

Content Server Fundamentals



Version 5.3 SP1
September 2005

Copyright © 1994-2005 EMC Corporation

Table of Contents

Preface	19	
Chapter 1	Introducing Content Server	21
	Content Server's role in the product suite	21
	Content management services	21
	Storage and retrieval	22
	Versioning	22
	Data dictionary	23
	Assembly and publishing	23
	Collaborative Services	23
	Retention Policy Services	24
	Process management features	24
	Workflows	24
	Life cycles	25
	Security features	25
	Repository security	25
	Accountability	26
	Distributed services	26
	Content Server architecture	26
	Internationalization	27
	Communicating with Content Server	27
	Documentum client applications	27
	Custom applications	27
	Interactive utilities	28
Chapter 2	Session and Transaction Management	29
	Introducing repository sessions	29
	Session configuration	30
	Primary sessions and subconnections	31
	Primary sessions	31
	Subconnections	31
	Inactive repository sessions	32
	Restricted sessions	32
	Closing repository sessions	32
	Role of the connection broker	33
	Specifying the connection broker at runtime	33
	Requesting a native or secure connection	34
	Using connection pooling	34
	At the DMCL level	34
	At the application level	35
	Login tickets	35
	Ticket format	36
	The login ticket key	36

Login ticket scope.....	37
Trusted repositories	37
Ticket expiration.....	38
Note on host machine time and login tickets.....	38
Revoking tickets	39
Restricting superuser use	39
Application access control tokens	39
How tokens work	40
Format of AAC tokens	41
Token scope	41
Token generation.....	42
Token expiration.....	42
Internal methods, user methods, and tokens.....	43
Defining the repository scope	43
Command line scoping	43
Default scoping	44
Concurrent sessions	44
Persistent client caches.....	45
Object caches.....	45
Type and data dictionary caches	46
Query caches.....	46
Subconnections and persistent caching.....	47
Using persistent client caching in an application.....	48
Identifying objects and queries for caching.....	48
Consistency checking.....	49
Determining if a consistency check is needed	50
When a keyword or integer defines the rule	50
When a cache config object defines the rule	50
Conducting consistency checks	51
Type and data dictionary cache consistency checks	51
The default consistency check rule.....	52
Cachequery consistency checks	52
Transaction management	52
Constraints on explicit transactions.....	53
Database-level locking in explicit transactions	54
Managing deadlocks.....	54
Handling deadlocks in internal transactions.....	54
Handling deadlocks in explicit transactions.....	55
Chapter 3 The Data Model	57
Introducing object types and objects	57
Supertypes, subtypes, and inheritance	58
Persistence	58
Naming	58
Content files and object types	59
Attributes	59
Repositories	60
Object type tables	60
Single-valued attribute tables	61
Repeating attribute tables.....	61
Defining the location and extents of object type tables.....	62
Object type index tables	63
Content storage areas.....	63
Registered tables	64
The data dictionary.....	64

Usage	64
Localization	64
Modifying the data dictionary	65
Publishing the data dictionary	65
What the data dictionary can contain	66
Constraints	66
Unique key	67
Primary key	67
Foreign key	68
Not null	69
Check	69
Default lifecycles for types	69
Component specifications	70
Default values for attributes	70
Localized text	70
Value assistance	70
Mapping information	71
The data dictionary and lifecycle states	71
Retrieving data dictionary information	71
Using DQL	72
Using the API	72
Manipulating object types	73
Creating new object types	73
Altering object types	74
Dropping object types	74
Manipulating objects	74
Methods and DQL	74
Permissions and privileges	75
Destroying objects	75
Changing an object's object type	76
Chapter 4 Security Services	79
Security overview	79
Standard security features	79
Trusted Content Services security features	81
Users and groups	82
Users	82
Groups	83
Dynamic groups	84
Mixing dynamic and non-dynamic groups in group memberships	84
User authentication	84
Password encryption	85
Application-level control of SysObjects	86
Privileges and permissions	87
User privileges	87
Basic user privileges	87
Extended user privileges	88
Object-level permissions	88
Table permits	90
Folder security	91
ACLs	91
ACL entries	92
Categories of ACLs	92

	Template ACLs.....	93
	Auditing and tracing	93
	Auditing	93
	Tracing	94
	Signature requirement support.....	95
	Electronic signatures.....	95
	What Addesignature does.....	97
	The default signature page template and signature method	98
	Default signature page template	98
	Default signature creation method	98
	How content is handled by default	99
	Audit trail entries	99
	What you can customize	100
	Verifying signatures.....	101
	General usage notes.....	101
	Digital signatures	101
	Signoff method usage	102
	Encrypted file store storage areas.....	103
	Digital shredding	104
Chapter 5	Server Internationalization	105
	What is internationalization?.....	105
	Content files and metadata.....	106
	Configuration requirements for internationalization	107
	Values set during installation	107
	The server config object.....	108
	Values set during sessions	108
	The api config object	108
	The session config object	109
	How values are set	109
	Where ASCII must be used	111
	User names, email addresses, and group names.....	111
	Lifecycles	112
	Docbasic	112
	Federations	112
	Object replication	113
	Other cross-repository operations	113
	Dump and load operations and the session code page.....	113
Chapter 6	Content Management Services	115
	Introducing SysObjects	115
	Documents.....	116
	Simple and virtual documents.....	116
	Content objects.....	116
	Page numbers.....	117
	Renditions.....	117
	Connecting source documents and renditions	117
	Primary content pages and renditions	118
	Translations	118
	Versioning.....	118
	Version labels	118
	Implicit version labels	119

Symbolic version labels	119
The CURRENT label	119
Uniqueness	120
Version trees	120
Branching	121
Removing versions	122
Changeable versions	123
Immutability	123
Effects of a Checkin or Branch method	124
Effects of a Freeze method	124
Effects of a retention policy	125
Attributes that remain changeable	125
Concurrent access control	126
Database-level locking	126
Repository-level locking	126
Optimistic locking	127
Document retention and deletion	128
Retention policies	128
Storage-based retention periods	129
If a retention policy and storage-based retention apply	129
Deleting documents under retention	130
Privileged deletions	130
Forced deletions	130
Deleting old versions and unneeded renditions	131
Documents and lifecycles	131
Creating SysObjects	132
Creating the object	132
Setting the object's attributes	132
owner_name attribute	133
keywords attribute	133
default_folder attribute	133
a_full_text attribute	134
language_code attribute	134
Adding content	135
The first content file	135
Additional primary content files	135
Macintosh files	136
Renditions	136
Assigning content to storage	136
Content assignment policies	137
Default storage allocation	137
Explicitly assigning a storage area	137
Assigning an ACL	138
Setting content attributes and metadata for content-addressed storage	138
Saving the new object	140
Modifying SysObjects	140
Getting a document from the repository	141
Modifying single-valued attributes	142
Setting the a_full_text attribute	142
Setting the a_content_type attribute	142
Modifying repeating attributes	142
Adding values	142
Replacing a value	143
Removing a value	143
Performance tip for repeating attributes	143

