycle.

7.11.1 Default ACLs

If a user or application creates and saves a new object without explicitly assigning an ACL or permissions to the object, Documentum CM Server assigns a default ACL. The ACL designated as the default ACL is recorded in the server config object of the Documentum CM Server, in the `default_acl` property. The designated ACL can be any of the following ACLs:

- The ACL associated with the object primary folder

  An object primary folder is the folder in which the object is first stored when it is created. If the object was placed directly in a cabinet, the server uses the ACL associated with the cabinet as the folder default.
- The ACL associated with the object creator

  Every user object has an ACL. It is not used to provide security for the user but only as a potential default ACL for any object created by the user.
- The ACL associated with the object type

  Every object type has an ACL associated with its type definition. You can use that ACL as a default ACL for any object of the type.

In a newly configured repository, the `default_acl` property is set to the value identifying the user ACL as the default ACL. You can change the setting through Documentum Administrator.

7.11.2 Template ACLs

A template ACL is identified by a value of 1 in the `acl_class` property of its `dm_acl` object. A template ACL typically uses aliases in place of actual user or group names in the access control entries in the ACL. When the template is assigned to an object, Documentum CM Server resolves the aliases to actual user or group names.

Template ACLs are used to make applications, workflows, and lifecycles portable. For example, an application that uses a template ACL could be used by a variety of departments within an enterprise because the users or groups within the ACL entries are not defined until the ACL is assigned to an actual document.

7.11.3 Assigning ACLs

When you create a document or other object, you can:

- Assign a default ACL (either explicitly or allow the server to choose)

  A Documentum CM Server automatically assigns the designated default ACL to new objects if the user or application does not explicitly assign a different ACL or does not explicitly grant permissions to the object.

- Assign an existing nondefault ACL

  A document owner or a superuser can assign any private ACL that you own or any public ACL, including any system ACL, to the document.

  If the application is designed to run in multiple contexts with each having differing access requirements, assigning a template ACL is recommended. The aliases in the template are resolved to real user or group names appropriate for the context in the new system ACL.

  To assign an ACL, set the `acl_name` and, optionally, the `acl_domain` attributes. You must set the `acl_name` attribute. When only the `acl_name` is set, Documentum CM Server searches for the ACL among the ACLs owned by the current user. If none is found, the server looks among the public ACLs.

  If `acl_name` and `acl_domain` are both set, the server searches the given domain for the ACL. You must set both attributes to assign an ACL owned by a group to an object.

- Generate a custom ACL for the object

7.11.3.1 Generating custom ACLs

Custom ACLs are created by using a `grantPermit` or `revokePermit` method against an object to define access control permissions for the object. There are four common situations that generate a custom ACL:

- Granting permissions to a new object without assigning an ACL

  The server creates a custom ACL when you create a SysObject and grants permissions to it, but does not explicitly associate an ACL with the object.

  The server bases the new ACL on the default ACL identified in the `default_acl` property of the server config object. It copies that ACL, makes the indicated changes, and then assigns the custom ACL to the object.

- Modifying the ACL assigned to an new object

  The server creates a custom ACL when you create a SysObject, associate an ACL with the object, and then modify the access control entries in the ACL before saving the object. To identify the ACL to be used as the basis for the custom ACL, use a `useACL` method.

  The server copies the specified ACL, applies the changes to the copy, and assigns the new ACL to the document.

- Using `grantPermit` when no default ACL is assigned

  The server creates a custom ACL when you create a new document, direct the server not to assign a default ACL, and then use a `grantPermit` method to specify access permissions for the document. In this situation, the object's owner is not automatically granted access to the object. If you create a new document this way, be sure to set the owner's permission explicitly.

  To direct the server not to assign a default ACL, you issue a `useacl` method that specifies none as an argument.

  The server creates a custom ACL with the access control entries specified in the `grantPermit` methods and assigns the ACL to the document. Because the `useacl` method is issued with none as an argument, the custom ACL is not based on a default ACL.

- Modifying the current, saved ACL

  If you fetch an existing document and change the entries in the associated ACL, the server creates a new custom ACL for the document that includes the changes. The server copies the document's current ACL, applies the specified changes to the copy, and then assigns the new ACL to the document.

A custom ACL name is created by the server and always begins with `dm_`. Generally, a custom ACL is only assigned to one object. However, a custom ACL can be assigned to multiple objects.

For more information about ACLs, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)* and *OpenText Documentum Content Management - Foundation Java API Development Guide (EDCPKCL250400-DGD)*.

7.11.4 Rooms and ACL assignments

Objects that are created in or moved to a collaborative room are assigned the default ACL for that room. Similarly, if you move an object to a room, the current ACL is removed and the default ACL for the room is applied.

If the object is moved out of the room, Documentum CM Server removes the default room ACL and assigns a new ACL:

- If the user moving the object out of the room is the object owner, Documentum CM Server assigns the default ACL defined in the repository configuration to the object.
- If the user moving the object out of the room is not the object owner, Documentum CM Server assigns the object owner default ACL to the object.

7.11.5 Removing permissions

At times, you might need to remove user access or extended permissions to a document. For example, an employee might leave a project or be transferred to another location. A variety of situations can make it necessary to remove user permissions.

You must be the owner of the object, a superuser, or have Change Permit permission to change the entries in an object's ACL.

You can revoke any of the following permit types:

- `AccessRestriction` or `ExtendedRestriction`
- `RequiredGroup` or `RequiredGroupSet`
- `ApplicationPermit` or `ApplicationRestriction`

When you remove user access or extended permissions, you can either:

- Remove permissions to one document

  To remove permissions to a document, call the `IDfSysobject revokePermit` method against the document. The server copies the ACL, changes the copy, and assigns the new ACL to the document. The original ACL is not changed. The new ACL is a custom ACL.
- Remove permissions to all documents using a particular ACL

  To remove permissions to all documents associated with the ACL, you must alter the ACL. To do that, call the `IDfACL revokePermit` method against the ACL. Documentum CM Server modifies the specified ACL. Consequently, the changes affect all documents that use that ACL.

Use a `revokePermit` method to remove object-level permissions. That method is defined for both the `IDfACL` and `IDfSysObject` interfaces.

Each execution of `revokePermit` removes a specific entry. If you revoke an entry whose permit type is `AccessPermit` without designating the specific base permission to be removed, the `AccessPermit` entry is removed, which also removes any extended permissions for that user or group. If you designate a specific base permission level, only that permission is removed but the entry is not removed if there are extended permissions identified in the entry.

If the user or group has access through another entry, the user or group retains that access permission. For example, suppose `janek` has access as an individual and also as a member of the group `engr` in a particular ACL. If you issue a `revokePermit` method for `janek` against that ACL, you remove only janek's individual access. The access level granted through the `engr` group is retained.

7.11.6 Replacing an ACL

It is possible to replace the ACL assigned to an object with another ACL. To do so requires at least Write permission on the object. Users typically replace an ACL using facilities provided by a client interface. To replace the ACL programmatically, reset the object attributes `acl_name`, `acl_domain`, or both. These two attributes identify the ACL assigned to an object.

For information about types of ACLs, types of entries, and how to create ACLs and the entries, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

7.12 Managing content across repositories

In a multirepository installation, users are not limited to working only with the objects found in the repository to which they connect when they open a session (the local repository). Within a session, users can also work with objects in the remote repositories, the other repositories in the distributed configuration. For example, they might create a virtual document in the local repository and add a document from a remote repository as a component. Or, they might find a remote document in their inbox when they start a session with the local repository.

Similar to other users, applications can also work with remote objects. After an application opens a session with a repository, it can work with remote objects by opening a session with the remote repository or by working with the mirror or replica object in the current repository that refers to the remote object. Mirror objects and replica objects are implemented as reference links.

A reference link is a pointer in one repository to an object in another repository. A reference link is a combination of a `dm_reference` object and a mirror object or a `dm_reference` object and a replica object.

A mirror object is an object in one repository that mirrors an object in another repository. The term mirror object describes the object function. It is not a type name. For example, if you check out a remote document, the system creates a document in the local repository that is a mirror of the remote document. The mirror object in the local repository is an object of type `dm_document`.

Mirror objects only include the original object attribute data. When the system creates a mirror object, it does not copy the object content to the local repository.

Note: If the repository in which the mirror object is created is running on Sybase, values in some string attributes may be truncated in the mirror object. The length definition of some string attributes is shortened when a repository is implemented on Sybase.

Replicas are copies of an object. Replicas are generated by object replication jobs. A replication job copies objects in one repository to another. The copies in the target repository are called replicas.

- *OpenText Documentum Content Management - Server and Server Extensions Installation Guide (EDCSY250400-IGD)* contains complete information about:
  - Reference links and the underlying architecture that supports them
  - How operations on mirror objects and replicas are handled
  - Object replication

7.13 Relationships between objects

A relationship is a formal association between two objects in the repository. One object is designated as the parent and one object is designated as the child. Before you can connect two objects in a relationship, the relationship must be described in the repository. Types of relationships are defined in `dm_relation_type` objects and instances of relationship types are recorded in `dm_relation` objects.

The definition of the relationship, recorded in the `dm_relation_type` object, names the relationship and defines some characteristics, such as the security applied to the relationship and the behavior if one of the objects involved in an instance of the relationship is deleted from the repository.

A relation object identifies the two objects involved in the relationship and the type of relationship. Relation objects also have some attributes that you can use to manage and manipulate the relationship.

7.13.1 System-defined relationships

Installing Documentum CM Server installs a set of system-defined relationships. For example, annotations are implemented as a system-defined relationship between a SysObject, generally a document, and a note object. Another system-defined relationship is `DM_TRANSLATION_OF`, used to create a relationship between a source document and a translated version of the document.

You can obtain the list of system-defined relationships by examining the `dm_relation_type` objects in the repository. The relation name of system-defined relationships begin with `dm_`.

7.13.2 User-defined relationships

You can create custom relationships. Additionally, the `dm_relation` object type can be subtyped, so that you can create relationships between objects that record business-specific information, if needed.

User-defined relationships are not managed by Documentum CM Server. The server only enforces security for user-defined relationships. Applications must provide or invoke user-written procedures to enforce any behavior required by a user-defined relationship. For example, suppose you define a relationship between two document subtypes that requires a document of one subtype to be updated automatically when a document of the other subtype is updated. The server does not perform this kind of action. You must write a procedure that determines when the first document is updated and then updates the second document.

- For more information about relationships, including instructions for creating relationship types and relationships between objects, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.
- For more information about how the system-defined translation relationship can be used, see “Managing translations” on page 148.
- For more information about annotations and how to work with them, see “Annotation relationships” on page 149.

7.14 Managing translations

Documents are often translated into multiple languages. Documentum CM Server supports managing translations with two features:

- The `language_code` attribute defined for SysObjects
- The built-in relationship functionality

The `language_code` attribute allows you identify the language in which the content of a document is written and the document country of origin. Setting this attribute will allow you to query for documents based on their language. For example, you might want to find the German translation of a particular document or the original of a Japanese translation.

7.14.1 Translation relationships

You can also use the built-in relationship functionality to create a translation relationship between two SysObjects. Such a relationship declares one object (the parent) the original and the second object (the child) a translation of the original. Translation relationships have a security type of child, meaning that security is determined by the object type of the translation. A translation relationship has the relation name of `DM_TRANSLATION_OF`.

When you define the child in the relationship, you can bind a specific version of the child to relationship or bind the child by version label. To bind a specific version, you set the `child_id` property of the `dm_relation` object to object ID of the child. To bind by version label, you set the `child_id` attribute to the chronicle ID of the version tree that contains the child, and the `child_label` to the version label of the translation. The chronicle ID is the object ID of the first version on the version tree. For example, if you want the `APPROVED` version of the translation to always be associated with the original, set `child_id` to the translation chronicle ID and `child_label` to `APPROVED`.

- *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)* contains more information about:
  - Recommended language and country codes
  - Properties defined for the relation object type

7.15 Annotation relationships

Annotations are comments that a user attaches to a document (or any other SysObject or SysObject subtype). Throughout document development, and often after it is published, people might want to record editorial suggestions and comments. For example, several managers might want to review and comment on a budget. Or perhaps several marketing writers working on a brochure want to comment on each other's work. In situations such as these, the ability to attach comments to a document without modifying the original text is very helpful.

Annotations are implemented as note objects, which are a SysObject subtype. The content file you associate with the note object contains the comments you want to attach to the document. After the note object and content file are created and associated with each other, you use the `IDfNote.addNoteEx` method to associate the note with the document. A single document can have multiple annotations. Conversely, a single annotation can be attached to multiple documents.

When you attach an annotation to a document, the server creates a relation object that records and describes the relationship between the annotation and the document. The relation object `parent_id` attribute contains the document object ID and its `child_id` attribute contains the note object ID. The `relation_name` attribute contains `dm_annotation`, which is the name of the relation type object that describes the annotation relationship.

You can create, attach, detach, and delete annotations. For information about creating, attaching, detaching, and deleting annotations, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

7.15.1 Object operations and annotations

This section describes how annotations are affected by common operations on the objects to which they are attached.

- Save, Check In, and Saveasnew:

  If you want to keep the annotations when you save, check in, or copy a document, the `permanent_link` attribute for the relation object associated with the annotation must be set to `TRUE`. This flag is `FALSE` by default.

- Destroy:

  Destroying an object that has attached annotations automatically destroys the relation objects that attach the annotations to the object. The note objects that are the annotations are not destroyed.

  Note: The `dm_clean` utility automatically destroys note objects that are not referenced by any relation object, that is, any that are not attached to at least one object.

- Object replication:

  If the replication mode is federated, then any annotations associated with a replicated object are replicated also.

- For information about description of relation objects, relation type objects, and their attributes, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.

- The associated *Javadocs* has more information about the `addNote` and `removeNote` methods in the `IDfSysObject` interface.

Chapter 8

Virtual documents

8.1 Overview

This section describes virtual documents, a feature supported by Documentum CM Server that allows you to create documents with varying formats.

Users create virtual documents using the Virtual Document Manager, a graphical user interface that allows them to build and modify virtual documents. However, if you want to write an application that creates or modifies a virtual document with no user interaction, you must use Foundation Java API.

Although the components of a virtual document can be any SysObject or SysObject subtype except folders, cabinets, or subtypes of folders or cabinets, the components are often simple documents. Be sure that you are familiar with the basics of creating and managing simple documents, described in “Content management services“ on page 109 before you begin working with virtual documents.

A virtual document is a hierarchically organized structure composed of component documents. The components of a virtual document are of type `dm_sysobject`, or a subtype of `dm_sysobject` (but excluding cabinets and folders). Most commonly, the components are of type `dm_document` or a subtype. The child components of a virtual document can be simple documents (that is, non-virtual documents), or they can themselves be virtual documents. Documentum CM Server does not impose any restrictions on the depth of nesting of virtual documents.

Note: A compound document (for example, an OLE or XML document) cannot be a child in a virtual document.

The root of a virtual document is version-specific and identified by an object identity (on Documentum CM Server, an `r_object_id`). The child components of a virtual document are not version-specific, and are identified by an `i_chronicle_id`. The relationship between a parent component and its children are defined in containment objects (`dmr_containment`), each of which connects a parent object to a single child object. The order of the children of the parent object is determined by the `order_no` property of the containment object.

The following illustration describes the relationships:

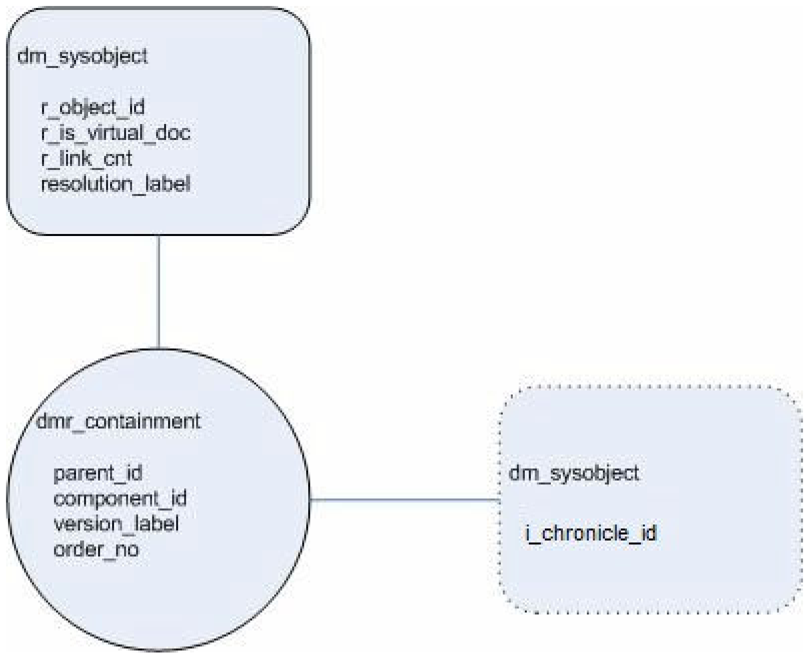


Figure 8-1: Parent-child relationships

The version of the child component is determined at the time the virtual document is `assembled`. A virtual document is assembled when it is retrieved by a client, and when a snapshot of the virtual document is created. The assembly is determined at runtime by a binding algorithm governed by metadata set on the `dmr_containment` objects.

8.1.1 Use of virtual documents

Virtual documents provide a way to combine multiple documents in multiple formats into a single document. Each component exists as an independent object in the repository. Virtual documents allow users to:

- Share document components in multiple virtual documents to manage content redundancy. When a changed component is checked in, the change is reflected in all virtual documents that include the component.
- Combine different types of related content into the same document (as an organizational tool).
- Increase flexibility of user access (multiple users can simultaneously check out and maintain different parts of the virtual document).
- Save snapshots of the virtual document that reflect the state of all components at the time the snapshot is created.

For some content types, such as Microsoft Word files and XML files used in XML applications, virtual documents are patched as they are retrieved to a client, and flattened into a single document. In other cases, the individual components of the virtual documents are retrieved as separate files.

8.1.2 Implementation

This section briefly describes how virtual documents are implemented within the OpenText Documentum CM system.

The components of a virtual document are associated with the containing document by containment objects. Containment objects contain information about the components of a virtual document. Each time you add a component to a virtual document, a containment object is created for that component. Containment objects store the information that links a component to a virtual document. For components that are themselves virtual documents, the objects also store information that the server uses when assembling the containing document.

You can associate a particular version of a component with the virtual document or you can associate the entire component version tree with the virtual document. Binding the entire version tree to the virtual document allows you to select which version is included at the time you assemble the document. This feature provides flexibility, letting you assemble the document based on conditions specified at assembly time.

The components of a virtual document are ordered within the document. By default, the order is managed by the server. The server automatically assigns order numbers when you add or insert a component.

If you bypass the automatic numbering provided by the server, you can use your own numbers. The `insertPart`, `updatePart`, and `removePart` methods allow you to specify order numbers. However, if you define order numbers, you must also perform the related management operations. The server does not manage user-defined ordering numbers.

The number of direct components contained by a virtual document is recorded in the document's `r_link_cnt` property. Each time you add a component to a virtual document, the value of this property is incremented by 1.

The `r_is_virtual_doc` property is an integer property that helps determine whether OpenText Documentum CM client applications treat the object as a virtual document. If the property is set to 1, the client applications always open the document in the Virtual Document Manager. The property is usually set to 1 when you use the Virtual Document Manager to add the first component to the containing document. Programmatically, you can set it using the `IDfSysObject.setIsVirtualDocument` method. You can set the property for any SysObject subtype except folders, cabinets, and their subtypes.

However, clients will also treat an object as a virtual document if `r_is_virtual_doc` is set to 0, and `r_link_cnt` is greater than 0. A document is not a virtual document only when both properties are set to 0. If either property is not 0, the object is treated as a virtual document.

8.1.3 Versioning

You can version a virtual document and manage its versions just as you do a simple document.

8.1.4 Deleting virtual documents and components

Deleting a virtual document version also removes the containment objects and any assembly objects associated with that version.

By default, Documentum CM Server does not allow you to remove an object from the repository if the object belongs to a virtual document. This ensures that the referential integrity of virtual documents is maintained. This behavior is controlled by the `compound_integrity` property in the server config object of the server. By default, this property is `TRUE`, which prohibits users from destroying any object contained in a virtual document.

If you set this property to `FALSE`, users can destroy components of unfrozen virtual documents. However, users can never destroy components of frozen virtual documents, regardless of the setting of `compound_integrity`.

You must have SysAdmin or Superuser privileges to set the `compound_integrity` property.

8.1.5 Assembling the virtual document

Documentum CM Server supports conditional assembly and snapshots for virtual documents. Both are features that allow you to see the document as an assembled whole.

- Conditional assembly:

  Assembling a virtual document selects a set of the document components for publication or some other operation, such as viewing or copying. Conditional assembly lets you identify which components to include. You can include all the components or only some of them. If a component version tree is bound to the virtual document, you can choose not only whether to include the component in the document but also which version of the component to include.

  If a selected component is also a virtual document, the component descendants can also be included. Whether descendants are included is controlled by two properties in the containment objects.

- Snapshots:

  Snapshots provide a way of persistently storing the results of virtual document assembly. The snapshot records the exact components of the virtual document at the time the snapshot was created, using version-specific object identities to represent each node.

  Snapshots are stored in the repository as a set of assembly objects (`dm_assembly`) associated with a `dm_sysobject`. Each assembly object in a snapshot represents one node of the virtual document, and connects a parent document with a specific version of a child document.