Adding content	143
Adding additional primary content	145
Replacing an existing content file.....	145
Removing content from a document	145
Sharing a content file	145
Writing changes to the repository	146
Checkin and Checkinapp methods.....	146
Save and SaveLock methods.....	146
Assigning ACLs	147
Assigning a default ACL	147
Assigning an existing non-default ACL.....	148
Generating custom ACLs	148
Granting permissions to a new object without assigning an ACL.....	149
Using grantPermit to modify the ACL assigned to a new object	149
Using grantPermit when no default ACL is assigned	149
Using grantPermit or revokePermit to modify the current ACL	149
Removing permissions.....	150
Removing permissions to a single document	151
Removing permissions to all documents	151
Replacing an ACL.....	151
Managing content across repositories.....	151
Reference links	152
Mirror objects.....	152
Reference objects	153
Replicas	153
Managing translations	153
Working with annotations.....	154
Creating annotations.....	155
The annotation file.....	155
Attaching the annotation to the document.....	156
Detaching annotations from a document	156
Deleting annotations from the repository	156
Object operations and annotations	156
Save, Check In, and Saveasnew	157
Destroy	157
Object replication	157
User-defined relationships	157
Creating a relationship between two objects	158
Destroying a relationship between objects	159
Relationships and object operations	159
Checkin, Saveasnew, and Branch	159
Save.....	160
Destroy	160
Object replication	160
Chapter 7 Virtual Documents	161
Introducing virtual documents	162
Virtual document architecture	162
Object types	162
SysObject attributes	163
Component ordering	163
Versioning.....	163
Referential integrity and freezing.....	164
Conditional assembly	164
snapshots.....	164

Virtual documents and content files	165
XML support.....	165
Virtual documents and retention policies	165
Early and late binding.....	166
Early binding	166
Absolute links	166
Symbolic links.....	166
Late binding.....	167
Defining component assembly behavior.....	167
use_node_ver_label	167
follow_assembly.....	169
Defining copy behavior.....	169
Creating virtual documents.....	169
Obtaining the object.....	170
Setting the r_is_virtual_doc attribute	170
Adding components	170
Saving the changes	171
Modifying virtual documents.....	171
Adding components	171
Removing components	172
Changing the component order	172
Modifying assembly behavior	172
Modifying copy behavior.....	172
Assembling a virtual document	173
Selecting components	173
SELECT processing.....	174
Snapshots	175
Creating snapshots	176
Modifying snapshots	177
Adding new assembly objects.....	177
Deleting an assembly object	178
Changing an assembly object.....	178
Deleting a snapshot	178
Freezing virtual documents and snapshots.....	178
Freezing a document	179
Unfreezing a document.....	179
Obtaining information about virtual documents	180
Querying virtual documents	180
Obtaining a path to a particular component.....	181
The path_name attribute	181
Vdmpath and Vdmpathdql methods	182
Chapter 8 Renditions	183
What a rendition is	183
Converter support	184
Media Transformation Services renditions.....	184
Creating renditions.....	184
Rendition formats.....	185
Rendition characteristics	186
Resolution characteristics	186
Encapsulation characteristics.....	187
Transformation loss characteristics.....	187
Reading and composing a full format specification	189
Adding and removing renditions.....	190

Determining the keep_flag setting	190
Supported conversions on Windows platforms.....	191
Supported conversions on UNIX platforms	191
PBM image converters	192
Miscellaneous converters	193
Implementing an alternate converter.....	194
Chapter 9 Workflows	197
Introducing workflows	197
Implementation.....	198
Template workflows	199
Business Process Manager and Workflow Manager	200
Workflow definitions	201
Process definitions	201
Activity types in a process definition	201
Multiple use.....	201
How activities are referenced in workflows	202
Links	202
Activity definitions.....	202
Manual and automatic activities	203
Activity priorities	203
Use of the priority defined in the process definition	203
Use of the work queue priority values.....	204
Delegation and extension	204
Delegation	204
Extension.....	205
Repeatable activities	205
Performer choices.....	205
Performer categories.....	206
Categories for manual activities.....	210
Categories for automatic activities	210
Defining the actual performer.....	211
At workflow initiation	211
At activity initiation.....	211
At the completion of a previous activity	212
Choosing the same performer set for multiple manual activities	212
Using aliases as performer names	212
Constraints on aliases as performer names	212
Alias resolution in workflows.....	213
Task subjects	213
Starting conditions.....	215
Port and package definitions	215
Port definitions.....	215
Package definitions.....	216
Empty packages	217
Scope of a package definition	217
Required Skill Levels for Packages	217
Package compatibility	218
Transition behavior.....	219
Number of completed tasks as transition trigger.....	219
Transition types.....	219
Limiting output choices in manual transitions	220
Setting preferences for output port use in manual transitions	220
Route cases for automatic transitions	221
Implementation without an XPath expression.....	222
Implementation with an XPath expression	222

Compatibility of the implementations	223
How the associated ports are recorded	223
Warning and suspend timers	223
Warning timers	224
Suspend timers	225
Package control	225
Process and activity definition states	226
Validation and installation	226
Validating process and activity definitions	226
What validation checks	227
Validating port connectability	227
Installing process and activity definitions	228
Architecture of workflow execution	228
Workflow objects	228
Activity instances	229
Work item objects	229
Work items and queue items	229
How manual activity work items are handled	230
Resetting priority values	230
Completing work items	231
Signing off manual work items	231
Package objects	232
Package notes	232
Attachments	233
The workflow supervisor	233
The workflow agent	234
The enable_workitem_mgmt key	234
Instance states	235
Workflow states	235
Activity instance states	236
Work item states	237
Starting a workflow	238
How execution proceeds	238
The workflow starts	241
Activity execution starts	241
Evaluating the starting condition	242
Package consolidation	242
Resolving performers and generating work items	243
Manual activities	243
Automatic activities	243
Resolving aliases	244
Executing automatic activities	244
Assigning an activity for execution	244
Executing an activity's program	245
Evaluating the activity's completion	246
When the activity is complete	249
How activities accept packages	249
How activity timers work	251
Pre-timer instantiation	252
Post-timer instantiation	252
Suspend timer instantiation	252
Activating pre- and post-timers	252
Activating suspend timers	253
Compatibility with pre-5.3 timers	253
User time and cost reporting	253

Reporting on completed workflows	254
Changing workflow, activity instance, and work item states	255
Halting a workflow.....	255
Activity instances in halted workflows.....	255
Halting an activity instance	255
Resuming a halted workflow or activity	256
Restarting a halted workflow or failed activity.....	256
Aborting a workflow	256
Pausing and resuming work items	257
Modifying a workflow definition.....	257
Changing process definitions	257
Overwriting a process definition.....	257
Versioning process definitions	258
Reinstalling after making changes.....	258
Changing activity definitions.....	259
Overwriting an activity definition.....	259
Adding ports and package definitions.....	259
Removing ports and package information	259
Saving the changes	260
Destroying process and activity definitions	260
Distributed workflow	260
Distributed notification	261
Remote object routing.....	261
Chapter 10 Lifecycles	263
Introducing lifecycles.....	263
Normal and exception states	264
Types of objects that may be attached to lifecycles.....	265
Entry criteria, actions on entry, and post-entry actions	266
Default lifecycles for object types	266
Lifecycle definitions	267
Draft, validated, and installed definitions.....	267
Validation	267
Installation.....	268
How lifecycles work	268
General overview	268
Attaching objects	269
Movement between states	270
Promotions	270
Batch promotions	270
Demotions	270
Suspensions	271
Resumptions	271
Scheduled transitions.....	272
Internal supporting methods.....	273
How state changes work	274
Object permissions and lifecycles.....	275
Integrating lifecycles and applications	276
Lifecycles, alias sets, and aliases	276
State extensions	276
State types.....	277
Designing a lifecycle	278
Lifecycle state definitions	278
Attachability	279
Return to base state setting.....	280

Demoting from a state.....	281
Allowing scheduled transitions	281
Entry criteria definitions	282
Java programs as entry criteria	282
Docbasic programs as entry criteria	283
Boolean expressions as entry criteria.....	284
Actions on entry definitions.....	285
Using system-defined actions on entry.....	285
Java programs as actions on entry.....	286
Docbasic programs as actions on entry.....	286
Post-entry action definitions.....	287
Java programs as post-entry actions.....	288
Docbasic programs as post-entry actions.....	288
Sample implementations of Java interfaces	289
Including electronic signature requirements.....	291
Using aliases in actions	291
Creating lifecycles	292
Basic procedure.....	292
Testing and debugging a lifecycle	292
Testing a lifecycle	293
Debugging a lifecycle.....	293
Java-based lifecycles	293
Docbasic-based lifecycles	293
Log file location.....	294
Adding state extensions	294
Creating extension objects.....	294
Implementing a custom validation program.....	295
Java validation program requirements	296
Docbasic validation program requirements	296
Adding the custom program to the lifecycle definition.....	296
Modifying lifecycles.....	297
Possible changes.....	297
Uninstall and installation notifications	297
Getting information about lifecycles	298
Lifecycle information.....	298
Object information	298
Deleting a lifecycle.....	299
Chapter 11 Tasks, Events, and Inboxes	301
Introducing tasks and events.....	301
Introducing Inboxes.....	302
What an Inbox contains.....	302
Accessing an Inbox	302
dmi_queue_item objects.....	303
Determining Inbox content.....	303
GET_INBOX administration method.....	304
Getevents method.....	304
The dm_queue view	304
Manual queuing and dequeuing	304
Queuing items.....	305
Dequeuing an Inbox item.....	305
Signing off tasks	305
Registering and unregistering for event notifications	306
Registering for events	306

	Removing a registration	307
	Querying for registration information	307
Appendix A	Aliases	309
	Introducing aliases	309
	Internal implementation	310
	Defining aliases	310
	Alias scopes	311
	Workflow alias scopes.....	311
	Non-workflow alias scopes	312
	Determining the lifecycle scope for SysObjects	313
	Resolving aliases in SysObjects	313
	Resolving aliases in template ACLs.....	314
	Resolving aliases in Link and Unlink methods.....	314
	Resolving aliases in workflows	315
	Resolving aliases during workflow startup.....	315
	Resolving aliases during activity startup	316
	The default resolution algorithm	316
	The package resolution algorithm.....	316
	The user resolution algorithm.....	317
	When a match is found	317
	Resolution errors	317
Appendix B	Writing Distributed Applications	319
	Client Library (DMCL) support for distributed applications	319
	Subconnections	320
	Method scoping.....	320
	Reference links	321
	Valid object types for reference links	321
	Reference link binding	321
	Reference link storage.....	322
	Type-specific behavior of reference links	322
	Indirect references	322
	Replicas	322
	Security	323
	Reference links	323
	Replicas	324
	Distributed messaging	324
	Information for client applications	325
	Query and method guidelines	325
	Executing Smart Lists or stored queries.....	326
	Executing scripts and procedures	326
	Annotations	326
	Object and type caches	326
	Defining a binding label for reference links	326
	Operations on reference links	327
	Operations on reference link source objects	327
	Operations on replicas	327
	Retrieving content	328
	Distributed operations	328
	Explicit transactions.....	329
	Reference link refreshes	329
	The dm_DistOperations job.....	329

Method access	329
The callback attributes	330
The new connection callback attributes	331
The connect success callback attributes	331
The connect failure callback attributes.....	331

List of Figures

Figure 3-1.	Sample hierarchy.....	78
Figure 6-1.	A version tree with branches.....	121
Figure 6-2.	Before and after pruning.....	123
Figure 7-1.	A sample virtual document showing node bindings.....	168
Figure 9-1.	Components of a workflow.....	199
Figure 9-2.	Workflow state diagram.....	235
Figure 9-3.	Activity instance state diagram.....	236
Figure 9-4.	Work item states.....	237
Figure 9-5.	Execution flow in a workflow.....	240
Figure 9-6.	Behavior if $r_complete_witem$ equals $r_total_workitem$	247
Figure 9-7.	Behavior if $r_complete_witem < r_total_workitem$	248
Figure 9-8.	Changes to a package during port transition.....	250
Figure 9-9.	Package arrival.....	251
Figure 10-1.	A lifecycle definition with exception states.....	264
Figure 10-2.	Simple lifecycle definition.....	269

List of Tables

Table 3–1.	Repeating attributes table.....	61
Table 3–2.	Row determination in a repeating attributes table.....	62
Table 4–1.	User privilege names and levels.....	87
Table 4–2.	Extended user privileges	88
Table 4–3.	Base object-level permissions.....	89
Table 4–4.	Extended object-level permissions	89
Table 4–5.	Table permits	90
Table 4–6.	Generic string attribute use for dm_addesignature events	99
Table 6–1.	Method choices for adding content	144
Table 8–1.	Resolution names	186
Table 8–2.	Encapsulation names	187
Table 8–3.	Sample transformations and their loss values	188
Table 8–4.	Supported input and output formats for automatic conversion.....	191
Table 8–5.	Acceptable input formats for PBMPLUS converters	192
Table 8–6.	Output formats of the PBMPLUS converters.....	193
Table 8–7.	Acceptable iInput formats for UNIX conversion utilities.....	194
Table 8–8.	Output formats of UNIX vonversion utilities	194
Table 9–1.	Performer categories for activities.....	206
Table 9–2.	Mode parameter values.....	246
Table 10–1.	Lifecycle methods.....	273
Table B–1.	Attributes implementing distributed messaging in dmi_queue_item.....	324

Preface

This manual describes the fundamental features and behaviors of Documentum Content Server. It provides an overview of the server and then discusses the basic features of the server in detail.

Intended audience

This manual is written for system and repository administrators, application programmers, and any other user who wishes to obtain a basic understanding of the services and behavior of Documentum Content Server. The manual assumes the reader has an understanding of relational databases, object-oriented programming, and SQL (Structured Query Language).

Conventions

This manual uses the following conventions in the syntax descriptions and examples.

Syntax conventions

Convention	Identifies
<i>italics</i>	A variable for which you must provide a value.
[] square brackets	An optional argument that may be included only once
{ } curly braces	An optional argument that may be included multiple times

Terminology changes

Two common terms are changed in 5.3 and later documentation:

- Docbases are now called repositories, except where the term “docbase” is used in the name of an object or attribute (for example, docbase config object).
- DocBrokers are now called connection brokers.