The following figure illustrates assembly relationships:

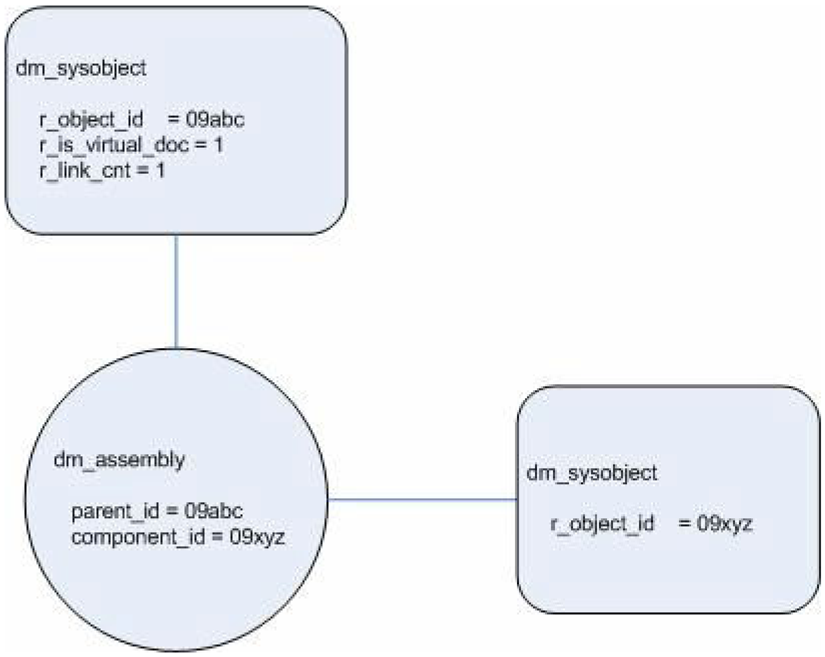


Figure 8-2: Assembly relationships

8.1.6 Virtual documents and content files

Typically, virtual documents do not have content files. However, because a virtual document is created from a SysObject or SysObject subtype, any virtual document can have content files in addition to component documents. If you do associate a content file with a virtual document, the file is managed just as if it belonged to a simple document and is subject to the same rules. For example, similar to the content files belonging to a simple document, all content files associated with a virtual document must have the same format.

8.1.7 XML support

XML documents are supported as virtual documents in Documentum CM Server. When you import or create an XML document using the Foundation Java API, the document is created as a virtual document. Other documents referenced in the content of the XML document as entity references or links are automatically brought into the repository and stored as directly contained components of the virtual document.

The connection between the parent and the components is defined in two properties of containment objects: `a_contain_type` and `a_contain_desc`. Foundation Java API uses the `a_contain_type` property to indicate whether the reference is an entity or link. It uses the `a_contain_desc` to record the actual identification string for the child.

These two properties are also defined for the `dm_assembly` type, so applications can correctly create and handle virtual document snapshots using the Foundation Java API.

To reference other documents linked to the parent document, you can use relationships of type `xml_link`.

Virtual documents with XML content are managed by XML applications, which define rules for handling and chunking the XML content.

8.1.8 Virtual documents and retention policies

You can associate a virtual document with a retention policy. Retention policies control an object's retention in the repository. They are applied using Retention Policy Services.

If a virtual document is subject to a retention policy, you cannot add, remove, or rearrange its components.

- For information about how versioning is handled for documents, see “Versioning” on page 117.
- For information about the properties that control conditional assembly for contained virtual documents, see “Defining component assembly behavior” on page 159.
- For information about creating and working with snapshots, see “Snapshots” on page 161.
- For information about how early and late binding work, see “Virtual document assembly and binding” on page 156.

8.2 Virtual document assembly and binding

A virtual document is assembled when it is retrieved by a client, and when a snapshot of the virtual document is created and stored in the repository.

Each virtual document node can be early or late bound.

- In early binding, the binding label is set on the containment object when the node is created and stored persistently. The binding label is stored in the `version_label` property of the `dmr_containment` object.
- In late binding, the version of the node is determined at the time the virtual document is assembled, using a “preferred version” or late binding label passed at runtime. If the `version_label` property of the `dmr_containment` object is empty or null, then the node is late bound.

The logic that controls the assembly of the virtual document at the time it is retrieved is determined by settings on the containment objects. The following table describes the binding logic:

API term	Documentum CM Server property	Description
`binding`	`version_label`	The early binding label of the virtual document node. If empty, then the node is late bound.
`overrideLateBinding`	`use_node_vers_label`	Override the late binding value for all descendants of this node, using the early bound label of this node.
`includeBrokenBindings`	`none` (provided by client API at runtime)	A broken binding occurs when there is no version label on the node corresponding to the `lateBindingValue`. If broken nodes are included, uses the `CURRENT` version of the node.

The following illustration describes the decision process when assembling a virtual document node:

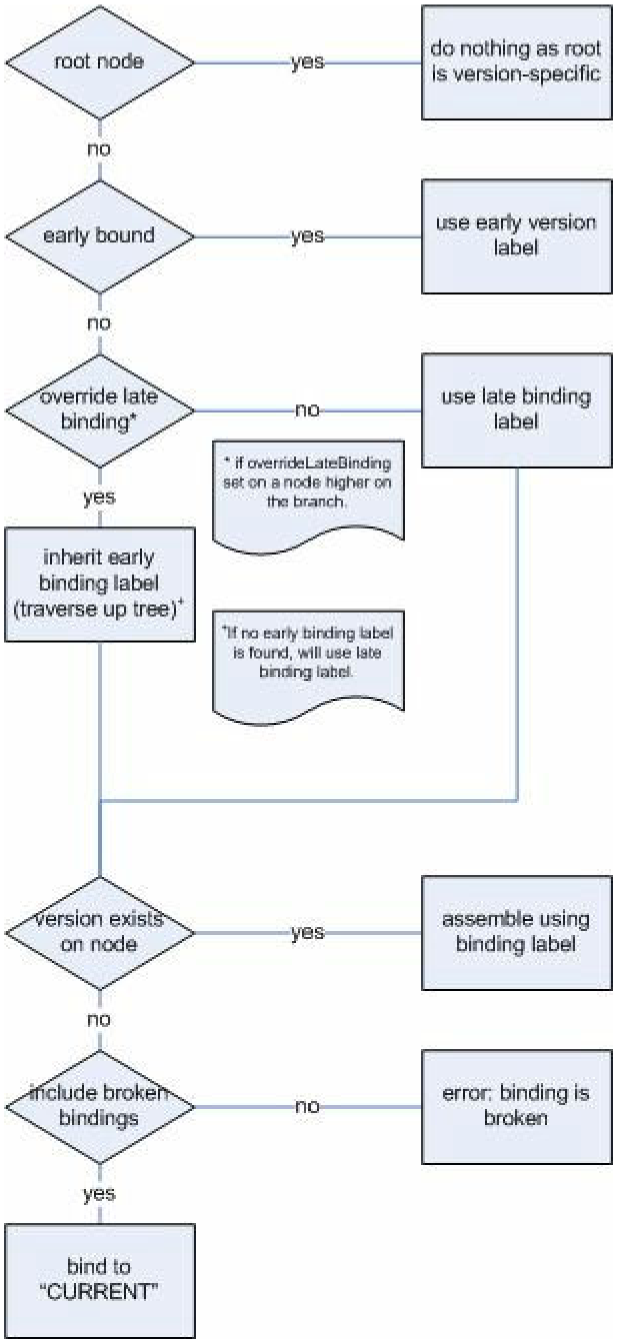

root node
yes
do nothing as root is version-specific
no
early bound
yes
use early version label
no
override late binding*
no
use late binding label
* if overrideLateBinding set on a node higher on the branch.
yes
inherit early binding label (traverse up tree)⁺
⁺If no early binding label is found, will use late binding label.
version exists on node
yes
assemble using binding label
no
include broken bindings
no
error: binding is broken
yes
bind to "CURRENT"

Figure 8-3: Decision process when assembling virtual document node

8.3 Defining component assembly behavior

There are two properties in containment objects that control how components that are themselves virtual documents behave when the components are selected for a snapshot. The properties are `use_node_ver_label` and `follow_assembly`. In an application, they are set by arguments in `appendPart`, `insertPart`, and `updatePart` methods.

8.3.1 use_node_ver_label

The `use_node_ver_label` property determines how the server selects late-bound descendants of an early-bound component.

If a component is early bound and `use_node_ver_label` in its associated containment object is set to `TRUE`, the server uses the component early bound version label to select all late-bound descendants of the component. If another early bound component is found that has `use_node_ver_label` set to `TRUE`, then that component label is used to resolve descendants from that point.

Late bound components that have no early bound parent or that have an early bound parent with `use_node_ver_label` set to `FALSE` are chosen by the binding conditions specified in the `SELECT` statement.

Figure 8-4 illustrates how `use_node_ver_label` works. In the figure, each component is labeled as early or late bound. For the early bound components, the version label specified when the component was added to the virtual document is shown. Assume that all the components in the virtual document have `use_node_ver_label` set to `TRUE`.

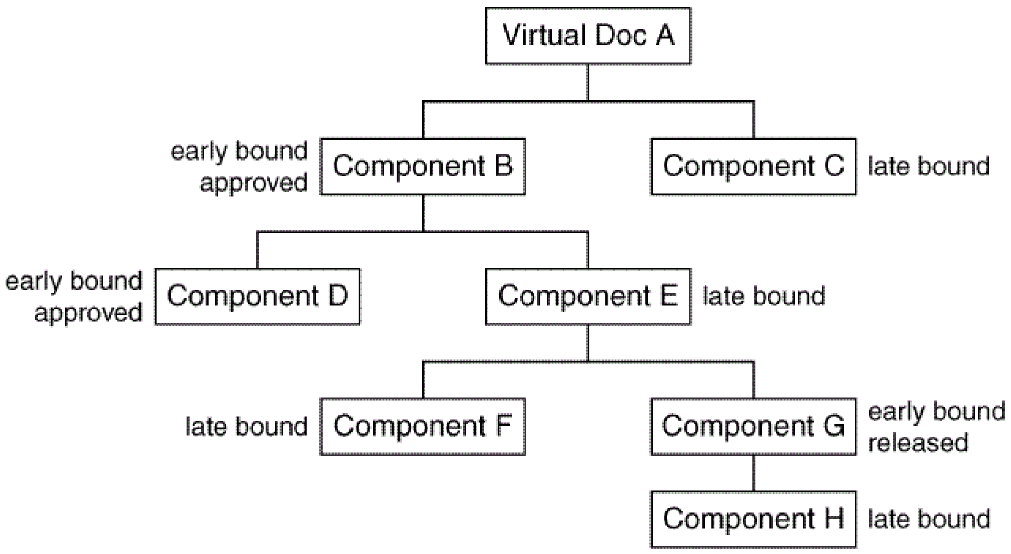


Figure 8-4: Virtual document with node bindings

Component B is early bound-the specified version is the one carrying the approved version label . Because Component B is early bound and `use_node_ver_label` is set to `TRUE`, when the server determines which versions of the Component B late bound descendants to include, it will choose the versions that have the approved symbolic version label. In our sample virtual document, Component E is a late-bound descendant of Component B. The server will pick the approved version of Component E for inclusion in the virtual document.

Descending down the hierarchy, when the server resolves the Component E late bound descendant, Component F, it again chooses the version that carries the approved version label. All late-bound descendant components are resolved using the version label associated with the early-bound parent node until another early bound component is encountered with `use_node_ver_label` set to `TRUE`.

In the example, Component G is early bound and has `use_node_ver_label` set to `TRUE`. Consequently, when the server resolves any late bound descendants of Component G, it will use the version label associated with Component G, not the label associated with Component B. The early bound version label for Component G is released. When the server chooses which version of Component H to use, it picks the version carrying the released label.

Component C, although late bound, has no early bound parent. For this component, the server uses the binding condition specified in the `IN DOCUMENT` clause to determine which version to include. If the `IN DOCUMENT` clause does not include a binding condition, the server chooses the version carrying the `CURRENT` label.

8.3.2 follow_assembly

The `follow_assembly` property determines whether the server selects component descendants using the containment objects or a snapshot associated with the component.

If you set `follow_assembly` to `TRUE`, the server selects component descendants from the snapshot associated with the component. If `follow_assembly` is `TRUE` and a component has a snapshot, the server ignores any binding conditions specified in the `SELECT` statement or mandated by the `use_node_ver_label` property.

If `follow_assembly` is `FALSE` or a component does not have a snapshot, the server uses the containment objects to determine component descendants.

8.4 Copy behavior

When a user copies a virtual document, the server can make a copy of each component or it can create an internal reference or pointer to the source component. The pointer or reference is internal. It is not an instance of a `dm_reference` object. Which option is used is controlled by the `copy_child` property in a component containment object. It is an integer property with three valid settings:

- 0, which means that the copy or reference choice is made by the user or application when the copy operation is requested
- 1, which directs the server to create a pointer or reference to the component
- 2, which directs the server to copy the component

Whether the component is copied or referenced, a new containment object for the component linking the component to the new copy of the virtual document is created.

Regardless of which option is used, when users open the new copy in the Virtual Document Manager, all document components are visible and available for editing or viewing, subject to user access permissions.

8.5 Snapshots

A snapshot is a record of the virtual document as it existed at the time you created the snapshot. Snapshots are a useful shortcut if you often assemble a particular subset of virtual document components. Creating a snapshot of that subset of components lets you assemble the set quickly and easily.

A snapshot consists of a collection of assembly objects. Each assembly object represents one component of the virtual document. All the components represented in the snapshot are absolutely linked to the virtual document by their object IDs.

Only one snapshot can be assigned to each version of a virtual document. If you want to define more than one snapshot for a virtual document, you must assign the additional snapshots to other documents created specifically for the purpose.

8.5.1 Modifying snapshots

You can add or delete components (by adding or deleting the assembly object representing the component) or you can modify an existing assembly object in a snapshot.

Any modification that affects a snapshot requires at least Version permission on the virtual document for which the snapshot was defined.

8.5.1.1 Adding new assembly objects

If you add an assembly object to an snapshot programmatically, be sure to set the following properties of the new assembly object:

- `book_id`, which identifies the topmost virtual document containing this component. Use the document object ID.
- `parent_id`, which identifies the virtual document that directly contains this component. Use the document object ID.
- `component_id`, which identifies the component. Use the component object ID.
- `comp_chronicle_id`, which identifies the chronicle ID of the component.
- `depth_no`, which identifies the depth of the component within the document specified in the `book_id`.
- `order_no`, which specifies the position of the component within the virtual document. This property has an integer datatype. You can query the `order_no` values for existing components to decide which value you want to assign to a new component.

You can add components that are not actually part of the virtual document to the document snapshot. However, doing so does not add the component to the virtual document in the repository. That is, the virtual document `r_link_cnt` property is not incremented and a containment object is not created for the component.

8.5.1.2 Deleting an assembly object

Deleting an assembly object only removes the component represented by the assembly object from the snapshot. It does not remove the component from the virtual document. You must have at least `Version` permission for the topmost document (the document specified in the assembly object `book_id` property) to delete an assembly object.

To delete a single assembly object or several assembly objects, use a destroy method. Do not use destroy to delete each object individually in an attempt to delete the snapshot.

8.5.1.3 Changing an assembly object

You can change the values in the properties of an assembly object. However, if you do, be very sure that the new values are correct. Incorrect values can cause errors when you attempt to query the snapshot. Snapshots are queried using the `USING ASSEMBLIES` option of the `SELECT` statement `IN DOCUMENT` clause.

8.5.2 Deleting a snapshot

Use a `IDfSysObject.disassemble` method to delete a snapshot. This method destroys the assembly objects that make up the snapshot. You must have at least `Version` permission for a virtual document to destroy its snapshot.

8.6 Frozen virtual documents and snapshots

A frozen virtual document or snapshot is a document that has been explicitly marked as immutable by an `IDfSysObject.freeze` method. Users cannot modify the content or properties of a frozen virtual document or of the frozen snapshot components. Nor can they add or remove snapshot components.

Issuing the freeze method automatically freezes the target virtual document. Freezing the associated snapshot is optional. If the document has multiple snapshots, only the snapshot actually associated with the virtual document itself can be frozen. The other snapshots, associated with simple documents, are not frozen.

If you want to freeze only the snapshot, you must freeze both the virtual document and the snapshot and then explicitly unfreeze the virtual document.

Users are allowed to modify any components of the virtual document that are not part of the frozen snapshot. Although users cannot remove those components from the document, they can change the component content files or properties.

8.6.1 Freezing a document

Freezing sets the following properties of the virtual document to `TRUE`:

- `r_immutable_flag`

  This property indicates that the document is unchangeable.

- `r_frozen_flag`

  This property indicates that the `r_immutable_flag` was set by a freeze method (instead of a `checkin` method).

If you freeze an associated snapshot, the `r_has_frzn_assembly` property is also set to `TRUE`.

Freezing a snapshot sets the following properties for each component in the snapshot:

- `r_immutable_flag`
- `r_frzn_assembly_cnt`

  The `r_frzn_assembly` count property contains a count of the number of frozen snapshots that contain this component. If this property is greater than zero, you cannot delete or modify the object.

8.6.2 Unfreezing a document

Unfreezing a document makes the document changeable again.

Unfreezing a virtual document sets the following properties of the document to `FALSE`:

- `r_immutable_flag`

  If the `r_immutable_flag` was set by versioning prior to freezing the document, then unfreezing the document does not set this property to `FALSE`. The document remains unchangeable even though it is unfrozen.

- `r_frozen_flag`

If you chose to unfreeze the document snapshot, the server also sets the `r_has_frzn_assembly` property to `FALSE`.

Unfreezing a snapshot resets the following properties for each component in the snapshot:

- `r_immutable_flag`

  This is set to `FALSE` unless it was set to `TRUE` by versioning prior to freezing the snapshot. In such cases, unfreezing the snapshot does not reset this property.

- `r_frzn_assembly_cnt`

  This property, which contains a count of the number of frozen snapshots that contain this component, is decremented by 1.

Chapter 9

Workflows

This chapter describes workflows, part of the process management services of Documentum CM Server. Workflows allow you to automate business processes.

9.1 Overview

A workflow is a sequence of activities that represents a business process, such as an insurance claims procedure or an engineering development process. Workflows can describe simple or complex business processes. Workflow activities can occur one after another, with only one activity in progress at a time. A workflow can consist of multiple activities all happening concurrently. A workflow might combine serial and concurrent activity sequences. You can also create a cyclical workflow, in which the completion of an activity restarts a previously completed activity.

9.1.1 Implementation

Workflows are implemented as two separate parts: a workflow definition and a runtime instantiation of the definition.

The workflow definition is the formalized definition of the business process. A workflow definition has two major parts, the structural, or process, definition and the definitions of the individual activities. The structural definition is stored in a `dm_process` object. The definitions of individual activities are stored in `dm_activity` objects. Storing activity and process definitions in separate objects allows activity definitions to be used in multiple workflow definitions. When you design a workflow, you can include existing activity definitions in addition to creating any new activity definitions needed.

When a user starts a workflow, the server uses the definition in the `dm_process` object to create a runtime instance of the workflow. Runtime instances of a workflow are stored in `dm_workflow` objects for the duration of the workflow. When an activity starts, it is instantiated by setting properties in the workflow object. Running activities may also generate work items and packages. Work items represent work to be performed on the objects in the associated packages. Packages generally contain one or more documents.

The following illustration describes how the components of a workflow definition and runtime instance work together:

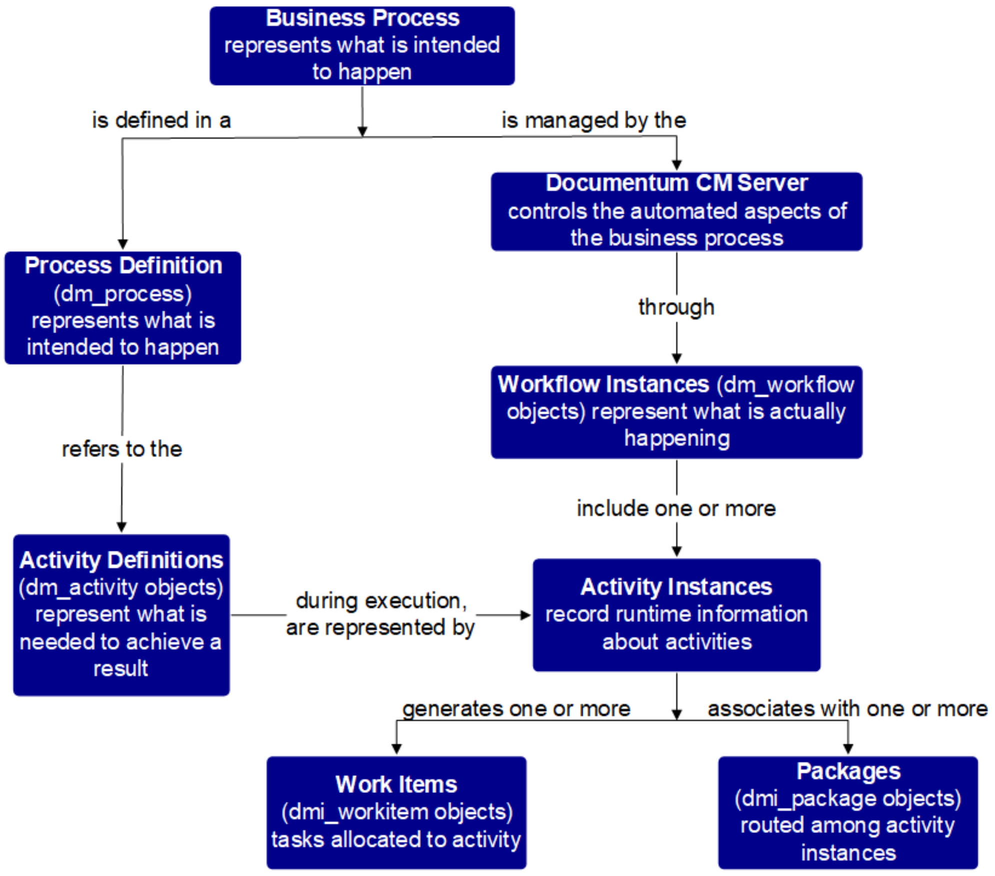

Business Process
represents what is intended to happen
is defined in a
is managed by the
Documentum CM Server
controls the automated aspects of the business process
Process Definition
(dm_process)
represents what is intended to happen
through
Workflow Instances (dm_workflow objects) represent what is actually happening
refers to the
include one or more
Activity Definitions
(dm_activity objects)
represent what is needed to achieve a result
during execution, are represented by
Activity Instances
record runtime information about activities
generates one or more
associates with one or more
Work Items
(dmi_workitem objects)
tasks allocated to activity
Packages
(dmi_package objects)
routed among activity instances

Figure 9-1: Workflow components

Users can repeatedly perform the business process. It is based on a stored definition, and the essential process is the same each time. Separating a workflow definition from its runtime instantiation allows multiple workflows based on the same definition to run concurrently.