Revision history

The following changes have been made to this document.

Revision history

Revision date	Description
September 2005	Initial publication

Introducing Content Server

This chapter introduces Documentum Content Server. It includes the following topics:

- [Content Server's role in the product suite, page 21](#)
- [Content management services, page 21](#)
- [Process management features, page 24](#)
- [Security features, page 25](#)
- [Distributed services, page 26](#)
- [Content Server architecture, page 26](#)
- [Internationalization, page 27](#)
- [Communicating with Content Server, page 27](#)

Content Server's role in the product suite

Content Server is the foundation of Documentum's content management system. Content Server is the core functionality that allows users to create, capture, manage, deliver, and archive enterprise content.

The functionality and features of Content Server provide content and process management services, security for the content and metadata in the repository, and distributed services.

Content management services

Content Server provides full set of content management services, including library services (check in and check out), version control, and archiving options.

Storage and retrieval

Documentum provides a single repository for content and metadata. Content Server uses an extensible object-oriented model to store content and metadata in the repository. Everything in a repository is stored as objects. The metadata for each object is stored in tables in the underlying RDBMS. Content files associated with an object can be stored in file systems, in the underlying RDBMS, in content-addressed storage systems, or on external storage devices. [Chapter 3, The Data Model](#), provides a detailed description of the repository data model.

Content files can be any of a wide variety of formats. If you install Documentum Media Transformation Services in addition to Content Server, your system can handle digital media content such as audio and video files and thumbnail renditions.

To retrieve metadata, you use the Document Query Language (DQL). DQL is a superset of the ANSI SQL that provides a single, unified query language for all the objects managed by Content Server. It extends SQL by providing the ability to query:

- The repository cabinet and folder hierarchy
- Metadata and contents of documents in the repository
- A virtual document's hierarchy
- Process management objects such as inboxes, lifecycles, and workflows

Calls to retrieve content files are handled by Content Server, Thumbnail Server (provided with Documentum Media Transformation Services), or a streaming server, depending on the content's format. (The streaming server must be purchased separately from a third-party vendor.)

For more information about DQL, refer to [Chapter 4, Using DQL](#). For information about how content files are handled, refer to *Content Server Administrator's Guide*. For information about Documentum Media Transformation Services, refer to *Administering Documentum Media Transformation Services*.

Versioning

One of the most important functions of a content management system is controlling, managing, and tracking multiple versions of the same document. Content Server has a powerful set of automatic versioning capabilities to perform those functions. At the heart of its version control capabilities is the concept of version labels. Each document in the repository has an implicit label, assigned by the server, and symbolic labels, typically assigned by the user. Content Server uses these labels to manage multiple versions of the same document.

For more about versioning, refer to [Versioning, page 118](#).

Data dictionary

The data dictionary stores information in the repository about object types and attributes. The information can be used by client applications to apply business rules or provide assistance for users. The data dictionary supports multiple locales, so you can localize much of the information for the ease of your users.

When Content Server is installed, a default set of data dictionary information is set up. You can modify this set easily with a user-defined data dictionary population script. You can also add data dictionary information for Documentum or user-defined types with the DQL CREATE TYPE and ALTER TYPE statements.

[The data dictionary, page 64](#), describes the data dictionary in more detail and the information you can store in it. For information about populating the data dictionary, refer the *Content Server Administrator's Guide*.

Assembly and publishing

A feature of both content management and process management services, virtual documents are a way to link individual documents into one larger document.

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

You can assemble any or all of a virtual document's contained documents for publishing or perusal. You can integrate the assembly and publishing services with popular commercial word processors and publishing tools. The assembly can be dynamically controlled by business rules and data stored in the repository.

For more information about virtual documents and assembling them, refer to [Chapter 7, Virtual Documents](#).

Collaborative Services

Collaborative Services is an optional license that enables support for the Collaborative Editions of EMC Documentum clients. The features supported by Collaborative Services and provided through the Collaborative Editions are rooms, discussions, contextual folders, and notes:

- Rooms are secured areas within a repository (based on folders architecturally) with access restrictions and a defined membership. Rooms provide users with a secure

virtual “workplace” by allowing the room’s members to restrict access to objects in the room to the room’s membership.

- Discussions are online comment threads that enable informal or spontaneous collaboration on objects.
- Contextual folders allow users to add descriptions and discussions to folders. This provides users with the ability to capture and express the work-oriented context of a folder hierarchy.
- Notes are simple documents that have built-in discussions and may contain rich text content. Using notes avoids the overhead of running an application for text-based collaboration.

Content Server provides the underlying architectural support for the features of Collaborative Services. The features are fully exposed in EMC Documentum Webtop. For more information about these features and how to use them, refer to the Webtop documentation.

Retention Policy Services

Retention Policy Services is an optional product supported by Content Server. This product, if installed, allows you to manage a SysObject’s retention in the repository as a formal, defined set of phases, with a formal disposition phase at the end. This product is distributed as a DocApp and if installed, is used through Documentum Administrator. The Retention Policy Services product may not be customized. You must use the product as it is delivered.

Process management features

The process management features of Content Server are robust features that allow you to enforce business rules and policies while users create and manipulate documents and other SysObjects. The primary process management features of Content Server are workflows and lifecycles.

Workflows

Documentum’s workflow model allows you to easily develop process and event-oriented applications for document management. The model supports both production and ad hoc workflows.

You can define workflows for individual documents, folders containing a group of documents, and virtual documents. A workflow's definition can include simple or complex task sequences (including those with dependencies). Users with appropriate permissions can modify in-progress workflows. Workflow and event notifications are automatically issued through standard electronic mail systems while documents remain under secure server control.

Workflow definitions are stored in the repository, allowing you to start multiple workflows based on one workflow definition.

For details about workflow, refer to [Chapter 9, Workflows](#).

Life cycles

Many documents within an enterprise have a recognizable life cycle. A document is created, often through a defined process of authoring and review, and then is used and ultimately superseded or discarded.

Content Server's life cycle management services let you automate the stages in a document's life. A document's life cycle is defined as a lifecycle and implemented internally as a `dm_policy` object. The stages in a life cycle are defined in the policy object. For each stage, you can define prerequisites to be met and actions to be performed before an object can move into the stage.

For details about lifecycles, refer to [Chapter 10, Lifecycles](#).

Security features

Content Server supports a strong set of security options that provide security for the content and metadata in your repository and accountability for operations.

For a complete overview of all options, refer to [Chapter 4, Security Services](#).

Repository security

A repository's security setting can be either ACL or none. None turns off repository security. ACL turns on a security model based on Access Control Lists (ACLs). Repositories are created and configured with a default security setting of ACL.

In the ACL model, every object that is a `SysObject` or `SysObject` subtype 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 which operations they can perform.

When repository security is on, Content Server enforces seven levels of base object-level permissions and five extended object-level permissions using ACLs.

Content Server also provides five levels of user privileges, three extended user privileges, folder security, and basic support for client-application roles and application-controlled SysObjects.

Accountability

Accountability is an important part of many business processes. Content Server has robust auditing and tracing facilities. Auditing provides a record of all audited operations that is stored in the repository. Tracing provides a record that you can use to troubleshoot problems when they occur.