9.1.2 Template workflows

You can create a template workflow definition, a workflow definition that can be used in many contexts. This is done by including activities whose performers are identified by aliases instead of actual performer names. When aliases are used, the actual performers are selected at runtime.

For example, a typical business process for new documents has four steps: authoring the document, reviewing it, revising it, and publishing the document. However, the actual authors and reviewers of various documents will be different people. Rather than creating a new workflow for each document with the authors and reviewers names hard-coded into the workflow, create activity definitions for the basic steps that use aliases for the authors and reviewers names and put those definitions in one workflow definition. Depending on how you design the workflow, the actual values represented by the aliases can be chosen by the workflow supervisor when the workflow is started or later, by the server when the containing activity is started.

Installing Documentum CM Server installs one system-defined workflow template. Its object name is `dmSendToList2`. It allows a user to send a document to multiple users simultaneously. This template is available to users of Documentum Webtop through the **Tools** menu.

9.2 Workflow definitions

A workflow definition consists of:

- one process definition
- a set of activity definitions
- port and package definitions

The following sections provide some basic information about the components of a definition.

9.2.1 Process definitions

A process definition defines the structure of a workflow. The structure represents a picture of the business process emulated by the workflow. Process definitions are stored as `dm_process` objects. A process object has properties that identify the activities that make up the business process, a set of properties that define the links connecting the activities, and a set of properties that define the structured data elements and correlation sets that may be associated the workflow. It also has properties that define some behaviors for the workflow when an instance is running.

Note: Structured data elements and correlation sets for a workflow may only be defined using Business Process Manager.

9.2.1.1 Activity types in a process definition

Activities represent the tasks that comprise the business process. When you create a workflow definition, you must decide how to model your business process in the sequence of activities that make up a workflow structure.

Each activity in a workflow is defined as one of the following kinds of activities:

- Initiate

  Initiate activities link to a Begin activity. These activities record how a workflow may be started. For example, a workflow might have two Initiate activities, one that allows the workflow to be started manually from Documentum Webtop, and one that allows the workflow to be started by submitting a form. Initiate activities may only be linked to Begin activities.

- Begin

  Begin activities start the workflow. A process definition must have at least one beginning activity.
- Step

  Step activities are the intermediate activities between the beginning and the end. A process definition can have any number of Step activities.
- End

  An End activity is the last activity in the workflow. A process definition can have only one ending activity.
- Exception

  An exception activity is associated with an automatic activity, to provide fault-handling functionality for the activity. Each automatic activity can have one exception activity.

You can use activity definitions more than once in a workflow definition. For example, suppose you want all documents to receive two reviews during the development cycle. You might design a workflow with the following activities: `Write`, `Review1`, `Revise`, `Review2`, and `Publish`. The `Review1` and `Review2` activities can be the same activity definition.

An activity that can be used more than once is called a repeatable activity. Whether an activity is repeatable is defined in the activity's definition.

A repeatable activity is an activity that can be used more than once in a particular workflow. By default, activities are defined as repeatable activities.

The `repeatable_invoke` property controls this feature. It is `TRUE` by default. To constrain an activity's use to only once in a workflow's structure, the property must be set to `FALSE`.

In a process definition, the activities included in the definition are referenced by the object IDs of the activity definitions. In a running workflow, activities are referenced by the activity names specified in the process definition.

When you add an activity to a workflow definition, you must provide a name for the activity that is unique among all activities in the workflow definition. The name you give the activity in the process definition is stored in the `r_act_name` property. If the activity is used only once in the workflow structure, you can use the name assigned to the activity when the activity was defined (recorded in the activity's `object_name` property). However, if the activity is used more than once in the workflow, you must provide a unique name for each use.

9.2.1.2 Links

A link connects two activities in a workflow through their ports. A link connects an output port of one activity to an input port of another activity. Think of a link as a one-way bridge between two activities in a workflow.

An input port on a Begin activity participates in a link, but it can only connect to an output port of an Initiate activity. Similarly, an output port of an Initiate activity may only connect to an input port of a Begin activity.

Output ports on End activities are not allowed to participate in links.

Each link in a process definition has a unique name.

9.2.2 Activity definitions

Activity definitions describe tasks in a workflow. OpenText Documentum CM implements activity definitions as `dm_activity` objects. The properties of an activity object describe the characteristics of the activity, including:

- How the activity is executed
- Who performs the work
- What starts the activity
- The transition behavior when the activity is completed

The definition also includes a set of properties that define the ports for the activities, the packages that each port can handle, and the structured data that is accessible to the activity.

9.2.2.1 Manual and automatic activities

An activity is either a manual activity or an automatic activity.

9.2.2.1.1 Manual activities

A manual activity represents a task performed by an actual person or persons. Manual activities can allow delegation or extension. Any user can create a manual activity.

9.2.2.1.2 Automatic activities

An automatic activity represents a task whose work is performed, on behalf of a user, by a script defined in a method object. Automatic activities cannot be delegated or extended. Additionally, you must have Sysadmin or Superuser privileges to create an automatic activity.

If the method executed by the activity is a Java method, you can configure the activity so that the method is executed by the `dm_bpm servlet`. This is a Java servlet dedicated to executing workflow methods. To configure the method to execute in this servlet, you must set the `a_special_app` property of the method object to a character string beginning with `workflow`. Additionally, the `classfile` of the Java method must be in a location that is included in the `classpath` of the `dm_bpm_servlet`.

If a Java workflow method is not executed by the `dm_bpm_servlet`, it is executed by the Java method server.

Note: The `dm_server_config.app_server_name` for the `dm_bpm_servlet` is `do_bpm`. The URL for the servlet is in the `app_server_uri` property, at the corresponding index position as `do_bpm` in `app_server_name`.

- For information, see “Delegation and extension” on page 171.
- For information about the instructions for creating a method for an automatic activity, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

9.2.2.2 Activity priorities

Priority values are used to designate the execution priority of an activity. Any activity may have a priority value defined for it in a process definition that contains the activity. An activity assigned to a work queue may have an additional priority assigned that is specific to the work queue. The uses of these two priority values are different.

9.2.2.2.1 Use of the priority defined in the process definition

When you create a workflow definition, you can set a priority for each activity in the workflow. The priority value is recorded in the process definition and is only applied to automatic tasks. Documentum CM Server ignores the value for manual tasks.

The workflow agent (the internal server facility that controls execution of automatic activities) uses the priority values in `r_act_priority` to determine the order of execution for automatic activities. When an automatic activity is instantiated, Documentum CM Server sends a notification to the workflow agent. In response, the agent queries the repository to obtain information about the activities ready for execution. The query returns the activities in priority order, highest to lowest.

9.2.2.2.2 Use of the work queue priority values

Every work item on a work queue is governed by a work queue policy object. The work queue policy defines how the item is handled on the queue. Among other things, the policy defines the priority of the work items on the queue. Every work item on a work queue is assigned a priority value at runtime, when the work item is generated.

The priority assigned by a work queue policy does not affect or interact with a priority value assigned to an activity in the process definition. Work queue policies are applied to manual activities, because only manual activities can be placed on a work queue. The priority values in the process definition are used by Documentum CM Server only for execution of automatic activities.

9.2.2.3 Process and activity definition states

There are three possible states for process and activity definitions: draft, validated, and installed.

A definition in the draft state has not been validated since it was created or last modified. A definition in the validated state has passed the server's validation checks, which ensure that the definition is correctly defined. A definition in the installed state is ready for use in an active workflow.

You cannot start a workflow from a process definition that is in the draft or validated state. The process definition must be in the installed state. Similarly, you cannot successfully install a process definition unless the activities it references are in the installed state.

9.2.2.4 Delegation and extension

Delegation and extension are features that you can set for manual activities.

Delegation allows the server or the activity performer to delegate the work to another performer. If delegation is allowed, it can occur automatically or be forced manually.

Automatic delegation occurs when the server checks the availability of an activity performer or performers and determines that the person or persons is not available. When this happens, the server automatically delegates the work to the user identified in the `user_delegation` property of the original performer user object.

If there is no user identified in `user_delegation` or that user is not available, automatic delegation fails. When delegation fails, Documentum CM Server reassigns the work item based on the value in the `control_flag` property of the activity object that generated the work item. If `control_flag` is set to `0` and automatic delegation fails, the work item is assigned to the workflow supervisor. If `control_flag` is set to `1`, the work item is reassigned to the original performer. The server does not attempt to delegate the task again. In either case, the workflow supervisor receives a `DM_EVENT_WI_DELEGATE_F` event.

Manual delegation occurs when an `IDfWorkitem.delegateTask` method is explicitly issued. Typically, only the work item performer, the workflow supervisor, or a superuser can execute the method.

If delegation is disallowed, automatic delegation is prohibited. However, the workflow supervisor or a superuser can delegate the work item manually.

9.2.2.4.1 Extension

Extension allows the activity performer to identify a second performer for the activity after he or she completes the activity the first time. If extension is allowed, when the original performers complete activity work items, they can identify a second round of performers for the activity. The server will generate new work items for the second round of performers. Only after the second round of performers completes the work does the server evaluate the activity transition condition and move to the next activity.

A work item can be extended only once. Programmatically, a work item is extended by execution of an `IDfWorkitem.repeat` method.

If extension is disallowed, only the workflow supervisor or a superuser can extend the work item.

Activities with multiple performers performing sequentially (user category 9), cannot be extended.

9.2.2.5 Performer choices

When you define a performer for an activity, you must first choose a performer category. Depending on the chosen category, you may also be required to identify the performer also. If so, you can either define the actual performer at that time or configure the activity to allow the performer to be chosen at one of the following times:

- When the workflow is started
- When the activity is started
- When a previous activity is completed

If you choose to define the performer during the design phase, you can either name the performer directly for many categories or define a series of conditions and associated performers. At runtime, the workflow engine determines which condition is satisfied and selects the performer defined as the choice for that condition.

There are multiple options when choosing a performer category. Some options are supported for both manual and automatic activities. Others are only valid choices for manual activities.

9.2.2.6 Task subjects

The task subject is a message that provides a work item performer with information about the work item. The message is defined in the activity definition, using references to one or more properties. At runtime, the actual message is constructed by substituting the actual property values into the string. For example, suppose the task subject is defined as:

```
Please work on the {dmi_queue_item.task_name} task

(from activity number {dmi_queue_item.r_act_seqno})of the workflow
{dmi_workflow.object_name}.

The attached package is {dmi_package_r_package_name}.
```

Assuming that `task_name` is `Review`, `r_act_seqno` is `2`, `object_name` is `Engr Proposal`, and `r_package_name` is `First Draft`, at runtime the user sees:

```
Please work on the Review task

(from activity number 2) of the workflow Engr Proposal.

The attached package is First Draft.
```

The text of a task subject message is recorded in the `task_subject` property of the activity definition. The text can be up to 255 characters and can contain references to the following object types and properties:

- `dm_workflow`, any property
- `dmi_workitem`, any property

  At runtime, references to `dmi_workitem` are interpreted as references to the work item associated with the current task.
- `dmi_queue_item`, any property except `task_subject`

  At runtime, references to `dmi_queue_item` are interpreted as references to the queue item associated with the current task.
- `dmi_package`, any property

The format of the object type and property references must be:

```
{object_type_name.property_name}
```

The server uses the following rules when resolving the string:

- The server does not place quotes around resolved object type and property references.
- If the referenced property is a repeating property, the server retrieves all values, separating them with commas.
- If the constructed string is longer than 512 characters, the server truncates the string.
- If an object type and property reference contains an error, for example, if the object type or property does not exist, the server does not resolve the reference. The unresolved reference appears in the message.

The resolved string is stored in the `task_subject` property of the associated task queue item object. After the server has created the queue item, the value of the `task_subject` property in the queue item will not change, even if the values in any referenced properties change.

9.2.2.7 Starting conditions

A starting condition defines the starting criteria for an activity. At runtime, the server will not start an activity until the activity starting condition is met. A starting condition consists of a trigger condition and, optionally, a trigger event.

The trigger condition is the minimum number of input ports that must have accepted packages. For example, if an activity has three input ports, you may decide that the activity can start when two of the three have accepted packages.

A trigger event is an event queued to the workflow. The event can be a system-defined event, such as dm_checkin, or you can make up an event name, such as promoted or released. However, because you cannot register a workflow to receive event notifications, the event must be explicitly queued to the workflow using an `IDfWorkflow.queue` method.

9.2.3 Port and package definitions

Ports are used to move packages in the workflow from one activity to the next. Packages contain the documents or other objects on which the work of the activity is performed. The definitions of both ports and packages are stored in properties in activity definitions.

9.2.3.1 Port definitions

Each port in an activity participates in one link. A port's type and the package definitions associated with the port define the packages the activity can receive or send through the link. The types of port include:

- Input

  An input port accepts a package as input for an activity. The package definitions associated with an input port define what packages the activity accepts. Each input port is connected through a link to an output port of a previous activity.

- Output

  An output port sends a package from an activity to the next activity. The package definitions associated with an output port define what packages the activity can pass to the next activity or activities. Each output port is connected by a link to an input port of a subsequent activity.

- Revert

  A revert port is a special input port that accepts packages sent back from a subsequent performer. A revert port is connected by a link to an output port of a subsequent activity.

- Exception

  An exception port is an output port that links an automatic activity to the input port of an Exception activity. Exception ports do not participate in transitions. The port is triggered only when the automatic activity fails.

9.2.3.2 Package definitions

Documents are moved through a workflow as packages moving from activity to activity through the ports. Packages are defined in properties of the activity definition.

Each port must have at least one associated package definition, and may have multiple package definitions. When an activity is completed and a transition to the next activity occurs, Documentum CM Server forwards to the next activity the package or packages defined for the activated output port.

If the package you define is an XML file, you can identify a schema to be associated with that file. If you later reference the package in an XPath expression in route case conditions of a manual activity for an automatic transition, the schema is used to validate the path. The XML file and the schema are associated using a relationship.

The actual packages represented by package definitions are generated at runtime by the server as needed and stored in the repository as `dmi_package` objects. You cannot create package objects directly.

9.2.3.2.1 Package compatibility

The package definitions associated with two ports connected by a link must be compatible.

The two ports referenced by a link must meet the following criteria to be considered compatible:

- They must have the same number of package definitions.

  For example, if `ActA_OP1` is linked to `ActB_IP2` and `ActA_OP1` has two package definitions, `ActB_IP2` must have two package definitions.

- The object types of the package components must be related as subtypes or supertypes in the object hierarchy. One of the following must be true:

  – The outgoing package type is a supertype of the incoming package type.

  – The outgoing package type is a subtype of the incoming package type.

  – The outgoing package type and the incoming package type are the same.

- For information about how the implementation actually moves packages from one activity to the next, see “Package acceptance” on page 176.

9.2.3.3 Package acceptance

When packages arrive at an input port, the server checks the port definition to see if the packages satisfy the port package requirements and verifies the number of packages and package types against the port definition.

If the port definitions are satisfied, the input port accepts the arriving packages by changing the `r_act_seqno`, `port_name`, and `package_name` properties of those packages.

The following illustration describes the process:

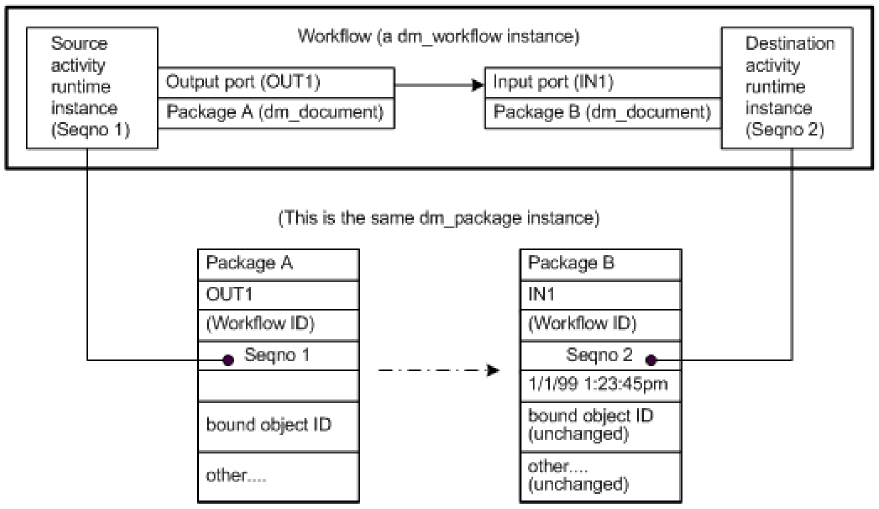


Figure 9-2: Package changes during port transition

In the figure, the output port named OUT1 of the source activity is linked to the input port named IN1 of the destination activity. OUT1 contains a package definition: Package A of type `dm_document`.

IN1 takes a similar package definition but with a different package name: Package B. When the package is delivered from the port OUT1 to the port IN1 during execution, the content of the package changes to reflect the transition:

- `r_package_name` changes from Package A to Package B
- `r_port_name` changes from OUT1 to IN1
- `r_activity_seq` changes from Seqno 1 to Seqno 2
- `i_acceptance_date` is set to the current time

In addition, at the destination activity, the server performs some bookkeeping tasks, including:

- Incrementing `r_trigger_revert` if the triggered port is a revert port

As soon as a revert port is triggered, the activity becomes active and no longer accepts any incoming packages (from input or other revert ports).

- Incrementing `r_trigger_input` if the triggered port is an input port

  As soon as this number matches the value of `trigger_threshold` in the activity definition, the activity stops accepting any incoming packages (from revert or other input ports) and starts its precondition evaluation.

- Setting `r_last_performer`

  This information comes directly from the previous activity.

Packages that are not needed to satisfy the trigger threshold are dropped. For example, in the following illustration, Activity C has two input ports: CI1, which accepts packages P1 and P2, and CI2, which accepts packages P1 and P3. Assume that the trigger threshold for Activity C is 1 - that is, only one of the two input ports must accept packages to start the activity.

Suppose Activity A completes and sends its packages to Activity C before Activity B and that the input port, CI1 accepts the packages. In that case, the packages arriving from Activity B are ignored.

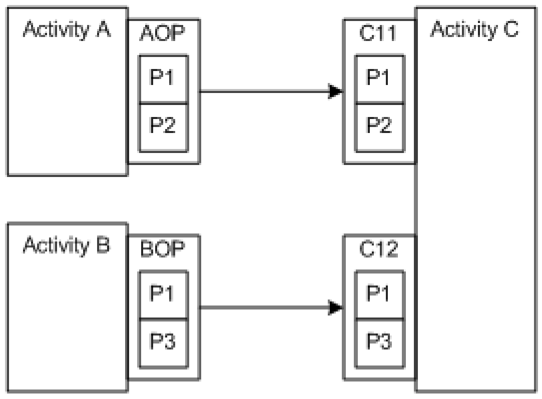


Figure 9-3: Workflow package arrival

9.2.4 Transition behavior

When an activity is completed, a transition to the next activity or activities occurs. The transition behavior defined for the activity defines when the output ports are activated and which output ports are activated. Transition behavior is determined by:

- The number of tasks that must be completed to trigger the transition

  By default, all generated tasks must be completed.

- The transition type

  If the number of completed tasks you specify is greater than the total number of work items for an activity, Documentum CM Server requires all work items for that activity to complete before triggering the transition. An activity transition type defines how the output ports are selected when the activity is complete. There are three types of transition:

- Prescribed

  If an activity transition type is prescribed, the server delivers packages to all the output ports. This is the default transition type.
- Manual

  If the activity transition type is manual, the activity performers must indicate at runtime which output ports receive packages.
- Automatic

  If the activity transition type is automatic, you must define one or more route cases for the transition.

9.2.5 Warning and suspend timers

Documentum CM Server supports the following timers for workflow activities:

- Warning timers

  The warning timers automate delivery of advisory messages to workflow supervisors and performers when an activity is not started within a given period or is not completed within a given period.

  Warning timers are defined when the activity is defined.

  There are two types of warning timer:

  - Pre-timers

    A pre-timer sends email messages if an activity is not started within a given time after the workflow starts.
  - Post-timers

    A post-timer sends messages when an activity is not completed within a specified interval, counting from the start of the activity.
- Suspend timers

  A suspend timer automates the resumption of a halted activity.

  Suspend timers are not part of an activity definition. They are defined by a method argument, at runtime, when an activity is halted with a suspension interval.

9.2.6 Package control

Package control is an optional feature. It is a specific constraint on Documentum CM Server that stops the server from recording package component object names specified in an `addPackage` or `addAttachment` method in the generated package or wf attachment object. By default, package control is not enabled. This means that if an `addPackage` or `addAttachment` method includes the component names as an argument, the names are recorded in the `r_component_name` property of the generated package or wf attachment object. If package control is enabled, Documentum CM Server sets the `r_component_name` property to a single blank even if the component names are specified in the methods.

If the control is enabled at the repository level, the setting in the individual workflow definitions is ignored. If the control is not enabled at the repository level, then you must decide whether to enable it for an individual workflow.

If you want to reference package component names in the task subject for any activities in the workflow, do not enable package control. Use package control only if you do not want to expose the object names of package components.

To enable package control in an individual workflow definition, set the `package_control` property to `1`.

For information about enabling or disabling package control at the repository level, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

9.3 Validation and installation

Activity and process definitions must be validated and installed before users can start a workflow based on the definitions.

To validate an activity or process definition requires either Relate permission on the process or activity definition or Sysadmin or Superuser privileges.

9.3.1 Validating process and activity definitions

Validating an activity definition verifies that:

- All package definitions are valid
- All objects referenced by the definition (such as a method object) are local
- The `transition_eval_cnt`, `transition_max_output_cnt`, and `transition_flag` properties have valid values.

Validating a process definition verifies that:

- The referenced activities have unique names within the process
- There is at least one Begin activity and only one End activity

- There is a path from each activity to the End activity
- All referenced `dm_activity` objects exist and are in the validated or installed state and that they are local objects
- All activities referenced by the link definitions exist
- The ports identified in the links are defined in the associated activity object
- There are no links that reference an input port of a Begin step and no links that reference an output port of an End step
- The ports are connectable and that each port participates in only one link

The validation verifies that both ports handle the same number of packages and the package definitions in the two ports are compatible.

The method checks all possible pairs of output/input package definitions in the two ports. If any pair of packages are incompatible, the connectivity test fails.

For more information about the rules for package compatibility, see “Package compatibility” on page 175.

9.3.2 Installing new process and activity definitions

Note: The information in this section applies to new process and activity definitions. If you are re-installing a modified workflow definition that has running instances, do not use the information in this section.

The process and activity definitions of a workflow definition must be installed before a workflow can be started from the definition.

A process or activity definition must be in the validated state before you install it.

You can install activity definitions individually, before you install the process definition, or concurrently with the process definition. You cannot install a process definition that contains uninstalled activities unless you install the activities concurrently. If you install only the process, the activities must be in the installed state.

Installing activity definitions and process definitions requires either:

- Relate permission on the process or activity definition
- Sysadmin or Superuser privileges

Refer to the associated *Javadocs* for information about the methods that install process and activity definitions.

9.4 Workflow execution

Workflow execution is implemented with the following object types:

- `dm_workflow`

  Workflow objects represent an instance of a workflow definition.

- `dmi_workitem`

  When an activity starts, the server creates one or more work items for the activity.

- `dmi_package`
- `dmi_queue_item`

  The server uses a queue item object to direct a work item to an inbox.

- `dmi_wf_timer`

9.4.1 Workflow objects

Workflow objects are created when the workflow is started by an application or a user. Workflow objects are subtypes of the persistent object type, and consequently, have no owner. However, every workflow has a designated supervisor (recorded in the `supervisor_name` property). This person functions similar to the owner of an object, with the ability to change the workflow properties and its state.

A workflow object contains properties that describe the activities in the workflow. These properties are set automatically, based on the workflow definition, when the workflow object is created. They are repeating properties, and the values at the same index position across the properties represent one activity instance.

The properties that make up the activity instance identify the activity, its current state, its warning timer deadlines (if any), and a variety of other information. As the workflow executes, the values in the activity instance properties change to reflect the status of the activities at any given time in the execution.

- For information, see “Workflow supervisor” on page 186.
- For information about the full list of the properties that make up an activity instance, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.

9.4.2 Work item and queue item objects

When an activity is started, the server creates one or more work items for the activity. A work item represents a task assigned to the activity performer (either a person or an invoked method).

Work items are instances of the `dmi_workitem` object type. A work item object contains properties that identify the activity that generated the work item and the user or method to perform the work, record the state of the work item, and record information for management.

The majority of the properties are set automatically, when the server creates the work item. A few are set at runtime. For example, if the activity performer executes a Repeat method to give the activity to a second round of performers, the work item `r_ext_performer` property is set.

Work item objects are not directly visible to users. To direct a work item to an inbox, the server uses a queue item object (`dmi_queue_item`). All work items for manual activities have peer queue item objects. Work items for automatic activities do not have peer queue item objects.

9.4.2.1 How manual activity work items are handled

The first operation that must occur on a work item is acquisition.

Users typically acquire a work item by selecting and opening the associated Inbox task. Internally, an acquire method is executed when a user acquires a work item. Acquiring a work item sets the work item state to acquired.

Users who have acquired a work item are called `performers`. The performer can perform the required work or delegate the work to another user if the activity definition allows delegation. The performer may also add or remove notes for the objects on which the work is performed. If the user performs the work, at its completion, the user can designate additional performers for the task if the activity definition allows extension.

All of these operations are supported internally using methods.

When a work item is finished, the performer indicates the completion through a client interface. Only a work item performer, the workflow supervisor, or a user with Sysadmin or Superuser privileges can complete a work item.

9.4.2.2 Priority values

Each work item inherits the priority value defined in the process definition for the activity that generated the work item. Documentum CM Server uses the inherited priority value of automatic activities, if set, to prioritize execution of the automatic activities. Documentum CM Server ignores priority values assigned to manual activities. A work item priority value can be changed at runtime.

Changing a work item priority generates an event that can be audited. Changing a priority value also changes the priority value recorded in any queue item object associated with the work item.

9.4.2.3 Signing off manual work items

Frequently, a business process requires the performers to sign off the work they do. Documentum CM Server supports three options to allow users to electronically sign off work items: electronic signatures, digital signatures, or simple sign-offs. You can customize work item completion to use any of these options.

- For information about the properties in the `dmi_workitem` and `dmi_queue_item` object types, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.
- For information about the options for signing off work items, see “Signature requirement support” on page 97.

9.4.3 Package objects

Packages contain the objects on which the work is performed. Packages are implemented as `dmi_package` objects. Package object properties:

- Identify the package and its contained objects
- Record the activity with which the package is associated
- Record when the package arrived at the activity
- Record information about any notes attached to the package

  At runtime, an activity performer can attach notes to packages, to pass information or instructions to the persons performing subsequent activities.
- Record whether the package is visible or invisible.

If a particular skill level is required to perform the task associated with the package, that information is stored in a `dmc_wf_package_skill` object. A wf package skill object identifies a skill level and a package. The objects are subtypes of `dm_relation` and are related to the workflow, with the workflow as the parent in the relationship. In this way, the information stays with the package for the life of the workflow.

A single instance of a package does not move from activity to activity. Instead, the server manufactures new copies of the package for each activity when the package is accepted and new copies when the package is sent on.

9.4.3.1 Package notes

Package notes are annotations that users can add to a package. Notes are used typically to provide instructions or information for a work item performer. A note can stay with a package as it moves through the workflow or it can be available only in the work items associated with one activity.

If an activity accepts multiple packages, Documentum CM Server merges any notes attached to the accepted packages.

If notes are attached to package accepted by a work item generated from an automatic activity, the notes are held and passed to the next performer of the next manual task.

Notes are stored in the repository as `dm_note` objects.

9.5 Activity timers

There are three types of timers for an activity. An activity can have a:

- Pre-timer that alerts the workflow supervisor if an activity has not started within a designated number of hours after the workflow starts
- Post-timer that alerts the workflow supervisor if an activity has not completed within a designated number of hours after the activity starts
- Suspend timer that automatically resumes the activity after a designated interval when the activity is halted

9.5.1 Pre-timer instantiation

When a workflow instance is created from a workflow definition, Documentum CM Server determines which activities in the workflow have pre-timers. For each activity with a pre-timer, it creates a `dmi_wf_timer` object. The object records the workflow object ID, information about the activity, the date and time at which to trigger the timer, and the action to take when the timer is triggered. The action is identified through a module config object ID. Module config objects point to business object modules stored in the Java method server.

If the activity is not started by the specified date and time, the timer is considered to be expired. Each execution of the `dm_WfmsTimer` job finds all expired timers and invokes the `dm_bpm_timer` method on each. Both the `dm_WfmsTimer` job and the `dm_bpm_method` are Java methods. The job passes the module config object ID to the method. The method uses the information in that object to determine the action. The `dm_bpm_method` method executes in the Java method server.

9.5.2 Post-timer instantiation

A post-timer is instantiated when the activity for which it is defined is started. When the activity is started, Documentum CM Server creates a `dmi_wf_timer` object for the post-timer. The timer records the workflow object ID, information about the activity, the date and time at which to trigger the timer, and the action to take when the timer is triggered.

9.5.3 Suspend timer instantiation

A suspend timer is instantiated when a user or application halts an activity with an explicit suspension interval. The interval is defined by an argument in the halt method. When the method is executed, Documentum CM Server creates a `dmi_wf_timer` object that identifies the workflow, the activity, and the date and time at which to resume the activity.

For more information about each kind of timer, see “Warning and suspend timers” on page 178.

9.6 Completed workflow reports

You can view reports about completed workflows using the Documentum Webtop Workflow Reporting tool. The data includes information such as when the workflow was started, how it finished (normally or aborted), when it was finished, and how long it ran. The report also provides similar information for the activities in the completed workflows.

The data is generated by the `dm_WFReporting` job, which invokes the `dm_WFReporting` method. The method examines audit trail entries for all workflow events for completed workflows. It collects the information from these events and generates objects of type `dmc_completed_workflow` and `dmc_completed_workitem`. Each object represents the data for one completed workflow or one completed work item in a completed workflow. The Documentum Webtop Workflow Reporting tool uses the information in the objects generated by the job to create its reports.

- *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)* contains information about:
  - Activating a job
  - Starting auditing
- The *OpenText Documentum Webtop* documentation contains information about accessing and using the Documentum Webtop Workflow Reporting tool.

9.7 Attachments

Attachments are objects that users attach to a running workflow or an uncompleted work item. Typically, the objects support the work required by the workflow activities. For example, if a workflow is handling an engineering proposal under development, a user might attach a research paper supporting that proposal. Attachments can be added at any point in a workflow and can be removed when they are no longer needed. After an attachment is added, it is available to the performers of all subsequent activities.

Attachments can be added by the workflow creator or supervisor, a work item performer, or a user with Sysadmin or Superuser privileges. Users cannot add a note to an attachment.

Internally, an attachment is saved in the repository as a `dmi_wf_attachment` object. The wf attachment object identifies the attached object and the workflow to which it is attached.

9.8 Workflow supervisor

Each workflow has a supervisor, who oversees execution of the entire workflow, receives any warning messages generated by the workflow, and resolves problems or obstacles encountered during execution. By default, the workflow supervisor is the person who creates the workflow. However, the workflow's creator can designate another user or a group as the workflow supervisor. In such cases, the creator has no special privileges for the workflow.

A normal workflow execution proceeds automatically, from activity to activity as each performer completes their work. However, the workflow's supervisor can affect the execution if needed. For example, the supervisor can change the workflow's state or an activity's state or manually delegate or extend an activity.

Users with Sysadmin or Superuser user privileges can act as the workflow supervisor. In addition, superusers are treated similar to the creator of a workflow and can change object properties, if necessary. However, messages that warn about execution problems are sent only to the workflow supervisor, not to superusers.

A workflow supervisor is recorded in the `supervisor_name` property of the workflow object.

9.9 Workflow agent

The workflow agent is the Documentum CM Server facility that controls the execution of automatic activities. The workflow agent is installed and started with Documentum CM Server. It maintains a master session and, by default, three worker sessions.

When Documentum CM Server creates an automatic activity, the server notifies the workflow agent. The master session is quiescent until it receives a notification from Documentum CM Server or until a specified sleep interval expires. When the master session receives a notification or the sleep interval expires, the master session wakes up. It executes a batch update query to claim a set of automatic activities for execution and then dispatches those activities to the execution queue. After all claimed activities are dispatched, the master session goes to sleep until either another notification arrives or the sleep interval expires again.

You can change the configuration of the workflow agent by changing the number of worker sessions and changing the default sleep interval. By default, there are three worker sessions and the sleep interval is 5 seconds. You can configure the agent with up to 1000 worker sessions. There is no maximum value on the sleep interval.

You can also trace the operations of the workflow agent or disable the agent. Disabling the workflow agent stops the execution of automatic activities.

For information about tracing or disabling the workflow agent, as well as changing the number of worker sessions and the sleep interval, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

9.10 Instance states

This section describes:

- workflow states

  A workflow current state is recorded in the `r_runtime_state` property of the `dm_workflow` object.
- activity states
- work item states

9.10.1 Workflow states

Every workflow instance exists in one of five possible states: dormant, running, finished, halted, or terminated. A workflow current state is recorded in the `r_runtime_state` property of the `dm_workflow` object.

The state transitions are driven by API methods or by the workflow termination criterion that determines whether a workflow is finished.

The following illustration describes the states:

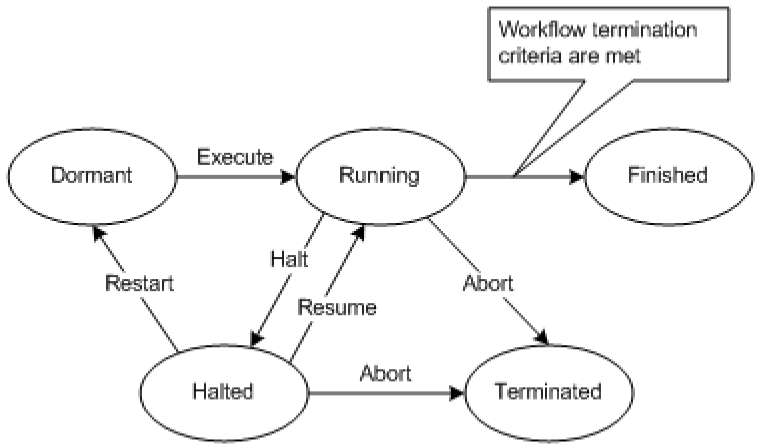


Figure 9-4: Workflow states

When a workflow supervisor first creates and saves a workflow object, the workflow is in the dormant state. When the Execute method is issued to start the workflow, the workflow state is changed to running.

Typically, a workflow spends its life in the running state, until either the server determines that the workflow is finished or the workflow supervisor manually terminates the workflow with the `IDfWorkflow.abort` method. If the workflow terminates normally, its state is set to finished. If the workflow is manually terminated with the abort method, its state is set to terminated.

A supervisor can halt a running workflow, which changes the workflow state to halted. From a halted state, the workflow supervisor can restart, resume, or abort the workflow.

9.10.2 Activity instance states

Every activity instance exists in one of five states: dormant, active, finished, failed, or halted. An activity instance state is recorded in the `r_act_state` property of the `dm_workflow` object, as part of the activity instance.

The following illustration describes the activity instance states and the operations or conditions that move the instance from one state to another:

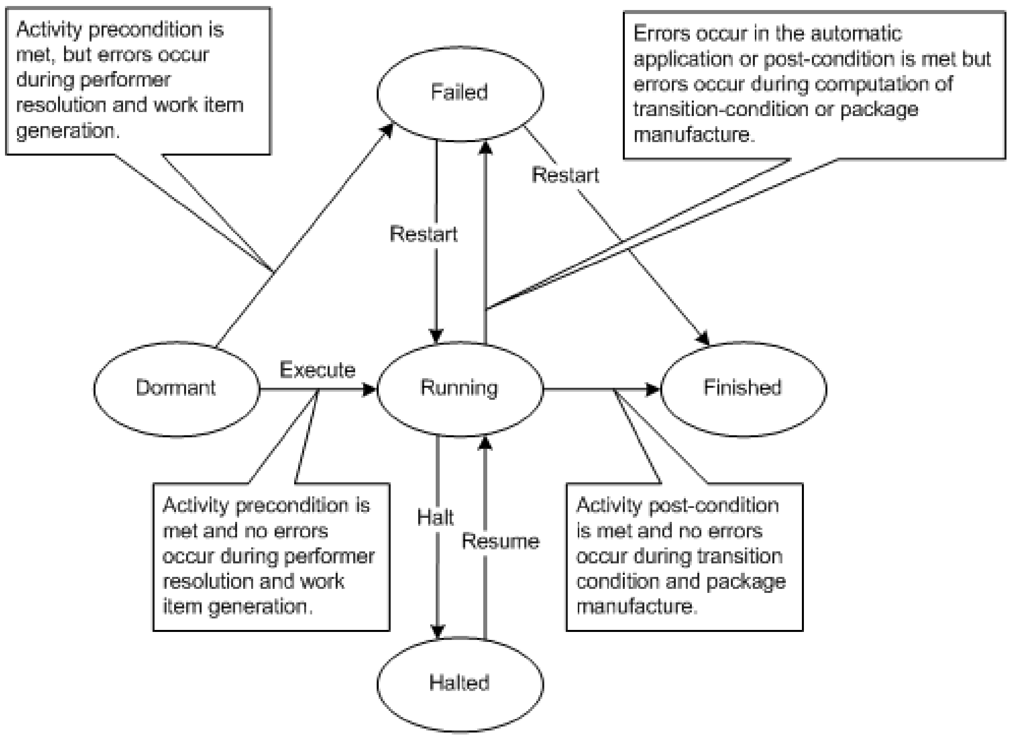

Activity precondition is met, but errors occur during performer resolution and work item generation.
Failed
Errors occur in the automatic application or post-condition is met but errors occur during computation of transition-condition or package manufacture.
Restart
Restart
Dormant
Execute
Running
Finished
Activity precondition is met and no errors occur during performer resolution and work item generation.
Halt
Resume
Activity post-condition is met and no errors occur during transition condition and package manufacture.
Halted

Figure 9-5: Activity instance states

During a typical workflow execution, an activity state is changed by the server to reflect the activity state within the executing workflow.

When an activity instance is created, the instance is in the dormant state. The server changes the activity instance to the active state after the activity starting condition is fulfilled and server begins to resolve the activity performers and generate work items.

If the server encounters any errors, it changes the activity instance state to failed and sends a warning message to the workflow supervisor.

The supervisor can fix the problem and restart a failed activity instance. An automatic activity instance that fails to execute can also change to the failed state, and the supervisor or the application owner can retry the activity instance.

The activity instance remains active while work items are being performed. The activity instance enters the finished state only when all its generated work items are completed.

A running activity can be halted. Halting an activity sets its state to halted. By default, only the workflow supervisor or a user with Sysadmin or Superuser privileges can halt or resume an activity instance.

Depending on how the activity was halted, it can be resumed manually or automatically. If a suspension interval is specified when the activity is halted, then the activity is automatically resumed after the interval expires. If a suspension interval is not specified, the activity must be manually resumed. Suspension intervals are set programmatically as an argument in the `IDfWorkflow.haltEx` method. Resuming an activity sets its state back to its previous state prior to being halted.

9.10.3 Work item states

A work item exists in one of the following states: dormant, paused, acquired, or finished.

The following illustration describes the work item states and the operations that move the work item from one state to another:

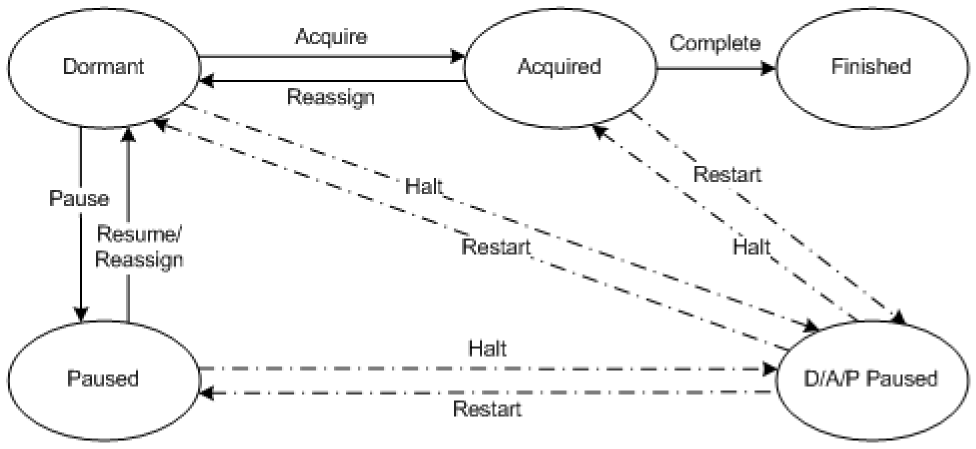


Figure 9-6: Work item states

A work item state is recorded in the `r_runtime_state` property of the `dmi_workitem` object.

When the server generates a work item for a manual activity, it sets the work item state to dormant and places the peer queue item in the performer inbox. The work item remains in the dormant state until the activity performer acquires it. Typically, acquisition happens when the performer opens the associated inbox item. At that time, the work item state is changed to acquired.

When the server generates a work item for an automatic activity, it sets the work item state to dormant and places the activity on the queue for execution. The application must issue the Acquire method to change the work item state to acquired.

After the activity work is finished, the performer or the application must execute the Complete method to mark the work item as complete. This changes the work item's state to finished.

A work item can be moved manually to the paused state by the activity performer, the workflow supervisor, or a user with Sysadmin or Superuser privileges. A paused work item requires a manual state change to return to the dormant or acquired state.

For more information about how suspension intervals are implemented, see "Activity timers" on page 184.

9.11 Typical workflow example

Users typically start a workflow through one of the client interfaces. If you are starting a workflow programmatically, there are two steps. First, a workflow object must be created and saved. Then, an execute method must be issued for the workflow object.