Content Server also supports electronic signatures. Content Server has the ability to store electronic sign-off information. In your custom applications, you can require users to sign off a document before passing the document to the next activity in a workflow or before moving the document forward in its life cycle. The sign-off information is stored in the repository.

Distributed services

A Documentum installation can have multiple repositories. There are a variety of ways to configure a site with multiple repositories. Content Server provides built-in, automatic support for all the configurations. For a complete description of the features supporting distributed services, refer to the *Documentum Distributed Configuration Guide*.

Content Server architecture

Content Server is written entirely in C++ and is implemented on the Windows platform and a variety of Unix platforms.

Internationalization

Content Server uses the UTF-8 code page and supports clients in a variety of code pages. For a summary of Content Server's internationalization features and behaviors, refer to [Chapter 5, Server Internationalization](#).

Communicating with Content Server

Documentum provides a full suite of products to give users access to Content Server.

Documentum client applications

Documentum provides Web-based and desktop clients. For information about these clients, refer to the *Documentum Product Catalog*.

Custom applications

You can write your own custom applications. Content Server supports all the Documentum Application Programming Interfaces. The primary APIs are the Documentum Foundation Classes (DFC) and the DMCL or client library. These two APIs give user applications full access to Content Server's features.

The Documentum Foundation Classes (DFC) are a set of Java interfaces that you can use to communicate directly to the DMCL without the need for an intervening protocol such as DDE. Applications written in Java, Visual Basic (through OLE COM), C++ (through OLE COM), and Docbasic can use the DFC.

For ease of development, Documentum provides a Web-based and a desktop development environment. You can develop custom applications (called DocApps) that you can deploy on the Web or desktop. You can also customize components of the Documentum client applications.

To learn more, refer to *System Development Guide*.

Interactive utilities

To interactively communicate with Content Server, Documentum provides Documentum Administrator and two utilities, IAPI and IDQL.

Documentum Administrator is the Web-based system administration tool. Using Documentum Administrator, you can perform all the administrative tasks for a single installation or a distributed enterprise from one location.

IAPI and IDQL are interactive utilities that let you execute API methods and DQL statements directly. These utilities are primarily useful as testing arenas for methods and statements that you may want to put in an application. They are also useful when you want to execute a quick ad hoc query against the repository.

For more information about Documentum Administrator, IAPI, and IDQL, refer to the *Content Server Administrator's Guide*.

Session and Transaction Management

This chapter describes how sessions and transactions are managed in Content Server. It includes the following topics:

- [Introducing repository sessions, page 29](#), which describes the basic characteristics of a repository session
- [Role of the connection broker, page 33](#), which describes how connection brokers work with clients and servers
- [Specifying the connection broker at runtime, page 33](#), which describes how to identify a connection broker at runtime in an application
- [Requesting a native or secure connection, page 34](#), which describes the connection options for a client connecting to a server with a trusted server license.
- [Using connection pooling, page 34](#), which describes how connection pooling works
- [Login tickets, page 35](#), which describes login tickets and how they work
- [Application access control tokens, page 39](#), which describes application access control tokens and how they work
- [Defining the repository scope, page 43](#), which defines repository scope and how it is defined in an application
- [Concurrent sessions , page 44](#), which describes the constraints on concurrent users for a server
- [Persistent client caches, page 45](#), which describes persistent client caches and how they are implemented
- [Transaction management, page 52](#), which describes internal and explicit transactions and their characteristics

Introducing repository sessions

A repository session is a client connection to a repository. Repository sessions are opened when end users or applications establish a connection to a server. Each repository

session has a unique session identifier, which is returned by the Connect method that establishes the server connection.

Before a user or application can open a repository session, the client library must be initialized. If the session is opened through a Documentum client, the client library is initialized automatically, by the Documentum client. An application must initialize the client library explicitly.

Initializing the client library opens a virtual session, called an API session, and creates an api config object. An api config object is a non-persistent object that defines the configuration of the API session. API sessions are not generally visible to end users, and they have no session identifiers. However, you can use the alias apisession or simply a in some method calls to refer to the API session. For example, you can use either alias in a Getdochasemap method.

If you are using DFC, an API session is started by instantiating the IDfClient class. If your application is using the DMCL API, an API session is started by calling dmAPIInit.

During any single API session, a client can open multiple repository sessions, with the same or different repositories.

Session configuration

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

A session's configuration is defined by a session config object, a non-persistent object. A session config object is constructed from values taken from the api config object, the server config object (a persistent object that defines a Content Server's configuration), and a connection config object (a non-persistent object that defines a particular repository connection).

Most of the values found in the api config object are taken from the dmcl.ini file used by the client. The dmcl.ini file is a client initialization file.

For information about how the configuration objects are used and about the dmcl.ini file, refer to the *Content Server Administrator's Guide*. For a listing of the attributes of the config object, refer to the individual descriptions in the *EMC Documentum Object Reference Manual*.

Primary sessions and subconnections

There are two kinds of repository sessions: primary sessions and subconnections.

Primary sessions

Repository sessions established by issuing a Connect method are called primary sessions. All primary sessions have session identifiers in the format S_n where n is an integer equal to or greater than zero.

Users can open multiple primary sessions with one or more repositories during an api session. The number of primary sessions that a user can establish is controlled by the `max_session_count` key in the `dmcl.ini` file. This key is set to 10 by default and can be reset.

To open an additional primary session with the same repository for the same user without specifying the user's password, use a login ticket. For information about login tickets, refer to [Login tickets, page 35](#).

Because some repositories have more than one Content Server and the servers are often running on different host machines, Connect methods let you be as specific as you like when requesting the server 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. For details, refer to the Javadocs or to [Connect, page 155](#), of the *Content Server API Reference Manual*.

Subconnections

A subconnection is a connection to a remote repository session opened within the context of an existing primary repository session. When an application or user performs an action on a remote object or replica, the client DMCL automatically opens a subconnection to the remote repository if one is needed. Occasionally, an application may need to open a subconnection explicitly. In these instances, use a `Getconnection` method.

The session identifier of a subconnection has the format S_nC_x . S_n identifies the primary session that opened the subconnection. C_x identifies the subconnection number.

The maximum number of subconnections that you can open under a primary session is determined by the `max_connection_per_session` key in the `dmcl.ini` file. The key is set to 30 when you install Content Server and can be reset. Refer to [Session subconnections, page 163](#), of the *Content Server Administrator's Guide* for information about resetting `dmcl.ini` keys.)

Inactive repository sessions

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

If a session is started with single-use login ticket and that session times out, the session cannot be automatically restarted by default because the login ticket cannot be re-used. To avoid this problem, an application can use `resetPassword`, an `IDfSession` method. This method allows an application to provide either the user's actual password 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
- Execute `resetPassword` to replace the single-use ticket with the user's actual password

Performing either option will ensure that the user is reconnected automatically if his or her session times out.

Restricted sessions

If a user connects with an operating system password that has expired, the system opens a restricted session for the user. The only operation allowed in a restricted session is changing the user's password. Applications can determine whether the session they begin is a restricted session by examining the value of the computed attribute `_is_restricted_session`. This attribute is T (TRUE) if the session is a restricted session.