Saving the new workflow object requires Relate permission on the process object (the workflow definition) used as the workflow template. The execute method must be issued by the workflow creator or supervisor or a user with Sysadmin or Superuser privileges. If the user is starting the workflow through a client interface, such as Documentum Webtop, the user must also be defined as a `Contributor`.

This section describes how a typical workflow executes. It describes what happens when a workflow is started and how execution proceeds from activity to activity. It also describes how packages are handled and how a warning timer behaves during workflow execution.

The following illustration describes the general execution flow:

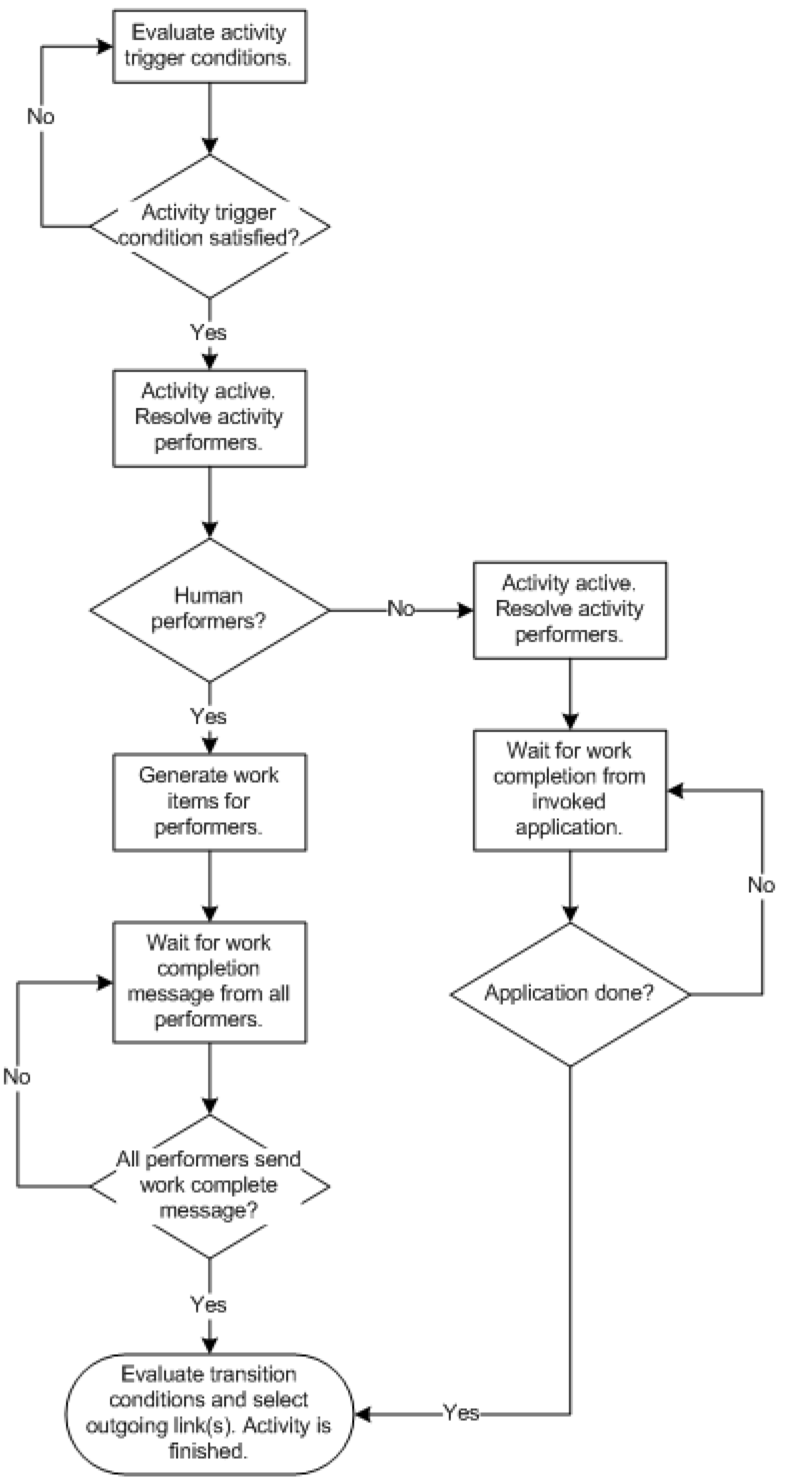

Evaluate activity trigger conditions.
No
Activity trigger condition satisfied?
Yes
Activity active. Resolve activity performers.
Human performers?
No
Activity active. Resolve activity performers.
Yes
Generate work items for performers.
Wait for work completion from invoked application.
No
Wait for work completion message from all performers.
Application done?
No
All performers send work complete message?
Yes
Evaluate transition conditions and select outgoing link(s). Activity is finished.
Yes

Figure 9-7: Workflow execution flow

9.11.1 Workflow starts

A workflow starts when a user issues the execute method against a `dm_workflow` object. The execute method does the following activities:

- Sets the `r_pre_timer` property for those activity instances that have pre-timers defined
- Examines the starting condition of each Begin activity and, if the starting condition is met:
  - Sets the `r_post_timer` property for the activity instance if a post timer is defined for the activity
  - Resolves performers for the activity
  - Generates the activity's work items
  - Sets the activity's state to active
- Records the workflow's start time

After the execute method returns successfully, the workflow's execution has begun, starting with the `Begin` activities.

9.11.2 Activity execution starts

For `Begin` activities, execution begins when an execute method is executed for the workflow. The starting condition of a typical Begin activity with no input ports is always considered fulfilled. If a `Begin` activity has input ports, the application or user must use an `addPackage` method to pass the required packages to the activity through the workflow. When the package is accepted, the server evaluates the activity starting condition just as it does for `Step` and `End` activities.

For `Step` and `End` activities, execution begins when a package arrives at one of the activity input ports. If the package is accepted, it triggers the server to evaluate the activity starting condition.

Note: For all activities, if the port receiving the package is a revert port and the package is accepted, the activity stops accepting further packages, and the server ignores the starting condition and immediately begins resolving the activity performers.

After the server determines that an activity starting condition is satisfied, it consolidates packages if necessary. Next, the server determines who will perform the work and generates the required work items. If the activity is an automatic activity, the server queues the activity for starting.

9.11.2.1 Evaluating the starting condition

An activity starting condition defines the number of ports that must accept packages and, optionally, an event that must be queued in order to start the activity. The starting condition is defined in the `trigger_threshold` and `trigger_event` properties in the activity definition. When a workflow is created, these values are copied to the `r_trigger_threshold` and `r_trigger_event` properties in the workflow object.

When an activity input port accepts a package, the server increments the activity instance `r_trigger_input` property in the workflow object and then compares the value in `r_trigger_input` to the value in `r_trigger_threshold`.

If the two values are equal and no trigger event is required, the server considers that the activity has satisfied its starting condition. If a trigger event is required, the server will query the `dmi_queue_item` objects to determine whether the event identified in `r_trigger_event` is queued. If the event is in the queue, then the starting condition is satisfied.

If the two values are not equal, the server considers that the starting condition is not satisfied.

The server also evaluates the starting condition each time an event is queued to the workflow.

After a starting condition that includes an event is satisfied, the server removes the event from the queue. If multiple activities use the same event as part of their starting conditions, the event must be queued for each activity.

When the starting condition is satisfied, the server consolidates the accepted packages if necessary and then resolves the performers and generates the work items. If it is a manual activity, the server places the work item in the performer inbox. If it is an automatic activity, the server passes the performer name to the application invoked for the activity.

9.11.2.2 Package consolidation

If activity input ports have accepted multiple packages with the same `r_package_type` value, the server consolidates those packages into one package.

For example, suppose that Activity C accepts four packages: two `Package_typeA`, one `Package_typeB`, and one `Package_typeC`. Before generating the work items, the server will consolidate the two `Package_typeA` package objects into one package, represented by one package object. It does this by merging the components and any notes attached to the components.

The consolidation order is based on the acceptance time of each package instance, as recorded in the `i_acceptance_date` property of the package objects.

9.11.2.3 Resolving performers and generating work items

After the starting condition is met and packages consolidated if necessary, the server determines the performers for the activity and generates the work items.

For manual activities, the server uses the value in the `performer_type` property in conjunction with the `performer_name` property, if needed, to determine the activity performer. After the performer is determined, the server generates the necessary work items and peer queue items.

If the server cannot assign the work item to the selected performer because the performer has `workflow_disabled` set to `TRUE` in his or her user object, the server attempts to delegate the work item to the user listed in the `user_delegation` property of the performer user object.

If automatic delegation fails, the server reassigns the work item based on the setting of the `control_flag` property in the definition of the activity that generated the work item.

Note: When a work item is generated for all members of a group, users in the group who are workflow disabled do not receive the work item, nor is the item assigned to their delegated users.

If the server cannot determine a performer, a warning is sent to the performer who completed the previous work item and the current work item is assigned to the supervisor.

For automatic activities, the server uses the value in the `performer_type` property in conjunction with the `performer_name` property, if needed, to determine the activity performer. The server passes the name of the selected performer to the invoked program.

The server generates work items but not peer queue items for work items representing automatic activities.

When the `performer_name` property contains an alias, the server resolves the alias using a resolution algorithm determined by the value found in the activity's `resolve_type` property.

If the server cannot determine a performer, a warning is sent to the workflow supervisor and the current work item is assigned to the supervisor.

- For information about how automatic activities are executed, see "Executing automatic activities" on page 196.
- For information about the resolution algorithms for performer aliases, see "Resolving aliases in workflows" on page 234.

9.11.2.3.1 Executing automatic activities

The master session of the workflow agent controls the execution of automatic activities. The workflow agent is an internal server facility.

9.11.2.3.2 Assigning an activity for execution

After the server determines the activity performer and creates the work item, the server notifies the workflow agent master session that an automatic activity is ready for execution. The master session handles activities in batches. If the master session is not currently processing a batch when the notification arrives, the session wakes up and does the following:

1. Executes an update query to claim a batch of work items generated by automatic activities.

   A workflow agent master session claims a batch of work items by setting the `a_wq_name` property of the work items to the name of the server config object representing the Documentum CM Server. The maximum number of work items in a batch is the lesser of 2000 or 30 times the number of worker threads.

2. Selects the claimed work items and dispatches the returned items to the execution queue.

   The work items are dispatched one item at a time. If the queue is full, the master session checks the size of the queue (the number of items in the queue). If the size is greater than a set threshold, it waits until it receives notification from a worker thread that the queue has been reduced. A worker thread checks the size of the queue each time it acquires a work item. When the size of the queue equals the threshold, the thread sends the notification to the master session. The notification from the worker thread tells the master session it can resume putting work items on the queue.

   The queue can have a maximum of 2000 work items. The threshold is equal to fives times the number of worker threads.

3. After all claimed work items are dispatched, the master agent returns to sleep until another notification arrives from Documentum CM Server or the sleep interval passes.

Note: If the Documentum CM Server associated with the workflow agent should fail while there are work items claimed but not processed, when the server is restarted, the workflow agent will pick up the processing where it left off. If the server cannot be restarted, you can use an administration method to recover those work items for processing by another workflow agent.

9.11.2.3.3 Executing an activity program

When a workflow agent worker session takes an activity from the execution queue, it retrieves the activity object from the repository and locks it. It also fetches some related objects, such as the workflow. If any of the objects cannot be fetched or if the fetched workflow is not running, the worker session sets `a_wq_name` to a message string that specifies the problem and drops the task without processing it. Setting `a_wq_name` also ensures that the task will not be picked up again.

After all the fetches succeed and after verifying the ready state of the activity, the worker thread executes the method associated with the activity. The method is always executed as the server regardless of the `run_as_server` property setting in the method object.

Note: If the activity is already locked, the worker session assumes that another workflow agent is executing the activity. The worker session simply skips the activity and no error message is logged. This situation can occur in repositories with multiple servers, each having its own workflow agent.

If an activity fails for any reason, the selected performer receives a notification.

The server passes the following information to the invoked program:

- Repository name
- User name (this is the selected performer)
- Login ticket
- Work item object ID
- Mode value

The information is passed in the following format:

```
-docbase_name repository_name -user user_name -ticket login_ticket-packageId workitem_id
mode mode_value
```

The mode value is set automatically by the server. The following table lists the values for the mode parameter:

Value	Meaning
0	Normal
1	Restart (previous execution failed)
2	Termination situation (re-execute because workflow terminated before automatic activity user program completed)

The method program can use the login ticket to connect back to the repository as the selected performer. The work item object ID allows the program to query the repository for information about the package associated with the activity and other information it may need to perform its work.

- For more information, see "Workflow agent" on page 187.
- For more information about the instructions for recovering work items for execution by an alternate workflow agent in case of a Documentum CM Server failure, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

9.11.3 Completing an activity

When a performer completes a work item, the server increments the `r_complete_witem` property in the workflow object and then evaluates whether the activity is complete. To do so, the server compares the value of the `r_complete_witem` property to the value in the workflow `r_total_workitem` property. The `r_total_witem` property records the total number of work items generated for the activity. The `r_complete_witem` property records how many of the activity work items are completed.

If the two values are the same and extension is not enabled for the activity, the server considers that the activity is completed. If extension is enabled, the server:

- Collects the second-round performers from the `r_ext_performer` property of all generated work items
- Generates another set of work items for the user or users designated as the second-round performers and removes the first round of work items
- Sets the `i_performer_flag` to indicate that the activity is in the extended mode and no more extension is allowed

The following illustration describes the decision process when the properties are equal:

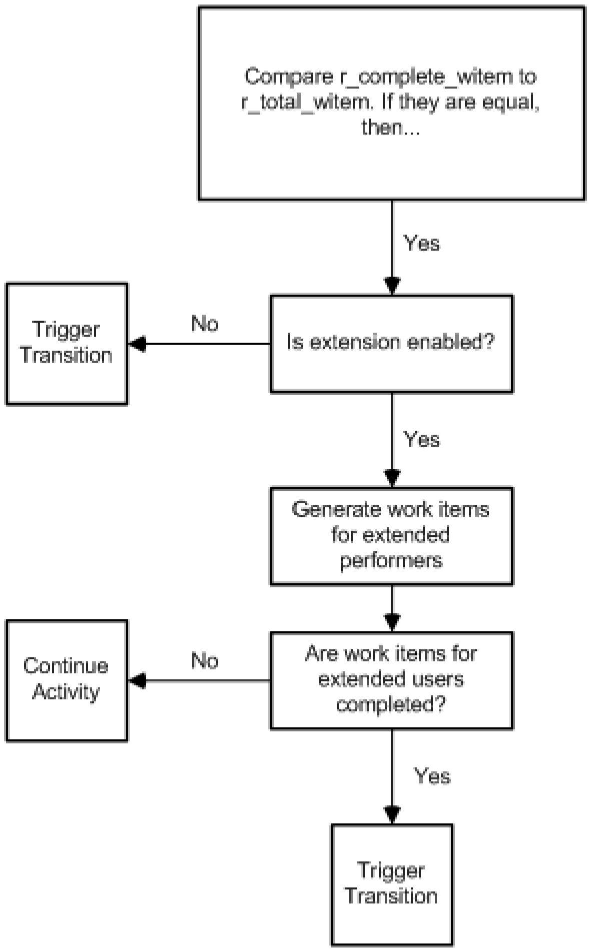

Compare r_complete_witem to r_total_witem. If they are equal, then...
Yes
Trigger Transition
No
Is extension enabled?
Yes
Generate work items for extended performers
Continue Activity
No
Are work items for extended users completed?
Yes
Trigger Transition

Figure 9-8: Completion equals behavior

If the number of completed work items is lower than the total number of work items, the server then uses the values in `transition_eval_cnt` and, for activities with a manual transition, the `transition_flag` property to determine whether to trigger a transition. The `transition_eval_cnt` property specifies how many work items must be completed to finish the activity. The `transition_flag` property defines how ports are chosen for the transition. The following illustration describes the decision process when `r_complete_witem` and `r_total_workitem` are not equal:

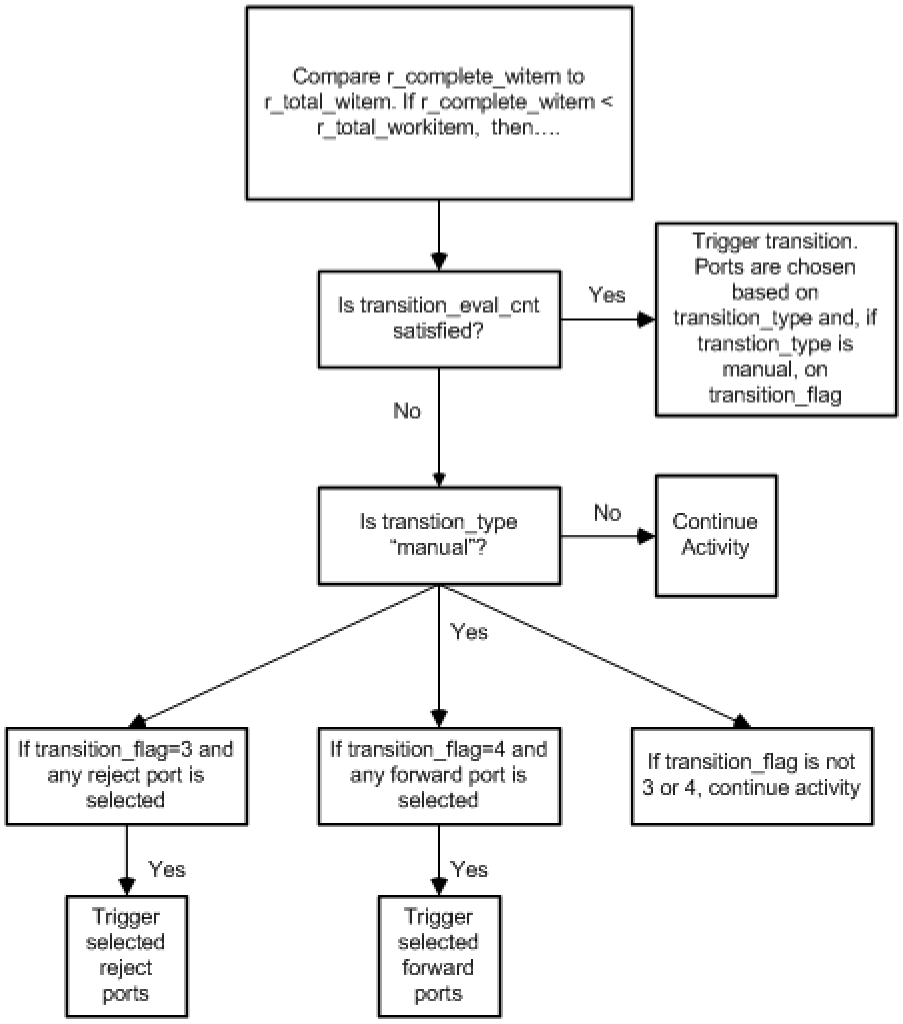

Compare r_complete_witem to r_total_witem. If r_complete_witem < r_total_workitem, then….
Is transition_eval_cnt satisfied?
Yes
Trigger transition. Ports are chosen based on transition_type and, if transtion_type is manual, on transition_flag
No
Is transtion_type "manual"?
No
Continue Activity
Yes
If transition_flag=3 and any reject port is selected
If transition_flag=4 and any forward port is selected
If transition_flag is not 3 or 4, continue activity
Yes
Trigger selected reject ports
Yes
Trigger selected forward ports

Figure 9-9: Completion less than behavior

If an activity transition is triggered before all the activity work items are completed, Documentum CM Server marks the unfinished work items as pseudo-complete and removes them from the inboxes of the performers. The server also sends an email message to the performers to notify them that the work items have been removed.

Note: Marking an unfinished work item as pseudo-complete is an auditable event. The event name is `dm_pseudocompleteworkitem`.

Additionally, if an activity transition is triggered before all work items are completed, any extended work items are not generated even if extension is enabled.

After an activity is completed, the server selects the output ports based on the transition type defined for the activity.

If the transition type is prescribed, the server delivers packages to all the output ports.

If the transition type is manual, the user or application must designate the output ports. The choices are passed to Documentum CM Server using one of the `Setoutput` methods. The number of choices may be limited by the activity's definition. For example, the activity definition may only allow a performer to choose two output ports. How the selected ports are used is also specified in the activity's definition. For example, if multiple ports are selected, the definition may require the server to send packages to the selected revert ports and ignore the forward selections.

If the transition type is automatic, the route cases are evaluated to determine which ports will receive packages. If the activity's `r_condition_id` property is set, the server evaluates the route cases. If the activity's `r_predicate_id` property is set, the server invokes the `dm_bpm_transition` method to evaluate the route cases. The `dm_bpm_transition` method is a Java method that executes in the Java method server. The server selects the ports associated with the first route case that returns a `TRUE` value.

After the ports are determined, the server creates the needed package objects. If the package creation is successful, the server considers that the activity is finished. At this point, the cycle begins again with the start of the next activity's execution.

9.12 Distributed workflow

A distributed workflow consists of distributed notification and object routing capability. Any object can be bound to a workflow package and passed from one activity to another.

Distributed workflow works best in a federated environment where users, groups, object types, and ACLs are known to all participating repositories.

In such an environment, users in all repositories can participate in a business process. All users are known to every repository, and the workflow designer treats remote users no differently than local users. Each user designates a home repository and receives notification of all work item assignments in the home inbox.

All process and activity definitions and workflow runtime objects must reside in a single repository. A process cannot refer to an activity definition that resides in a different repository. A user cannot execute a process that resides in a repository different from the repository where the user is currently connected.

9.12.1 Distributed notification

When a work item is assigned to a remote user, a work item and the peer queue item are generated in the repository where the process definition and the containing workflow reside. The notification agent for the source repository replicates the queue item in the user home repository. Using these queue items, the home inbox connects to the source repository and retrieves all information necessary for the user to perform the work item tasks.

A remote user must be able to connect to the source repository to work on a replicated queue item.

The process is:

1. A work item is generated and assigned to user A (a remote user). A peer queue item is also generated and placed in the queue. Meanwhile, a mail message is sent to user A.
2. The notification agent replicates the queue item in user A home repository.
3. User A connects to the home repository and acquires the queue item. The user home inbox makes a connection to the source repository and fetches the peer work item. The home inbox executes the Acquire method for the work item.
4. User A opens the work item to find out about arriving packages. The user home inbox executes a query that returns a list of package IDs. The inbox then fetches all package objects and displays the package information.
5. When user A opens a package and wants to see the attached instructions, the user home inbox fetches the attached notes and contents from the source repository and displays the instructions.
6. User A starts working on the document bound to the package. The user home inbox retrieves and checks out the document and contents from the source repository. The inbox decides whether to create a reference that refers to the bound document.
7. When user A is done with the package and wants to attach an instruction for subsequent activity performers, the user home inbox creates a note object in the source repository and executes the `addNote` method to attach notes to the package. The inbox then executes the Complete method for the work item and cleans up objects that are no longer needed.

9.13 Tasks and events

Tasks and events are occurrences within an application or repository that are of interest to users.

Tasks are items sent to a user that require the user to perform some action. Tasks are usually assigned to a user as a result of a workflow. When a workflow activity starts, Documentum CM Server determines who is performing the activity and assigns that user the task. It is also possible to send tasks to users manually.

Events are specific actions on specific documents, folders, cabinets, or other objects. For example, a checkin on a particular document is an event. Promoting or demoting a document in a lifecycle is an event. Documentum CM Server supports a large number of system-defined events, representing operations such as checkins, promotions, and demotions.

Events can also be defined by an application. If an application defines an event, the application is responsible for triggering the email notification. For example, an application might want to notify a particular department head if some application-specific event occurs. When the event occurs, the application issues a queue method to send a notification to the department head. In the method, the application can set an argument that directs Documentum CM Server to send a message with the event notification.

Tasks and event notifications are stored in the repository as `dmi_queue_item` objects. Tasks generated by workflows also have a `dmi_workitem` object in the repository.

9.13.1 Accessing tasks and events

Typically, users access tasks and event notifications through their repository inboxes.

Tasks are sent to the inbox automatically, when the task is generated. Users must register to receive events. Users can register to receive notifications of system-defined events. When a system-defined event occurs, Documentum CM Server sends an event notification automatically to any user who is registered to receive the event.

Users cannot register for application-defined events. Generating application-defined events and triggering notifications of the events are managed completely by the application.

- For information about the tables listing all system-defined events, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.
- For information, see “Work item and queue item objects” on page 182.
- For information, see “Inboxes” on page 204.

9.14 Inboxes

Electronic inbox holds various items that require your attention.

An inbox is a virtual container that holds tasks, event notifications, and other items sent to users manually (using a queue method). For example, one of your employees might place a vacation request in your inbox, or a coworker might ask you to review a presentation. Each user in a repository has an inbox.

9.14.1 Accessing an Inbox

Users access their inboxes through the client applications. If your enterprise has defined a home repository for users, the inboxes are accessed through the home repository. All inbox items, regardless of the repository in which they are generated, appear in the home repository inbox. Users must login to the home repository to view their inbox.

If you do not define home repositories for users, Documentum CM Server maintains an inbox for each repository. Users must log in to each repository to view the inbox for that repository. The inbox contains only those items generated within the repository.

Applications access inbox items by querying and referencing `dmi_queue_item` objects.

All items that appear in an inbox are managed by the server as objects of type `dmi_queue_item`. The properties of a queue item object contain information about the queued item. For example, the `sent_by` property contains the name of the user who sent the item and the `date_sent` property tells when it was sent.

The `dmi_queue_item` objects are persistent. They remain in the repository even after the items they represent have been removed from an inbox, providing a persistent record of completed tasks. Two properties that are set when an item is removed from an inbox contain the history of a project with which tasks are associated. These properties are:

- `dequeued_by` contains the name of the user that removed the item from the inbox.
- `dequeued_date` contains the date and time that the item was removed.

For information about the `dmi_queue_item` object type, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.

9.15 Obtaining inbox content

You can obtain the content of a particular inbox programmatically using the following ways:

- `GET_INBOX` administration method

  `GET_INBOX` returns a collection containing the inbox items in query result objects. Using `GET_INBOX` is the simplest way to retrieve all items in a user's inbox.

- `getEvents` method

  An `IDfSession.getEvents` method returns all new (unread) items in the current user's queue. Unread items are all queue item objects placed on the queue after the last `getEvents` execution against that queue.

  The queue item objects are returned as a collection. Use the collection identifier to process the returned items.

- The `dm_queue` view

  The `dm_queue` view is a view on the `dmi_queue_item` object type. To obtain information about a queue using DQL, query against this view. Querying against this view is the simplest way to view all the contents of a queue. For example, the following DQL statement retrieves all the items in Haskell's inbox. For each item, the statement retrieves the name of the queued item, when it was sent, and its priority.

  ```
  SELECT "item_name","date_sent","priority" FROM "dm_queue"
  WHERE "name" = 'Haskell'
  ```

To determine whether to refresh an inbox, you can use an `IDfSession.hasEvents` method to check for new items. A new item is defined as any item queued to the inbox after the previous execution of `getEvents` for the user. The method returns `TRUE` if there are new items in the inbox or `FALSE` if there are no new items.

- For information about the instructions on using `GET_INBOX`, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.
- For information about the reference information about the properties of a queue item object, see *OpenText Documentum Content Management - Server System Object Reference Guide (EDCCS250400-ORD)*.

9.16 Manual queuing and dequeuing

Most inbox items are generated automatically by workflows or an event registration. However, you can manually or programmatically queue a SysObject or a workflow-related event notification using a queue method. You can also manually or programmatically take an item out of an inbox by dequeuing the item.

9.16.1 Queuing items

Use a queue method to place an item in an inbox. Executing a queue method creates a queue item object. You can queue a SysObject or a user- or application-defined event.

When you queue an object, including an event name is optional. You may want to include one, however, to be manipulated by the application. Documentum CM Server ignores the event name.

When you queue a workflow-related event, the event value is not optional. The value you assign to the parameter should match the value in the `trigger_event` property for one of the workflow's activities.

Although you must assign a priority value to queued items and events, your application can ignore the value or use it. For example, the application might read the priorities and present the items to the user in priority order. The priority is ignored by Documentum CM Server.

You can also include a message to the user receiving the item.

9.16.2 Dequeuing an inbox item

Use an `IDfSession.dequeue` method to remove an item placed in an inbox using a queue method. Executing a dequeue method sets two queue item properties:

- `dequeued_by`

  This property contains the name of the user who dequeued the item.

- `dequeued_date`

  This property contains the date and time that the item was dequeued.

9.17 Registering and unregistering for event notifications

An event notification is a notice from Documentum CM Server that a particular system event has occurred. To receive an event notification for a system event, you must register for the event.

The event can be a specific action on a particular object or a specific action on objects of a particular type. You can also register to receive notification for all actions on a particular object.

For instance, you might want to know whenever a particular document is checked out. Or you might want to know when any document is checked out. You might want to know when any action (checkin, checkout, promotion, and so on) happens to a particular document. Each of these actions is an event, and you can register to receive notification when the event occurs. After you have registered for an event, the server continues to notify you when the event occurs until you remove the registration.

9.17.1 Registering for events

You can register to receive events using Documentum Administrator. You can also use an `IDfSysObject.registerEvent` method.

Although you must assign a priority value to an event when you use the `registerEvent` method, your application can ignore the value or use it. This argument is provided as an easy way for your application to manipulate the event when the event appears in your inbox. For example, the application might sort out events that have a higher priority and present them first. The priority is ignored by Documentum CM Server.

You cannot register another user for an event. Executing a `registerEvent` method registers the current user for the specified event.

9.17.2 Removing a registration

To remove an event registration, use Documentum Administrator or an `IDfSysObject.unRegister` method.

Only a user with Sysadmin or Superuser privileges can remove another user registration for an event notification.

If you have more than one event defined for an object, the `unRegister` method only removes the registration that corresponds to the combination of the object and the event. Other event registrations for that object remain in place.

For more information about the system events for which you may register for notification, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD).*

9.17.3 Querying for registration information

Registrations are stored in the repository as `dmi_registry` objects. You can query this type to obtain information about the current registrations. For example, the following query returns the registrations for a particular user:

```
SELECT * FROM "dmi_registry"
WHERE "user_name" = 'user'
```

Chapter 10

Lifecycles

10.1 Overview

A lifecycle is one of the process management services provided with Documentum CM Server. Lifecycles automate management of documents throughout their `lives` in the repository.

A lifecycle is a set of states that define the stages in the life of an object. The states are connected linearly. An object attached to a lifecycle progresses through the states as it moves through its lifetime. A change from one state to another is governed by business rules. The rules are implemented as requirements that the object must meet to enter a state and actions to be performed on entering a state. Each state can also have actions to be performed after entering a state.

Lifecycles contain:

- States

  A lifecycle can be in one of a normal progression of states or in an exception state.

- Attached objects

  Any system object or subtype (except a lifecycle object itself) can have an attached lifecycle.

- Entry and post entry actions

  A lifecycle can trigger custom behavior in the repository when an object enters or leaves a lifecycle state.

You use the Lifecycle Editor, accessed through Documentum Composer, to create a lifecycle. Design states in the lifecycle can then attach an object (for example, a document) to the lifecycle. Entry criteria apply to each state defined in the lifecycle.

For example, a lifecycle for a Standard Operating Procedure (SOP) might have states representing the draft, review, rewrite, approved, and obsolete states of an SOP life. Before an SOP can move from the rewrite state to the approved state, business rules might require the SOP to be signed off by a company vice president, and converted to HTML format for publishing on a company web site. After the SOP enters the approved state, an action can send an email message to employees informing them the SOP is available.

10.1.1 Normal and exception states

There are two kinds of state: normal and exception. Normal states are the states that define the typical stages of object life. Exception states represent situations outside of the normal stages of object life. All lifecycles must have normal states. Exception states are optional. Each normal state in a lifecycle definition can have one exception state.

If an exception state is defined for a normal state, when an object is in that normal state, you can suspend the object progress through the lifecycle by moving the object to the exception state. Later, you can resume the lifecycle for the object by moving the object out of the exception state back to the normal state or returning it to the base state.

For example, if a document describes a legal process, you can create an exception state to temporarily halt the lifecycle if the laws change. The document lifecycle cannot resume until the document is updated to reflect the changes in the law.

The following figure illustrates an example of a lifecycle with exception states. Similar to normal states, exception states have their own requirements and actions.

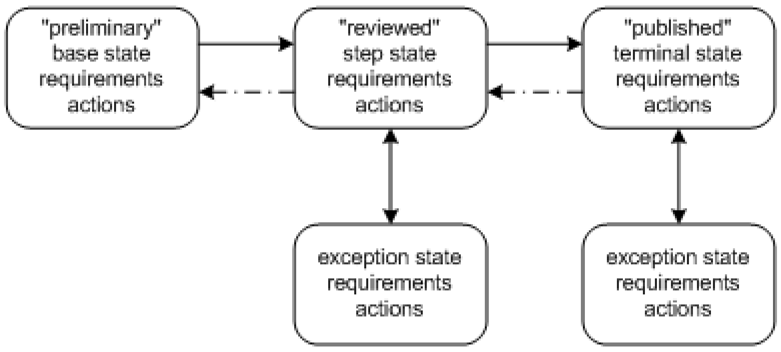


Figure 10-1: Lifecycle definition with exception states

Which normal and exception states you include in a lifecycle depends on which object types will be attached to the lifecycle. The states reflect the stages of life for those particular objects. When you are designing a lifecycle, after you have determined which objects you want the lifecycle to handle, decide what the life states are for those objects. Then, decide whether any or all of those states require an exception state.

10.1.2 Attaching an object to a lifecycle

After a lifecycle is validated and installed, users may begin attaching objects to the lifecycle. Because the states are states of being, not tasks, attaching an object to a lifecycle does not generate any runtime objects.

When an object is attached to a lifecycle state, Documentum CM Server evaluates the entry criteria for the state. If the criteria are met, the attach operation succeeds. The server then:

- Stores the object ID of the lifecycle definition in the object `r_policy_id` property
- Sets the `r_alias_set_id` to the object ID of the alias set associated with the lifecycle, if any
- Executes any actions defined for the state
- Sets the `r_current_state` property to the number of the state

From this point, the object continues through the lifecycle. If the object was attached to a normal state, it can move to the next normal state, to the previous normal state, or to the exception state defined for the normal state. If the object was attached to an exception state, it can move to the normal state associated with the exception state or to the base state.

Each time the object is moved forward to a normal state or to an exception state, Documentum CM Server evaluates the entry criteria for the target state. If the object satisfies the criteria, the server performs the entry actions, and resets the `r_current_state` property to the number of the target state. If the target state is an exception state, Documentum CM Server also sets `r_resume_state` to identify the normal state to which the object can be returned. After changing the state, the server performs any post-entry actions defined for the target state. The actions can make fundamental changes (such as changes in ownership, access control, location, or properties) to an object as that object progresses through the lifecycle.

If an object is demoted back to the previous normal state, Documentum CM Server only performs the actions associated with the state and resets the properties. It does not evaluate the entry criteria.

Objects cannot skip normal steps as they progress through a lifecycle.

The following illustration describes an example of a simple lifecycle with three states: preliminary, reviewed, and published. Each state has its own requirements and actions. The preliminary state is the base state.

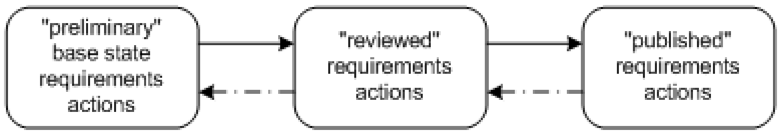


Figure 10-2: Simple lifecycle definition

10.1.2.1 Attaching objects

An object may be attached to any attachable state. By default, unless another state is explicitly identified when an object is attached to a lifecycle, Documentum CM Server attaches the object to the first attachable state in the lifecycle. Typically, this is the base state.

A state is attachable if the `allow_attach` property is set for the state.

When an object is attached to a state, Documentum CM Server tests the entry criteria and performs the actions on entry. If the entry criteria are not satisfied or the actions fail, the object is not attached to the state.

Programmatically, attaching an object is accomplished using an `IDfSysObject.attachPolicy` method.

10.1.2.2 Moving between states

Objects move between states in a lifecycle through promotions, demotions, suspensions, and resumptions. Promotions and demotions move objects through the normal states. Suspensions and resumptions are used to move objects into and out of the exception states.

10.1.2.2.1 Promotions

Promotion moves an object from one normal state to the next normal state. Users who own an object or are superusers need only Write permission to promote the object. Other users must have Write permission and Change State permission to promote an object. If the user has only Change State permission, Documentum CM Server will attempt to promote the object as the user defined in the `a_bpaction_run_as` property in the docbase config object. In those instances, that user must be either the owner or a superuser with Write permission or have Write and Change State permission on the object.

A promotion only succeeds if the object satisfies any entry criteria and actions on entry defined for the target state.

It is possible to bypass the entry criteria. If you choose to do that, the server does not enforce the entry criteria, but simply performs the actions associated with the destination state and, on their completion, moves the object to the destination state. You must own the lifecycle policy object or be a superuser to bypass entry criteria.

Promotions are accomplished programmatically using one of the promote methods in the `IDfSysObject` interface. Bypassing the entry criteria is accomplished by setting the override argument in the method to true.

Batch promotion is the promotion of multiple objects in batches. Documentum CM Server supports batch promotions using the `BATCH_PROMOTE` administration method. You can use it to promote multiple objects in one operation.

10.1.2.2.2 Demotions

Demotion moves an object from a normal state back to the previous normal state or back to the base state. Demotions are only supported by states that are defined as allowing demotions. The value of the `allow_demote` property for the state must be `TRUE`. Additionally, to demote an object back to the base state, the `return_to_base` property value must be `TRUE` for the current state.

Users who own an object or are superusers need only Write permission to demote the object. Other users must have Write permission and Change State permission to demote an object. If the user has only Change State permission, Documentum CM Server will attempt to demote the object as the user defined in the `a_bpaction_run_as` property in the docbase config object. In those instances, that user must be either the owner or a superuser with Write permission or have Write and Change State permission on the object.

If the object current state is a normal state, the object can be demoted to either the previous normal state or the base state. If the object current state is an exception state, the object can be demoted only to the base state. Demotions are accomplished programmatically using one of the demote methods in the `IDfSysObject` interface.

10.1.2.2.3 Suspensions

Suspension moves an object from the current normal state to the state exception state. Users who own an object or are superusers need only Write permission to suspend the object. Other users must have Write permission and Change State permission to suspend an object. If the user has only Change State permission, Documentum CM Server will attempt to suspend the object as the user defined in the `a_bpaction_run_as` property in the docbase config object. In those instances, that user must be either the owner or a superuser with Write permission or have Write and Change State permission on the object.

When an object is moved to an exception state, the server checks the state entry criteria and executes the actions on entry. The criteria must be satisfied and the actions completed to successfully move the object to the exception state.

It is possible to bypass the entry criteria. If you choose to do that, the server does not enforce the entry criteria, but simply performs the actions associated with the destination state and, on their completion, moves the object to the destination state. You must own the lifecycle policy object or be a superuser to bypass entry criteria.

Suspending an object is accomplished programmatically using one of the suspend methods in the `IDfSysObject` interface. Bypassing the entry criteria is accomplished by setting the override argument in the method set to true.

10.1.2.2.4 Resumptions

Resumption moves an object from an exception state back to the normal state from which it was suspended or back to the base state. Users who own an object or are superusers need only Write permission to resume the object. Other users must have Write permission and Change State permission to resume an object. If the user has only Change State permission, Documentum CM Server will attempt to resume the object as the user defined in the `a_bpaction_run_as` property in the docbase config object. In those instances, that user must be either the owner or a superuser with Write permission or have Write and Change State permission on the object.

Additionally, to resume an object back to the base state, the exception state must have the `return_to_base` property set to `TRUE`.

When an object is resumed to either the normal state or the base state, the object must satisfy the target state entry criteria and action on entry. The criteria must be satisfied and the actions completed to successfully resume the object to the destination state.

It is possible to bypass the entry criteria. If you choose to do that, the server does not enforce the entry criteria, but simply performs the actions associated with the destination state and, on their completion, moves the object to the destination state. You must own the lifecycle policy object or be a superuser to bypass entry criteria.

Programmatically, resuming an object is accomplished using one of the resume methods in the `IDfSysObject` interface. Bypassing the entry criteria is accomplished by setting the override argument in the method set to true.

10.1.2.2.5 Scheduled transitions

A scheduled transition is a transition from one state to another at a predefined date and time. If a lifecycle state is defined as allowing scheduled transitions, you can automate moving objects out of that state with scheduled transitions. All of the methods that move objects between states have a variation that allows you to schedule a transition for a particular date and time. If you issue a method that schedules the movement between states, Documentum CM Server creates a job for the state change.

The job scheduling properties are set to the specified date and time. The job runs as the user who issued the initial method that created the job, unless the `a_bpaction_run_as` property is set in the repository configuration object. If that is set, the job runs as the user defined in that property.

The destination state for a scheduled change can be an exception state or any normal state except the base state. You cannot schedule the same object for multiple state transitions at the same time.

You can unschedule a scheduled transition. Each of methods governing movement also has a variation that allows you to cancel a schedule change. For example, to cancel a scheduled promotion, you use `cancelSchedulePromote`.

10.1.2.3 Internal supporting methods

Installing Documentum CM Server installs a set of methods, implemented as method objects, that support lifecycle operations. There is a set for lifecycles that use Java and a corresponding set for lifecycles that use Docbasic. The following table lists the methods:

Method name		Purpose
Java	Docbasic	
`dm_bp_transition_java`	`dm_bp_transition`	Executes state transitions.
`dm_bp_batch_java`	`dm_bp_batch`	Invoked by `BATCH_PROMOTE` to promote objects in batches.
`dm_bp_schedule_java`	`dm_bp_schedule`	Invoked by jobs created for scheduled state changes. Calls `bp_transition` to execute the actual change.
`dm_bp_validate_java`	`dm_bp_validation`	Validates the lifecycle definition.

10.1.2.4 State changes

Movement from one state to another is handled by the `dm_bp_transition_java` and `dm_bp_transition` methods. The `dm_bp_transition_java` method is used for Java-based lifecycles. The `dm_bp_transition` method is used for Docbasic-based lifecycles.

When a user or application issues a promote, demote, suspend, or resume method that does not include a scheduling argument, the appropriate transition method is called immediately. If the state-change method includes a scheduling argument, the `dm_bp_schedule_java` (or `dm_bp_schedule`) method is invoked to create a job for the operation. The job scheduling properties are set to the date and time identified in the scheduling argument of the state-change method. When the job is executed, it invokes `dm_bp_transition_java` or `dm_bp_transition`.

Note: The `dm_bp_transition_java` and `dm_bp_transition` methods are also invoked by an attach method.

The `dm_bp_transition_java` and `dm_bp_transition` methods perform the following actions:

1. Use the supplied login ticket to connect to Documentum CM Server.
2. If the policy does not allow the object to move from the current state to the next state, return an error and exit.
3. Open an explicit transaction.
4. Execute the user entry criteria program.