Closing repository sessions

A user or application's repository session is terminated when the user or application issues a `Disconnect` method or when another user assumes ownership of the session. If connection pooling is implemented, a `Disconnect` method only terminates the user's use of a session. The session itself is held for future use in the connection pool. If connection pooling is not implemented, a `Disconnect` method terminates the session also. ([Using connection pooling, page 34](#), describes how connection pooling is implemented.)

After all repository sessions are terminated, an application should also close the API session. In the DMCL API, this is done by calling `dmAPIDefinit`.

Role of the connection broker

A connection broker is a name server for the Documentum Content Server. When a Connect method is issued, the request goes to a connection broker identified in the client's dmcl.ini file. The connection broker returns the connection information for the repository or particular server identified in the Connect method.

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

Which connection brokers a client can communicate with is configured in the client's dmcl.ini file. You can define primary and backup connection brokers in a dmcl.ini file. Doing so ensures that users will rarely encounter a situation in which they cannot obtain a connection to a repository.

For more information about how servers, clients, and connection brokers interact, refer to the *Content Server Administrator's Guide*.

Specifying the connection broker at runtime

The connection broker or brokers that an application can communicate with are specified in the dmcl.ini file that the application uses. When an application begins a session, the connection broker specifications in the dmcl.ini file are copied into attributes of the apiconfig object, a non-persistent object that contains configuration information for the session.

Applications can set those attributes directly in the apiconfig object before requesting a connection. Setting them directly allows the application to use a connection broker that may not be included in the connection brokers specified in the dmcl.ini.

To use this feature, the application must set the following apiconfig attributes:

- primary_host
- primary_port
- primary_protocol
- primary_service
- primary_timeout

These attributes are set using the keyword apisession. For example:

```
IDfClient client = DfClient.getLocalClient();
IDfTypedObject apiconfig = client.getClientConfig();
apiconfig.setString("primary_host", "lapdog2");
```

or

```
dmAPISet ("set, apisession, apiconfig, primary_host", "lapdog2")
```

Set the attributes before issuing a Connect method.

Requesting a native or secure connection

Content Servers can provide a secure connection for client sessions. The secure connections are made using the secure socket layer (SSL) protocol.

To provide a secure connection, the server must be configured to listen on a secure port and the client must ask for a secure connection.

By default, all Content Servers are configured to support only native (non-SSL) connections. For instructions on resetting the secure connection default, refer to [Setting the secure connection mode, page 109](#), of the *Content Server Administrator's Guide*.

Similarly, all client sessions, by default, request a native connection. The default can be changed by resetting a dmcl.ini key or overridden by explicitly requesting a secure connection in the Connect method. For instructions on how to request a secure connection, refer to [Requesting a native or secure connection, page 168](#), of the *Content Server Administrator's Guide*.

The connection type requested by the client interacts with the connection type configured for the server to determine whether the Connect method succeeds and what type of connection is established. The interaction is described in [Table 2-41, page 161](#), of the *Content Server API Reference Manual*.

Using connection pooling

Connection pooling is an optional feature that allows a primary repository session to be recycled and used by more than one user. Connection pooling can be implemented at the DMCL level or at the application level.

At the DMCL level

Implementing connection pooling at the DMCL level provides performance benefits for applications that execute frequent connections and disconnections for multiple users. When connection pooling is enabled in the DMCL, primary repository sessions are not closed when a user disconnects. Instead, they are held in a connection pool created by the DMCL. When another user requests a connection to the repository, the user is authenticated and given the free connection. (If there is no free connection to the

repository, the system establishes a new connection and registers it with the connection pool.)

The system automatically resets any security and cache-related information as needed for the new user. It also resets the error message stack and rolls back any open transactions.

For instructions about enabling and configuring connection pooling, refer to [Connection pooling, page 170](#), of the *Content Server Administrator's Guide*.

At the application level

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

When connection pooling is simulated using an Assume method, there is no connection pool created or maintained. Instead, ownership of a primary repository session passes from one user to another by executing the Assume method within the application. (An application can, of course, create a connection pool in imitation of the DMCL-level functionality. If that is done, the application should still use Assume to pass the open connection to another user.)

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 details about using an Assume method, refer to the Javadocs or [Assume, page 106](#), of the *Content Server API Reference Manual*.

Connection pooling and subconnections

When a user disconnects or a new user assumes ownership of a primary repository session, all subconnections open in that session are closed.

Login tickets

A login ticket is an ASCII-encoded string that an application can use in place of a user's password when connecting to a repository. Login tickets can be used to establish a connection with the local or a remote repository. Though they must be created in a session, you can use an existing ticket to create a primary session, a subconnection, or another session with the primary session's repository.

By default, you can use a login ticket 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.

Ticket format

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 such information as when the ticket was created, the repository in which it was created, and who created the ticket. (For troubleshooting purposes, Content Server supports a method called `Dumploginticket` that returns the encoded values in readable text format. [Table 2-57, page 199](#), describes the values returned by the method.) The signature is generated using the login ticket key installed in the repository.

The login ticket key

The login ticket key (LTK) is a symmetric key installed in a repository when the repository is created. Each repository has one LTK. The LTK is stored in the `ticket_crypto_key` attribute of the docbase config object.

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 Content 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.

Content 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 DFC methods in the `IDfSession` interface. For information about executing these methods, refer to [Exporting and importing a login ticket key, page 449](#), in the *Content Server Administrator's Guide*.

It is also possible to reset a repository's LTK if needed. Resetting a key removes the old key and generates a new key for the repository. For instructions, refer to [Resetting a login ticket key, page 449](#), in the *Content Server Administrator's Guide*.

Login ticket scope

The scope of a login ticket defines which Content 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's issuing repository if:

- The 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

[Trusted repositories, page 37](#), describes how trust is determined between repositories.

Trusted repositories

In addition to using identical login ticket keys, repositories that accept login tickets from each other must also trust each other. All repositories run in either trusting or non-trusting mode. Whether a repository is running in trusting or non-trusting mode is defined in the `trust_by_default` attribute in the docbase config object.

If `trust_by_default` is set to T, then the repository is running in trusting mode and trusts all other repositories. In trusting mode, the repository accepts any global login ticket generated with an LTK that matches its LTK, regardless of the ticket's source repository. If the attribute is set to F, then the repository is running in non-trusting mode. A non-trusting repository accepts global login tickets generated with a matching LTK that come from repositories specifically named as trusted repositories. The list of trusted repository names is recorded a repository's `trusted_docbases` attribute in its docbase config object.

To illustrate, 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 from all three of the other repositories. RepositoryK has `trust_by_default` set to F. RepositoryK

also has two repositories listed in its `trusted_docbases` attribute: `RepositoryM` and `RepositoryN`. `RepositoryK` rejects login tickets from `RepositoryA` because `RepositoryA` is not in the list of trusted repositories. It accepts tickets from `RepositoryM` and `RepositoryN` because they are listed in the `trusted_docbases` attribute.

For instructions on setting up a repository's trust mode, refer to [Configuring a repository's trusted repositories, page 449](#), in the *Content Server Administrator's Guide*.