Note: This step does not occur if the operation is a demotion.

5. Execute any system-defined actions on entry.
6. Execute any user-defined actions on entry.
7. If any one of the preceding steps fails, abort the transaction and return.
8. Set the `r_current_state` and `r_resume_state` properties. For an `attachPolicy` method, also set the `r_policy_id` and `r_alias_set_id` properties.
9. Save the SysObject.
10. If no errors occurred, commit the transaction.
11. If errors occurred, abort the transaction.
12. Execute any post-entry actions.

By default, the transition methods run as the user who issued the state-change method. To change the default, you must set the `a_bpaction_run_as` property in the docbase config object. If the `a_bpaction_run_as` property is set in the docbase config object, the actions associated with state changes are run as the user indicated in the property. Setting `a_bpaction_run_as` ensures that users with the extended permission Change State but without adequate access permissions to an object are able to change an object state. If the property is not set, the actions are run as the user who changed the state.

If an error occurs during execution of the `dm_bp_transition_java` or `dm_bp_transition` method, a log file is created. It is named `bp_transition_session_.out` in `%DOCUMENTUM%\dba\log\repository_id\bp` (`$DOCUMENTUM/dba/log/repository_id/bp`). If an error occurs during execution of the `dm_bp_schedule_java` or `dm_bp_schedule` methods, a log file named `bp_schedule_session_.out` is created in the same directory.

Note: If you set the `timeout_default` value for the `bp_transition` method to a value greater than five minutes, OpenText recommends that you also set the `client_session_timeout` key in the `server.ini` file to a value greater than that of `timeout_default`. The default value for `client_session_timeout` is five minutes. If a procedure run by `bp_transition` runs more than five minutes without making a call to Documentum CM Server, the client session will time out if the `client_sesion_timeout` value is five minutes. Setting `client_session_timeout` to a value greater than the value specified in `timeout_default` prevents that from happening.

- For more information about the `BATCH_PROMOTE` administration method, see *OpenText Documentum Content Management - Server DQL Reference Guide (EDCCS250400-DRD)*.
- For more about the jobs that process transitions, see "Scheduled transitions" on page 214.

10.1.3 Types of objects that can be attached to lifecycles

Lifecycles handle objects of type `dm_sysobject` or SysObject subtypes with one exception. The exception is policy objects (lifecycle definitions) you cannot attach a lifecycle definition to a lifecycle.

When you define a lifecycle, you specify which types of object it handles. Lifecycles are a reflection of the states of life of particular objects. Consequently, when you design a lifecycle, you are designing it with a particular object type or set of object types in mind. The scope of object types attachable to a particular lifecycle can be as broad or as narrow as needed. You can design a lifecycle to which any SysObject or SysObject subtype can be attached. You can also create a lifecycle to which only a specific subtype of `dm_document` can be attached.

If the lifecycle handles multiple types, the chosen object types must have the same supertype or one of the chosen types must be the supertype for the other included types.

The chosen object types are recorded internally in two properties: `included_type` and `include_subtypes`. These are repeating properties. The `included_type` property records, by name, the object types that can be attached to a lifecycle. The `include_subtypes` property is a Boolean property that records whether subtypes of the object types specified in `included_type` may be attached to the lifecycle. The value at a given index position in `include_subtypes` is applied to the object type identified at the corresponding position in `included_type`.

An object can be attached to a lifecycle if either:

- The lifecycle `included_type` property contains the document type, or
- The lifecycle `included_type` contains the document supertype and the value at the corresponding index position in the `include_subtypes` property is set to `TRUE`

For example, suppose a lifecycle definition has the following values in those properties:

```
included_type[0]=dm_sysobject
included_type[1]=dm_document
```

```
include_subtypes[0]=F
include_subtypes[1]=T
```

For this lifecycle, users can attach any object that is the `dm_sysobject` type. However, the only SysObject subtype that can be attached to the lifecycle is a `dm_document` or any of the `dm_document` subtypes.

The object type defined in the first index position (`included_type[0]`) is called the primary object type for the lifecycle. Object types identified in the other index positions in `included_type` must be subtypes of the primary object type.

You can define a default lifecycle for an object type. If an object type has a default lifecycle, when users create an object of that type, they can attach the lifecycle to the object without identifying the lifecycle specifically. Default lifecycles for object types are defined in the data dictionary.

10.1.4 Object permissions and lifecycles

Lifecycles do not override the object permissions of an attached object. Before you attach a lifecycle to an object, set the object permissions so that state transitions do not fail.

For example, suppose an action on entry moves an object to a different location (such as moving an approved SOP to an SOP folder). The object ACL must grant the user who promotes the document permission to move the document in addition to the permissions needed to promote the document. Promoting the document requires Write permission on the object and Change State permission if the user is not the object owner or a superuser. Moving the document to the SOP folder requires the Change Location permission and the appropriate base object-level permission to unlink the document from its current folder and link it to the SOP folder.

The actions associated with a state can be used to reset permissions as needed.

10.1.5 Entry criteria, actions on entry, and post-entry actions

Each state in a lifecycle may have entry criteria, actions on entry, and post-entry actions. Entry criteria are typically conditions that an object must fulfill to be a candidate to enter the state. Actions on entry are typically operations to be performed if the object meets the entry criteria. For example, changing the ACL might be an action on entry. Both entry criteria and actions on entry, if present, must successfully complete before the object is moved to the state. Post-entry actions are operations on the object that occur after the object is successfully moved to the state. For example, placing the object in a workflow might be a post-entry action.

Programs written for the entry criteria, actions on entry, and post-entry actions for a particular lifecycle must be either all Java programs or all Docbasic programs. You cannot mix programs in the two languages in one lifecycle.

Note: In entry criteria, you may use Docbasic Boolean expressions instead of or in addition to a program regardless of the language used for the programs in the actions and entry criteria.

It is possible to bypass entry criteria for a state. If you choose to do that, the server does not enforce the entry criteria, but simply performs the actions associated with the destination state and, on their completion, moves the object to the destination state. Only the owner of the policy object that stores the lifecycle definition or a superuser can bypass entry criteria.

- For more information, see “Actions on entry definitions” on page 223.
- For more information, see “Post-entry action definitions” on page 223.

10.2 Repository storage

The definition of a lifecycle is stored in the repository as a `dm_policy` object. The properties of the object define the states in the lifecycle, the object types to which the lifecycle may be attached, whether state extensions are used, and whether a custom validation program is used.

The state definitions within a lifecycle definition consist of a set of repeating properties. The values at a particular index position across those properties represent the definition of one state. The sequence of states within the lifecycle is determined by their position in the properties. The first state in the lifecycle is the state defined in index position [0] in the properties. The second state is the state defined in position [1], the third state is the state defined in index position [2], and so on.

State definitions include such information as the name of the state, a state type, whether the state is a normal or exception state, entry criteria, and actions to perform on objects in that state.

10.2.1 Lifecycle design phases

Lifecycle definitions are stored in the repository in one of three phases: draft, validated, and installed. A draft lifecycle definition is a definition that has been saved to the repository without validation. After the draft is validated, it is set to the validated state. After validation, the definition can be installed.

The state of a lifecycle definition is recorded in the `r_definition_state` property of the policy object.

Validation of a lifecycle definition ensures that the lifecycle is correctly defined and ready for use after it is installed. There are two system-defined validation programs: `dm_bp_validate_java` and `dm_bp_validate`. The Java method is invoked by Documentum CM Server for Java-based lifecycles. The other method is invoked for Docbasic-based lifecycles. Each method checks the following when validating a lifecycle:

- The policy object has at least one attachable state.
- The primary type of attachable object is specified, and all subtypes defined in the later position of the `included_type` property are subtypes of the primary attachable type.
- All objects referenced by object ID in the policy definition exist.
- For Java-based lifecycles, that all Service Based Objects (SBOs) referenced by service name exist.

In addition to the system-defined validation, you can write a custom validation program for use. If you provide a custom program, Documentum CM Server executes the system-defined validation first and then the custom program. Both programs must complete successfully to successfully validate the definition.

Validating a lifecycle definition requires at least Write permission on the policy object.

Lifecycles that have passed validation can be installed. Only after installation can users begin to attach objects to the lifecycle. A user must have Write permission on the policy object to install a lifecycle.

Internally, installation is accomplished using an install method.

- For more information about a detailed list of the information that makes up a state definition, see "Lifecycle state definitions" on page 221.
- For more information about writing a custom validation program, see "Custom validation programs" on page 225.

10.3 Designing a lifecycle

Lifecycles are typically created using the Lifecycle Editor, which is accessed through Documentum Composer. It is possible to create a lifecycle definition by directly issuing the appropriate methods or DQL statements to create, validate, and install the definition. However, using the Lifecycle Editor is the recommended and easiest way to create a lifecycle.

When you design a lifecycle, you must make the following decisions:

- What objects will use the lifecycle
- What normal and exception states the lifecycle will contain and the definition of each state

  A state definition includes a number of items, such as whether it is attachable, what its entry criteria, actions on entry, and post-entry actions are, and whether it allows scheduled transitions.
- Whether to include an alias set in the definition
- Whether you want to assign state types

  If objects attached to the lifecycle will be handled by the clients, you must assign state types to the states in the lifecycle. Similarly, if the objects will be handled by a custom application whose behavior depends upon a lifecycle state type, you must assign state types.
- What states, if any, will have state extensions
- Whether you want to use a custom validation program

- For information about how the object types whose instances may be attached to a lifecycle are specified, see "Types of objects that can be attached to lifecycles" on page 217.
- For information about guidelines for defining lifecycle states, see "Lifecycle state definitions" on page 221.
- For information about how alias sets are used with lifecycles, see "Lifecycles, alias sets, and aliases" on page 226.

- For information about the purpose and use of state types, see "State types" on page 227.
- For information about the purpose and use of state extensions, see "State extensions" on page 226.

10.4 Lifecycle state definitions

Each state in the lifecycle has a state definition. All of the information about states is stored in properties in the `dm_policy` object that stores the lifecycle definition. The properties that record a state definition are repeating properties, and the values at a particular index position across the properties represent the definition of one state in the lifecycle. If you are using the Lifecycle Editor to create a lifecycle, these properties are set automatically when you create the definition. If you are creating a lifecycle outside the Editor, you must set the properties yourself.

Lifecycle states have the following characteristics:

- State name

  Each state must have a name that is unique within the policy. State names must start with a letter, and cannot contain colons, periods, or commas. the `state_name` property of the `dm_policy` object holds the names of the states.

- Attachability

  Attachability is the state characteristic that determines whether users can attach an object to the state. A lifecycle must have at least one normal state to which users can attach objects. It is possible for all normal states in a lifecycle to allow attachments. The number of states in a lifecycle that allow attachments depends on the lifecycle.

  Whether a state allows attachments is defined in the `allow_attach` property. This is a Boolean property.

- Base state

  The starting point in the lifecycle to which an object might be returned after a particular action.

- Demotion

  Demotion moves an object from one state in a lifecycle to a previous state. If an object in a normal state is demoted, it moves to the previous normal state. If an object in an exception state is demoted, it moves to the base state.

  The ability to demote an object from a particular state is part of the state definition. By default, states do not allow users to demote objects. Choosing to allow users to demote objects from a particular state sets the `allow_demote` property to `TRUE` for that state.

  When an object is demoted, Documentum CM Server does not check the entry criteria of the target state. However, Documentum CM Server does perform the system and user-defined actions on entry and post-entry actions.

- Scheduled transitions

  A scheduled transition moves an object from one state to another at a scheduled date and time. Normal states can allow scheduled promotions to the next normal state or a demotion to the base state. Exception states can allow a scheduled resumption to a normal state or a demotion to the base state.

  Whether a state can be scheduled to transition to another state is recorded in the `allow_schedule` property. This property is set to `TRUE` if you decide that transitions out of the state may be scheduled. It is set to `FALSE` if you do not allow scheduled transitions for the state.

  The setting of this property only affects whether objects can be moved out of a particular state at scheduled times. It has no effect on whether objects can be moved into a state at a scheduled time. For example, suppose StateA allows scheduled transitions and StateB does not. Those settings mean that you can promote an object from StateA to StateB on a scheduled date, but you cannot demote an object from StateB to StateA on a scheduled date.

- Entry criteria

  Entry criteria are the conditions an object must meet before the object can enter a normal or exception state when promoted, suspended, or resumed. The entry criteria are not evaluated if the action is a demotion. Each state may have its own entry criteria.

  If the lifecycle is Java-based, the entry criteria can be:

  – A Java program

    Access the lifecycle through the interface `IDfLifecycleUserEntryCriteria`.

  – One or more Boolean expressions

  – Both Boolean expressions and a Java program

  Java-based programs are stored in the repository as SBO modules and a JAR file. For more information about SBO modules, see *OpenText Documentum Content Management - Foundation Java API Development Guide (EDCPKCL250400-DGD)*.

  If the lifecycle is Docbasic-based, the entry criteria can be:

  – A Docbasic program

  – One or more Boolean expressions

  – Both Boolean expressions and a Docbasic program

10.4.1 Actions on entry definitions

In addition to entry criteria, you can define actions on entry for a state. You can use actions on entry to perform such actions as changing a object ACL, changing an object repository location, or changing an object version label. You can also use an action on entry to enforce a signature requirement. Actions on entry are performed after the entry criteria are evaluated and passed. The actions must complete successfully before an object can enter the state.

A set of pre-defined actions on entry are available through the Lifecycle Editor. You can choose one or more of those actions, define your own actions on entry, or both.

If you define your own actions on entry, the program must be a Java program if the lifecycle is Java-based. Java-based actions on entry are stored in the repository as SBO modules and a JAR file. If the lifecycle is Docbasic-based, the actions on entry program must be a Docbasic program.

If both system-defined and user-defined actions on entry are specified for a state, the server performs the system-defined actions first and then the user-defined actions. An object can only enter the state when all actions on entry complete successfully.

Actions on entry include:

- System-defined actions

  A set of pre-defined actions on entry are available for use. When you create or modify a lifecycle using Lifecycle Editor, you can choose one or more of these actions.

- Java programs

  A Java program used as an action on entry program must implement the interface `IDfLifecycleUserAction`.

- Docbasic programs

  Docbasic actions on entry programs are stored in the repository as `dm_procedure` objects. The object IDs of the procedure objects are recorded in the `user_action_id` property. These properties are set internally when you identify the programs while creating or modifying a lifecycle using Lifecycle Editor.

10.4.2 Post-entry action definitions

Post-entry actions are actions performed after an object enters a state. You can define post-entry actions for any state. For example, for a Review state, you might want to add a post-entry action that puts the object into a workflow that distributes the object for review. Or, when a document enters the Publish state, perhaps you want to send the document to an automated publishing program.

If the lifecycle is Docbasic-based, the post-entry action programs must be Docbasic programs.

Poet-entry actions can be:

- Java-based

  If the lifecycle is Java-based, the post-entry action programs must be Java programs. The programs are stored in the repository as SBO modules and a JAR file. A Java program used as an post-entry action program must implement the `IDfLifecycleUserPostProcessing` interface.

- Docbasic-based

  Docbasic post-entry actions are functions named `PostProc` and follow a specific format.

10.4.3 Including electronic signature requirements

Because entry criteria and actions on entry are processed before an object is moved to the target state, you can use a program for entry criteria or actions on entry to enforce sign-off requirements for objects moving to that state. In the program, include code that asks the user to provide a sign-off signature. When a user attempts to promote or resume an object to the state, the code can ensure that if the user sign-off does not succeed, the entry criteria or action does not complete successfully and the object is not moved to the state.

10.4.4 Using aliases in actions

Aliases provide a way to make the actions you define for a state flexible and usable in multiple contexts. Many documents may have the same life stages, but have differing business requirements. For example, most documents go through a writing draft stage, a review stage, and a published or approved stage. However, some of those documents may be marketing documents, some may be engineering documents, and some may be human resource documents. Each kind of document requires different users to write, review, and approve them.

Using aliases in actions can make it possible to design one lifecycle that can be attached to all these kinds of documents. You can substitute an alias for a user or group name in an ACL and in certain properties of a SysObject. You can use an alias in place of a path name in the `Link` and `Unlink` methods.

In template ACLs, aliases can take the place of the accessor name in one or more access control entries. When the ACL is applied to an object, the server copies the template, resolves the aliases in the copy to real names, and assigns the copy to the object.

In the `Link` and `Unlink` methods, aliases can replace the folder path argument. When the method is executed, the alias is resolved to a folder path and the object is linked to or unlinked from the proper folder.

When the actions you define for a state assign a new ACL to an object or use the `Link` or `Unlink` methods, using template ACLs and aliases in the folder path arguments ensures that the ACL for an object or its linked locations are always appropriate.

- For information about adding a signature requirement to a program, see "Including electronic signature requirements" on page 224.
- For information about how scheduled transitions are implemented internally, see "Scheduled transitions" on page 214.
- For information about the listing of the pre-defined actions, see the *OpenText Documentum Composer* documentation.
- For information about simple modules, see *OpenText Documentum Content Management - Foundation Java API Development Guide (EDCPKCL250400-DGD)*.
- For information about using digital and electronic signatures or simple sign-offs, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.
- For information about how aliases are implemented, "Aliases" on page 229.

10.5 Custom validation programs

This section outlines the basic procedure for creating and installing a user-defined lifecycle validation method. The OpenText Documentum CM system provides standard technical support only for the default validation method installed with the Documentum CM Server software. For assistance in creating, implementing, or debugging a user-defined validation method, contact OpenText Global Technical Services.

If you want to use a custom validation program, the program must be written in the same language as that used for any entry criteria, actions on entry, or post-entry actions written for the lifecycle. This means that if those programs are written in Java, the custom validation program must be in Java also. If the programs are written in Docbasic, the validation program must be in Docbasic also.

Note: Docbasic is a deprecated language.

After you write the program, use Documentum Composer to add the custom validation program to the lifecycle definition. You must own the lifecycle definition (the policy object) or have at least Version permission on it to add a custom validation program to the lifecycle.

The *Documentum Composer* documentation contains instructions for creating a custom validation program and adding to a lifecycle definition.

10.6 Integrating lifecycles and applications

This section discusses the lifecycle features that make it easy to integrate lifecycles and applications.

10.6.1 Lifecycles, alias sets, and aliases

A lifecycle definition can reference one or more alias sets. When an object is attached to the lifecycle, Documentum CM Server chooses one of the alias sets in the lifecycle definition as the alias set to use to resolve any aliases found in the attached object's properties. Sysobjects can use aliases in the `owner_name`, `acl_name`, and `acl_domain` properties. Which alias set is chosen is determined by how the client application is designed. The application may display a list of the alias sets to the user and allow the user to pick one. Or, the application may use the default resolution algorithm for choosing the alias set.

Additionally, you can use template ACLs, which contain aliases, and aliases in folder paths in actions defined for states to make the actions usable in a variety of contexts.

If you define one or more alias sets for a lifecycle definition, those choices are recorded in the policy object's `alias_set_ids` property.

10.6.2 State extensions

State extensions are used to provide additional information to applications for use when an object is in a particular state. For example, an application may require a list of users who have permission to sign off a document when the document is in the Approval state. You can provide such a list by adding a state extension to the Approval state.

Note: Documentum CM Server does not use information stored in state extensions. Extensions are solely for use by client applications.

You can add a state extension to any state in a lifecycle. State extensions are stored in the repository as objects. The objects are subtypes of the `dm_state_extension` type. The `dm_state_extension` type is a subtype of `dm_relation` type. Adding state extensions objects to a lifecycle creates a relationship between the extension objects and the lifecycle.

If you want to use state extensions with a lifecycle, determine what information is needed by the application for each state requiring an extension. When you create the state extensions, you will define a `dm_state_extension` subtype that includes the properties that store the information required by the application for the states. For example, suppose you have an application called `EngrApp` that will handle documents attached to LifecycleA. This lifecycle has two states, Review and Approval, that require a list of users and a deadline date. The state extension subtype for this lifecycle will have two defined properties: `user_list` and `deadline_date`. Or perhaps the application needs a list of users for one state and a list of possible formats for another. In that case, the properties defined for the state extension subtypes will be `user_list` and `format_list`.

State extension objects are associated with particular states through the `state_no` property, inherited from the `dm_state_extension` supertype.

State extensions must be created manually. The Lifecycle Editor does not support creating state extensions.

10.6.3 State types

A state type is a name assigned to a lifecycle state that can be used by applications to control behavior of the application. Using state types makes it possible for a client application to handle objects in various lifecycles in a consistent manner. The application bases its behavior on the type of the state, regardless of the state's name or the including lifecycle.

Document Control Management and Web Content Management expect the states in a lifecycle to have certain state types. The behavior of either Document Control Management or Web Content Management when handling an object in a lifecycle is dependent on the state type of the object's current state. When you create a lifecycle for use with objects that will be handled using Document Control Management or Web Content Management, the lifecycle states must have state types that correspond to the state types.

Custom applications can also use state types. Applications that handle and process documents can examine the `state_type` property to determine the type of the object's current state and then use the type name to determine the application behavior.

In addition to the repeating property that defines the state types in the policy object, state types may also be recorded in the repository using `dm_state_type` objects. State type objects have two properties: `state_type_name` and `application_code`. The `state_type_name` identifies the state type and `application_code` identifies the application that recognizes and uses that state type. You can create these objects for use by custom applications. For example, installing Document Control Management creates state type objects for the state types recognized by Document Control Management. Document Control Management uses the objects to populate pick lists displayed to users when users are creating lifecycles.

Use the Lifecycle Editor to assign state types to states and to create state type objects. If you have subtyped the state type object type, you must use the API or DQL to create instances of the subtype.

10.6.4 Lifecycle scope for SysObjects and aliases in actions

- For more information about the default resolution algorithm for choosing the alias set to be used with a lifecycle, see “Determining the lifecycle scope for SysObjects” on page 232.
- For more information about using aliases in templates and lifecycle actions, see “Using aliases in actions” on page 224.

Chapter 11

Aliases

This chapter describes how aliases are implemented and used. Aliases support Documentum CM Server's process management services.

11.1 Overview

Aliases are placeholders for user names, group names, or folder paths. You can use an alias in the following places:

- In SysObjects or SysObject subtypes, in the `owner_name`, `acl_name`, and `acl_domain` properties
- In ACL template objects, in the `r_accessor_name` property

  Note: Aliases are not allowed as the `r_accessor_name` for ACL entries of type `RequiredGroup` or `RequiredGroupSet`.
- In workflow activity definitions (`dm_activity` objects), in the `performer_name` property
- In a `link` or `Unlink` method, in the folder path argument

You can write applications or procedures that can be used and reused in many situations because important information such as the owner of a document, a workflow activity performer, or the user permissions in a document ACL is no longer hard coded into the application. Instead, aliases are placeholders for these values. The aliases are resolved to real user names, group names, or folder paths when the application executes.

For example, suppose you write an application that creates a document, links it to a folder, and then saves the document. If you use an alias for the document `owner_name` and an alias for the folder path argument in the link method, you can reuse this application in any context. The resulting document will have an owner that is appropriate for the application context and be linked into the appropriate folder also.

The application becomes even more flexible if you assign a template ACL to the document. Template ACLs typically contain one or more aliases in place of accessor names. When the template is assigned to an object, the server creates a copy of the ACL, resolves the aliases in the copy to real user or group names, and assigns the copy to the document.

Aliases are implemented as objects of type `dm_alias_set`. An alias set object defines paired values of aliases and their corresponding real values. The values are stored in the repeating properties `alias_name` and `alias_value`. The values at each index position represent one alias and the corresponding real user name, group name, or folder path.

For example, given the pair `alias_name[0]=engr_vp` and `alias_value[0]=henryp`, `engr_vp` is the alias and `henryp` is the corresponding real user name.

11.2 Defining aliases

When you define an alias in place of a user name, group name, or folder path, use the following format for the alias specification:

%[*alias_set_name*.]*alias_name*

alias_set_name identifies the alias set object that contains the specified alias name. This value is the `object_name` of the alias set object. Including *alias_set_name* is optional.

alias_name specifies one of the values in the `alias_name` property of the alias set object.

To put an alias in a SysObject or activity definition, use a set method. To put an alias in a template ACL, use a grant method. To include an alias in a link or unlink method, substitute the alias specification for the folder path argument.

For example, suppose you have an alias set named `engr_aliases` that contains an `alias_name` called `engr_vp`, which is mapped to the user name `henryp`. If you set the `owner_name` property to `%engr_alias.engr_vp`, when the document is saved to the repository, the server finds the alias set object named `engr_aliases` and resolves the alias to the user name `henryp`.

It is also valid to specify an alias name without including the alias set name. In such cases, the server uses a predefined algorithm to search one or more alias scopes to resolve the alias name.

11.3 Alias scopes

The alias scopes define the boundaries of the search when the server resolves an alias specification.

If the alias specification includes an alias set name, the alias scope is the alias set named in the alias specification. The server searches that alias set object for the specified alias and its corresponding value.

If the alias specification does not include an alias set name, the server resolves the alias by searching a predetermined, ordered series of scopes for an alias name matching the alias name in the specification. The scopes that are searched depend on where the alias is found.

11.3.1 Workflow alias scopes

To resolve an alias in an activity definition that does not include an alias set name, the server searches one or more of the following scopes:

- Workflow
- Session
- User performer of the previous work item
- The default group of the previous work item performer
- Server configuration

Within the workflow scope, the server searches in the alias set defined in the workflow object `r_alias_set_id` property. This property is set when the workflow is instantiated. The server copies the alias set specified in the `perf_alias_set_id` property of the workflow definition (process object) and sets the `r_alias_set_id` property in the workflow object to the object ID of the copy.

Within the session scope, the server searches the alias set object defined in the session configuration `alias_set` property.

In the user performer scope, the server searches the alias set defined for the user who performed the work item that started the activity containing the alias. A user alias set is defined in the `alias_set_id` property of the user object.

In the group scope, the server searches the alias set defined for the default group of the user who performed the work item that started the activity containing the alias. The group alias set is identified in the `alias_set_id` property.

Within the server configuration scope, the search is conducted in the alias set defined in the `alias_set_id` property of the server config object.

11.3.2 Non-workflow alias scopes

Aliases used in non-workflow contexts have the following possible scopes:

- Lifecycle
- Session
- User
- Group
- Server configuration

When the server searches within the lifecycle scope, it searches in the alias set defined in the SysObject `r_alias_set_id` property. This property is set when the object is attached to a lifecycle.

Within the session scope, the server searches the alias set object defined in the session configuration `alias_set` property.

Within the user scope, the search is in the alias set object defined in the `alias_set_id` property of the user object. The user is the user who initiated the action that caused the alias resolution to occur. For example, suppose a a document is promoted and the actions of the target state assign a template ACL to the document. The user in this case is either the user who promoted the document or, if the promotion was part of an application, the user account under which the application runs.

In the group scope, the search is in the alias set object associated with the user default group.

Within the system scope, the search is in the alias set object defined in the `alias_set_id` property of the server config object.

11.3.3 Determining the lifecycle scope for SysObjects

A SysObject lifecycle scope is determined when a policy is attached to the SysObject. If the policy object has one or more alias sets listed in its `alias_set_ids` property, you can either choose one from the list as the object lifecycle scope or allow the server to choose one by default.

The server uses the following algorithm to choose a default lifecycle scope:

- The server uses the alias set defined for the session scope if that alias set is listed in the policy object `alias_set_ids` property.
- If the session scope's alias set isn't found, the server uses the alias set defined for the user's scope if it is in the `alias_set_ids` list.
- If the user scope alias set is not found, the server uses the alias set defined for the user default group if that alias set is in the `alias_set_ids` list.
- If the default group scope alias set is not found, the server uses the alias set defined for the system scope if that alias set is in the `alias_set_ids` list.
- If the system scope's alias set isn't found, the server uses the first alias set listed in the `alias_set_ids` property.

If the policy object has no defined alias set objects in the `alias_set_ids` property, the SysObject `r_alias_set_id` property is not set, and an error is generated.

11.4 Resolving aliases in SysObjects

The server resolves an alias in a SysObject when the object is saved to the repository for the first time.

If there is no `alias_set_name` defined in the alias specification, the server uses the following algorithm to resolve the `alias_name`:

- The server first searches the alias set defined in the object `r_alias_set_id` property. This is the lifecycle scope.
- If the alias is not found in the lifecycle scope or if `r_alias_set_id` is undefined, the server looks next at the alias set object defined for the session scope.

- If the alias is not found in the session scope, the server looks at the alias set defined for the user scope.
- If the alias is not found in the user scope, the server looks at the alias set defined for the user default group scope.
- If the alias is not found in the user default group scope, the server looks at the alias set defined for the system scope.

If the server does not find a match in any of the scopes, it returns an error.

11.5 Resolving aliases in template ACLs

An alias in a template ACL is resolved when the ACL is applied to an object.

If an alias set name is not defined in the alias specification, the server resolves the alias name in the following manner:

- If the object to which the template is applied has an associated lifecycle, the server resolves the alias using the alias set defined in the `r_alias_set_id` property of the object. This alias set is the object lifecycle scope. If no match is found, the server returns an error.
- If the object to which the template is applied does not have an attached lifecycle, the server resolves the alias using the alias set defined for the session scope. This is the alias set identified in the `alias_set` property of the session config object. If a session scope alias set is defined, but no match is found, the server returns an error.
- If the object has no attached lifecycle and there is no alias defined for the session scope, the server resolves the alias using the alias set defined for the user scope. This is the alias set identified in the `alias_set_id` property of the `dm_user` object for the current user. If a user scope alias set is defined but no match is found, the server returns an error.
- If the object has no attached lifecycle and there is no alias defined for the session or user scope, the server resolves the alias using the alias set defined for the user default group. If a group alias set is defined but no match is found, the system returns an error.
- If the object has no attached lifecycle and there is no alias defined for the session, user, or group scope, the server resolves the alias using the alias set defined for the system scope. If a system scope alias set is defined but no match is found, the system returns an error.

If no alias set is defined for any level, Documentum CM Server returns an error stating that an error set was not found for the current user.

11.6 Resolving aliases in Link and Unlink methods

An alias in a `Link` or `Unlink` method is resolved when the method is executed. If there is no alias set name defined in the alias specification, the server resolves the alias name with the algorithm used for resolving aliases in SysObjects.

11.7 Resolving aliases in workflows

In workflows, aliases can be resolved when:

- The workflow is started
- An activity is started

Resolving aliases when the workflow is started requires user interaction. The person starting the workflow provides alias values for any unpaired alias names in the workflow definition alias set.

Resolving an alias when an activity starts is done automatically by the server.

11.7.1 Resolving aliases during workflow startup

A workflow definition can include an alias set to be used to resolve aliases found in the workflow activities. The alias set can have alias names that have no corresponding alias values. Including an alias set with missing alias values in the workflow definition makes the definition a flexible workflow template. It allows the workflow starter to designate the alias values when the workflow is started.

When the workflow is instantiated, the server copies the alias set and attaches the copy to the workflow object by setting the workflow `r_alias_set_id` property to the object ID of the copy.

If the workflow is started through a client application, the application prompts the starter for alias values for the missing alias names. The server adds the alias values to the alias set copy attached to the workflow object. If the workflow is started through a custom application, the application must prompt the workflow starter for the absent alias values and add them to the alias set.

If the workflow scope is used at runtime to resolve aliases in the workflow activity definitions, the scope will have alias values that are appropriate for the current instance of the workflow.

Note: The server generates a runtime error if it matches an alias in an activity definition to an unpaired alias name in a workflow definition.

11.7.2 Resolving aliases during activity startup

The server resolves aliases in activity definitions at runtime, when the activity is started. The alias scopes used in the search for a resolution depend on how the designer defined the activity. There are three possible resolution algorithms:

- Default
- Package
- User

11.7.2.1 Default resolution algorithm

The server uses the default resolution algorithm when the activity `resolve_type` property is set to 0. The server searches the following scopes, in the order listed:

- Workflow
- Session
- User performer of the previous work item
- The default group of the previous work item performer
- Server configuration

The server examines the alias set defined in each scope until a match for the alias name is found.

11.7.2.2 Package resolution algorithm

The server uses the package resolution algorithm if the activity's `resolve_type` property is set to `1`. The algorithm searches only the package or packages associated with the activity incoming ports. Which packages are searched depends on the setting of the activity `resolve_pkg_name` property.

If the `resolve_pkg_name` property is set to the name of a package, the server searches the alias sets of the package components. The search is conducted in the order in which the components are stored in the package.

If the `resolve_pkg_name` property is not set, the search begins with the package defined in `r_package_name[0]`. The components of that package are searched. If a match is not found, the search continues with the components in the package identified in `r_package_name[1]`. The search continues through the listed packages until a match is found.

11.7.2.3 User resolution algorithm

The server uses the user resolution algorithm if the activity `resolve_type` property is set to `2`. In such cases, the search is conducted in the following scopes:

- The alias set defined for the user performer of the previous work item
- The alias set defined for the default group of the user performer of the previous work item

The server first searches the alias set defined for the user. If a match isn't found, the server searches the alias set defined for the user default group.

11.7.2.4 When a match is found

When the server finds a match in an alias set for an alias in an activity, the server checks the `alias_category` value of the match. The `alias_category` value must be one of:

- 1 (user)
- 2 (group)
- 3 (user or group)

If the `alias_category` is appropriate, the server next determines whether the alias value is a user or group, depending on the setting in the activity `performer_type` property. For example, if `performer_type` indicates that the designated performer is a user, the server will validate that the alias value represents a user, not a group. If the alias value matches the specified `performer_type`, the work item is created for the activity.

11.7.2.5 Resolution errors

If the server does not find a match for an alias, or finds a match but the associated alias category value is incorrect, the server:

- Generates a warning
- Posts a notification to the inbox of the workflow supervisor
- Assigns the work item to the supervisor

Chapter 12
Internationalization summary

This chapter describes the approach to server internationalization, or how Documentum CM Server handles code pages. It discusses Unicode, which is the foundation of internationalization at OpenText Documentum CM, and summarizes the internationalization requirements of various features of Documentum CM Server.

12.1 Overview

Internationalization refers to the ability of the Documentum CM Server to handle communications and data transfer between itself and client applications in a variety of code pages. This ability means that the Documentum CM Server does not make assumptions based on a single language or locale. A locale represents a specific geographic region or language group.

Documentum CM Server runs internally with the UTF-8 encoding of Unicode. The Unicode Standard provides a unique number to identify every letter, number, symbol, and character in every language. UTF-8 is a varying-width encoding of Unicode, with each single character represented by one to four bytes.

Documentum CM Server handles transcoding of data from national character sets (NCS) to and from Unicode. A national character set is a character set used in a specific region for a specific language. For example, the Shift-JIS and EUC-JP character sets are used for representing Japanese characters. ISO-8859-1 (sometimes called Latin-1) is used for representing English and European languages. Data can be transcoded from a national character set to Unicode and back without data loss. Only common data can be transcoded from one NCS to another. Characters that are present in one NCS cannot be transcoded to an NCS in which they are not available.

Note: Internationalization and localization are different concepts. Localization is the ability to display values such as names and dates in the languages and formats specific to a locale. Documentum CM Server uses a data dictionary to provide localized values for applications. For information about the data dictionary and localization, see "Data model" on page 43.

12.2 Content files

You can store content files created in any code page and in any language in a repository. The files are transferred to and from a repository as binary files.

Note: OpenText recommends that all XML content use one code page.

12.3 Metadata

The metadata values you can store depend on the code page of the underlying database. The code page may be a national character set or it may be Unicode.

If the database was configured using a national character set as the code page, you can store only characters allowed by that code page. For example, if the database uses EUC-KR, you can store only characters that are in the EUC-KR code page as metadata.

All code pages supported by OpenText Documentum CM include ASCII as a subset of the code page. You can store ASCII metadata in databases using any supported code page.

If you configured the database using Unicode, you can store metadata using characters from any language. However, your client applications must be able to read and write the metadata without corrupting it. For example, a client using the ISO-8859-1 (Latin-1) code page internally cannot read and write Japanese metadata correctly. Client applications that are Unicode-compliant can read and write data in multiple languages without corrupting the metadata.

12.4 Client communications with Documentum CM Server

All communications between Foundation Java API and Documentum CM Server are performed using the UTF-8 (Unicode) code page.

12.5 Constraints

A UTF-8 Unicode repository can store metadata from any language. However, if your client applications are using incompatible code pages in national character sets, they may not be able to handle metadata values set in different code page. For example, if an application using Shift-JIS or EUC-JP (the Japanese code pages) stores objects in the repository and another application using ISO-8859-1 (Latin-1 code page) retrieves that metadata, the values returned to the ISO-8859-1 application will be corrupted because there are characters in the Japanese code page that are not found in the Latin-1 code page.

12.6 Configuration requirements for internationalization

Before you install Documentum CM Server, it is important to set the Documentum CM Server host locale and code page properly and to install the database to use a supported code page. For information about the values that are set prior to installing Documentum CM Server, see *OpenText Documentum Content Management - Server and Server Extensions Installation Guide (EDCSY250400-IGD)*.

During the Documentum CM Server installation process, a number of configuration parameters are set in the `server.ini` file and server config object that define the expected code page for clients and the host machine operating system. These parameters are used by Documentum CM Server in managing data, user authentication, and other functions.

There are recommended locales for the Documentum CM Server host and recommended code pages for Documentum CM Server host and database.

12.6.1 Values set during installation

Some locales and code pages are set during Documentum CM Server installation and repository configuration. The following sections describe what is set during installation.

12.6.1.1 server config object

The server config object describes a Documentum CM Server and contains information that the server uses to define its operations and operating environment.

- `locale_name`

  The locale of the server host, as defined by the host operating system. The value is determined programmatically and set during server installation. The `locale_name` determines which data dictionary locale labels are served to clients that do not specify their locale.

- `default_client_codepage`

  The default code page used by clients connecting to the server. The value is determined programmatically and set during server installation. OpenText strongly recommends that you do not reset the value.

- `server_os_codepage`

  The code page used by the server host. Documentum CM Server uses this code page when it transcodes user credentials for authentication and the command-line arguments of server methods. The value is determined programmatically and set during server installation. OpenText strongly recommends that you do not reset the value.

For information about the default values for code pages by locale, see *OpenText Documentum Content Management - Server Administration and Configuration Guide (EDCCS250400-AGD)*.

12.6.2 Values set during sessions

Properties defining code pages are set when Foundation Java API is initialized and when a session is started.

12.6.2.1 client config object

A client config object records global information for client sessions. It is created when Foundation Java API is initialized. The values reflect the information found in the `dfc.properties` file used by the Foundation Java API instance. Some of the values are then used in the session config object when a client opens a repository session.

The following properties for internationalization are present in a client config object:

- `dfc.codepage`

  The `dfc.codepage` property controls conversion of characters between the native code page and UTF-8. The value is taken from the `dfc.codepage` key in the `dfc.properties` file on the client host. This code page is the preferred code page for repository sessions started using the Foundation Java API instance. The value of `dfc.codepage` overrides the value of the `default_client_codepage` property in the server config object.

  The default value for this key is `UTF-8`.

- `dfc.locale`

  This is the client preferred locale for repository sessions started by the Foundation Java API instance.

12.6.2.2 session config object

A session config object describes the configuration of a repository session. It is created when a client opens a repository session. The property values are taken from values in the client config object, the server config object, and the connection config object.

The following properties for internationalization are set in the session config object:

- `session_codepage`

  This property is obtained from the client config object `dfc.codepage` property. It is the code page used by a client application connecting to the server from the client host.

  If needed, set the `session_codepage` property in the session config object early in the session and do not reset it.

- `session_locale`

The locale of the repository session. The value is obtained from the `dfc.locale` property of the client config object. If `dfc.locale` is not set, the default value is determined programmatically from the locale of the client host machine.

12.6.2.3 Setting values

The values of the `dfc.codepage` and `dfc.locale` properties in the client config object determine the values of `session_codepage` and `session_locale` in the session config object. These values are determined in the following manner:

1. Use the values supplied programmatically by an explicit set on the client config object or session config object.
2. If the values are not explicitly set, examine the settings of `dfc.codepage` and `dfc.locale` keys in the `dfc.properties` file.

   If not explicitly set, the `dfc.codepage` key and `dfc.locale` keys are assigned default values. Foundation Java API derives the default values from the Java Virtual Machine (JVM), which gets the defaults from the operating system.

12.7 Where ASCII must be used

Some objects, names, directories, and property values must contain only ASCII characters. These include:

- Documentum CM Server's host machine name
- Repository names
- Repository owner user name and password
- Installation owner user name and password
- Registered table names and column names
- The directory in which Documentum CM Server is installed
- Location object names
- The value of the `file_system_path` property of a location object
- Mount point object names
- Format names
- Format DOS extensions
- All file store names
- Object type names and property names
- Federation names
- Content stored in turbo storage
- String literals included in check constraint definitions
- String literals included in the expression string referenced in the conditional clauses of value assistance definitions

- Text specified in an AS clause in a `SELECT` statement

12.8 Other requirements

12.8.1 User names, email addresses, and group names

There are code page-based requirements for the following property values:

- `dm_user.user_name`
- `dm_user.user_os_name`
- `dm_user.user_db_name`
- `dm_user.user_address`
- `dm_group.group_name`

The requirements for these differ depending on the site configuration. If the repository is a standalone repository, the values in the properties must be compatible with the code page defined in the server `server_os_codepage` property. A standalone repository does not participate in object replication or a federation, and its users never access objects from remote repositories.

If the repository is in an installation with multiple repositories but all repositories have the same code page defined in `server_os_codepage`, the values in the user property must be compatible with the `server_os_codepage`. However, if the repositories have different code pages identified in `server_os_codepage`, the values in the properties listed must consist of only ASCII characters.

12.8.2 Lifecycles

The scripts that you use as actions in lifecycle states must contain only ASCII characters.

12.8.3 Docbasic

Docbasic does not support Unicode. For all Docbasic server methods, the code page in which the method is written and the code page of the session the method opens must be the same and must both be the code page of the Documentum CM Server host (the `server_os_codepage`).

Docbasic scripts that run on client machines must be in the code page of the client operating system.

12.8.4 Federations

Federations are created to keep global users, groups, and external ACLs synchronized among member repositories.

A federation can include repositories using different server operating system code pages (`server_os_codepage`). In a mixed-code page federation, the following user and group property values must use only ASCII characters:

- `user_name`
- `user_os_name`
- `user_address`
- `group_address`

ACLs can use Unicode characters in ACL names.

12.8.5 Object replication

When object replication is used, the databases for the source and target repositories must use the same code page or the target repository must use Unicode. For example, you can replicate from a Japanese repository to a French repository if the French repository database uses Unicode. If the French repository database uses Latin-1, replication fails.

In mixed code page environments, the source and target folder names must contain only ASCII characters. The folders contained by the source folder are not required to be named with only ASCII characters.

12.8.6 Other cross-repository operations

In other cross-repository operations, such as copying folders from one repository to another, the user performing the operation must have identical user credentials (user names and passwords and email addresses) in the two repositories.