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` attribute. The maximum period is defined in the `max_login_ticket_timeout` attribute.

A validity period specified as an argument overrides the default defined in the server config object unless the argument value exceeds the configured maximum. If the method argument exceeds the maximum validity period, 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. If an application creates a login ticket while connected to that server and sets the ticket's validity period to 20 minutes, that value overrides the default, and the ticket is valid for 20 minutes. If the application attempts to set the ticket's 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.

For instructions on configuring the default and maximum validity periods in a repository, refer to [Configuring the default login ticket timeout, page 450](#), in the *Content Server Administrator's Guide*.

Note on host machine time and login tickets

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 is to allow 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 on which applications and repositories reside are set as closely as possible to the correct time.

Revoking tickets

You can set a cutoff date for login tickets on an individual repositories. If you set a cutoff date for a repository, the repository servers consider any login tickets generated prior to the specified date and time to be revoked, or invalid. 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` attribute of the repository's `doibase` 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's creation date.

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, he or she 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 instructions, refer to [Restricting a Superuser's use of global tickets, page 451](#), in *Content Server Administrator's Guide*.

Application access control tokens

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 Content 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 server to use AAC tokens, you can control:

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

How tokens work

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 may designate one server in 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. For instructions on enabling token use in a server, refer to [Enabling AAC token use by a server, page 452](#), in *Content Server Administrator's Guide*.

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's 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 and higher client libraries are capable of appending a token or machine identifier to a connection request. Configuring a client library to append a token is optional. For information about implementing this behavior, refer to [Enabling token retrieval by the client library, page 452](#), in the *Content Server Administrator's Guide*.

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. Is the connection request on behalf of the specified user or a user who is a member of the specified group? Or, was the request issued from the specified host? If the token restricts use to a particular application, does the connection request include a matching application ID? If the constraints are satisfied, the server allows the connection. If not, the server rejects the connection request.

Format of AAC tokens

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, Content Server supports a method called `Dumploginticket` that returns the encoded values in readable text format. [Table 2-58, page 199](#), describes the values returned by the method.) The signature is generated using the repository's login ticket key. For information about the login ticket key, refer to [The login ticket key, page 36](#).

Token scope

The scope of an application access control token identifies which Content 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 token's scope is a single repository, then the token is only accepted by Content Servers of that repository. The application using the token can send its connection request to any of the repository's 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 the other repositories. [Trusted repositories, page 37](#), describes how trust is determined between repositories.

Token generation

Application access control tokens can be generated at runtime or you can generate and store tokens for later retrieval by the client library.

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 the client library 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 a `dmcl.ini` file key. Token use is enabled by another `dmcl.ini` key. If use is enabled, a token can be retrieved and appended to a connection request by the client library when needed. For instructions on using `dmtkgen`, refer to [Generating tokens for storage, page 453](#), in the *Content Server Administrator's Guide*. For instructions on implementing token use, refer to [Enabling token retrieval by the client library, page 452](#), also in the *Administrator's Guide*.

Token expiration

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.)

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 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 for the client library on the host on which the method is executed. Token retrieval allows the client library to append a token retrieved from storage to the connect request. The token must be generated by the `dmtkgen` utility and must be a valid token for the connect request. For instructions on setting up and enabling token retrieval, refer to [Enabling token retrieval by the client library, page 452](#), in the *Content Server Administrator's Guide*.

Defining the repository scope

Repository scope identifies the repository in which a method performs its operations. When a method is issued, its instructions are executed by a Content Server against one or more objects in the repository associated with that server.

In a single-repository configuration, the repository scope is static. In a multi-repository configuration, such as a federation, the repository scope can change with each method issued. If your enterprise is using a multi-repository configuration, it is very important to make sure that each method in an application is issued against the intended repository.

Command line scoping

For many methods, the repository scope is determined by an argument in the command line that defines the scope implicitly. The scoping argument can be an object ID, a repository ID, or a repository name.

For example, in a Checkout method, you must include the object ID of the object you are checking in to the repository. An object contains the repository ID within itself. Consequently, when an application issues a Checkout method, it isn't necessary to direct the method to the appropriate repository because the object ID identifies the repository. The client DMCL reads the object ID and uses or obtains the appropriate repository subconnection.

Default scoping

Methods that do not take a scoping argument in the command line operate in the default scope unless they are explicitly directed to a different repository. For example, the methods that query the repository, such as Execquery, Close, or Retrieve, do not take scoping arguments. They operate in the default scope. The default scope is the repository defined in the `docbase_scope` attribute of the session config object.

There are two ways to direct such a method to a particular repository:

- Set the default scope to the repository before executing the method.
- Identify the appropriate subconnection in the method call.

To set the default scope, set the `docbase_scope` attribute of the session config object. The change is effective immediately.

To identify the subconnection in the method call, use the subconnection identifier as the session argument. With the exception of Connect methods, all methods take a session argument in their command line. This argument can identify a session or a subconnection.

To obtain the appropriate subconnection identifier, use the `Getconnection` method. The following example illustrates how to use `Getconnection` to obtain the identifier for a subconnection to the Engineering repository and use it with the `Query` method:

```
subconnectID=dmAPIGet("getconnection,s0,Engineering")
cmd_str="query," subconnectID ",select owner_name from dm_document"
qry_id=dmAPIGet(cmd_str)
```

If you use subconnection identifiers to redirect a method to a particular repository, be sure to issue `Getconnection` before each execution of the method to obtain a current subconnection identifier for the repository. Subconnection identifiers do not always persist for the life of the primary session. Because only a limited number of subconnections are allowed in each primary session, the system may close a subconnection that has not been active to allow a new subconnection to be created.

Note: While it is possible to use a subconnection identifier in a method that has a scoping argument in its command line, the scope defined by the subconnection identifier is ignored. The scope defined by the scoping argument overrides any scope defined by the subconnection identifier.

Concurrent sessions

There are limits placed on the number of repository connections that each Content Server can handle concurrently. The default is 100 connections. 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 Content Server, whether a primary connection or a subconnection, counts as one connection. In addition to the connections explicitly opened by a user or application, the DMCL may open connections to complete particular operations requested by the user.

Content Server returns an error if the maximum number of sessions defined in the `concurrent_sessions` key is exceeded. You may find it necessary to reset the `concurrent_sessions` key. Instructions for setting `server.ini` file keys are found in the *Content Server Administrator's Guide*.

Persistent client caches

Persistent client caches are caches of repository objects and query results managed and maintained across client sessions. Persistent client caches are implemented through the client DMCL and supported by Content Server. Using this feature provides performance benefits during session start up and when users or applications access cached objects and query results.

The ability to cache objects and query results is enabled by default for every repository and every client session. Documentum DesktopClient and Documentum Webtop both take advantage of this feature. Applications can take advantage of the feature through the API methods that support requests for persistent client caching—`Fetch` and `Query_cmd`.

Consistency between the repository and cached objects and query results is checked and maintained using a consistency check rule identified in the `Fetch` or `Query_cmd` method call that references them. A consistency checking rule can be applied to individual objects or query results or a rule can be defined for a set of cached data. [Consistency checking, page 49](#), describes how consistency checking is defined and implemented.

Note: The `Cachequery` method is supported for caching query results. However, due to consistency checking limitations inherent in the implementation of `Cachequery`, `Query_cmd` is the recommended method for caching query results.

For information about using the methods to implement persistent client caching, refer to their descriptions in the *Content Server API Reference Manual*. For information about how persistent client caching is enabled or disabled, refer to [Enabling and disabling persistent client caching, page 192](#), in the *Content Server Administrator's Guide*.

Object caches

The DMCL maintains an in-memory object cache for each repository session for the duration of the repository session. The cache stores a copy of every object fetched by the client during the session and copies of those objects fetched and persistently cached

in previous sessions. If the client requests persistent caching for a fetched object, the in-memory copy is marked for persistent caching, the object is written out to a file. The file is written after defined intervals and when the application terminates. (Refer to [Defining the persistent cache write interval, page 198](#), in the *Content Server Administrator's Guide* for information about the intervals.) The file is stored in the following directory:

```
root/object_caches/machine_name/repository_id/abbreviated_user_name
```

root is the value of the `local_path` key in the client's `dmcl.ini` file. The default is the current working directory.

The next time the user starts a session with the repository on the same machine, the DMCL loads the file back into memory. (For a detailed description of how the persistent object file is handled, refer to [Defining the persistent cache write interval, page 198](#), in the *Content Server Administrator's Guide*.)

Clients request persistent object caching by setting an argument in the `Fetch` method that fetches the object from the repository.

Type and data dictionary caches

In conjunction with the object cache, the DMCL maintains a type cache and a data dictionary cache. The type and data dictionary caches are global caches, shared by all sessions in a multi-threaded application.

When an object is fetched, the DMCL also fetches and caches in memory the object's object type and the associated data dictionary objects if they are not already in the cache. Type and data dictionary objects in the DMCL's in-memory caches are automatically persistently cached if persistent caching is enabled. They are stored in a file located in the following directory:

```
root/type_caches/machine_name/repository_id
```

root is the value in the `local_path` key in the client's `dmcl.ini` file. The default is the current working directory.

Query caches

Query results are only cached when persistent caching is requested and persistent caching is enabled. The results are cached in a file. They are not stored in memory. The file is stored on the client disk, with a randomly generated extension, in the following directory

On Windows:

```
\root\qrycache\machine_name\repository_id\user_name
```

On UNIX:

```
/root/qrycache/machine_name/repository_id/user_name
```

root is the value in the *local_path* key in the user's *dmcl.ini* file. The default is the current working directory.

The query cache files for each user consist of a *cache.map* file and files with randomly generated extensions. The *cache.map* file maps each cached query to the file that contains the results of the query (one of the files with the randomly generated extensions).

The queries are cached by user name because access permissions may generate different results for different users.

Note: The *cache.map* file and the cached results files are stored in ASCII format. They are accessible and readable through the operating system. If security is an issue, make sure that the directory in which they are stored is truly local to each client, not on a shared disk.

Clients can request query caching by setting an argument in the *Query_cmd* method that executes the query. Executing a *Cachequery* method also caches query results, but due to consistency checking limitations imposed by the *Cachequery* method, *Query_cmd* is the recommended way to cache query results.

Subconnections and persistent caching

In federated environments or any distributed environment that has multiple repositories, users can work in multiple repositories through one primary client session. The DMCL treats each of the subconnections to a different repository as a separate session for the purposes of persistent caching. For example, suppose JohnDoe opens a session with repository A and fetches a persistently cached document. Then, the user also fetches a persistently cached document from repository B. On termination, the DMCL writes two persistent object caches:

```
root/object_caches/machine_name/repositoryA/JohnDoe
```

and

```
root/object_caches/machine_name/repositoryB/JohnDoe
```

Similarly, if the user queries either repository and caches the results, the DMCL creates a query cache file specific to the queried repository and user.

Using persistent client caching in an application

Some Documentum clients, such as Desktop Client, use persistent client caching by default. If you want to use it in your applications, you must:

- Ensure that persistent client caching is enabled.

Persistent client caching is enabled at the repository and session levels by default. For information about the configuration keys that control persistent client caching and how to enable or disable the feature, refer to [Enabling and disabling persistent client caching, page 192](#), in the *Content Server Administrator's Guide*.

- Identify the objects or queries or both that you want to cache.

You identify the data to cache in the Fetch and Query_cmd methods in the application. [Identifying objects and queries for caching, page 48](#), describes how this is done.

- Define the consistency check rule for cached data.

A consistency check rule defines how often cached data is checked for consistency with the repository. The Fetch and Query_cmd methods support a variety of rule options through a method argument. [Consistency checking, page 49](#), describes the consistency checking rule options supported by Fetch and Query_cmd and how they are defined and applied for cached data. The Cachequery method does not have an argument that defines a consistency check for its results and, consequently, results obtained and cached using Cachequery are checked using a default mechanism. [Cachequery consistency checks, page 52](#), describes the mechanism.

Identifying objects and queries for caching

You identify objects to persistently cache by setting the persistent_cache argument in the Fetch methods that fetch the objects from the repository to T (TRUE). For instructions on using Fetch, refer to [Fetch, page 209](#), in the *Content Server API Reference Manual*.

You identify the queries whose results you want to cache by executing the queries using a Query_cmd method with the persistent_caching argument set to T or a Cachequery method.

Query_cmd is the preferred method. Because of the constraint on consistency checking in Cachequery (described in [Cachequery consistency checks, page 52](#)), it is recommended that you use Query_cmd, instead of Cachequery, to cache query results. However, Cachequery is supported for backwards compatibility. For instructions on using these methods, refer to [Query_cmd, page 334](#), and [Cachequery, page 133](#), in the *Content Server API Reference Manual*.

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 cached object or 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. A keyword or integer value is an explicit directive to the client DMCL that tells the DMCL how often to conduct the check. A cache config object identifies the data to be cached as part of a set of cached data managed by the consistency check rule defined in the cache config object. The data defined by a cache config object can be objects or queries or both. Using a cache config object to group cached data has the following benefits:

- More efficient validation of cached data
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 automatically validate cached data

Consistency checking is basically a two-part process:

1. The DMCL determines whether a consistency check is necessary.
2. The DMCL conducts the consistency check if needed.

[Determining if a consistency check is needed, page 50](#), describes how the DMCL determines whether a check is needed. [Conducting consistency checks , page 51](#), describes how the check is conducted.

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 queries, it is applied only to query results obtained using a `Query_cmd` method. The `Cachequery` method does not support an argument that allows you to define a consistency check rule for the results. Consequently, query results cached by a `Cachequery` method are only updated through a repository setting that flushes all caches. For details, refer to [Cachequery consistency checks, page 52](#).

Determining if a consistency check is needed

To determine whether a check is needed, the DMCL 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.

When a keyword or integer defines the rule

If the rule was specified as a keyword or an integer value, the DMCL 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 the DMCL 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 the DMCL 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 the DMCL 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.

When a cache config object defines the rule

If a consistency check rule names a cache config object, the DMCL 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 the DMCL with a timestamp that indicates when the information was obtained. The information includes

the `r_last_changed_date` and the `client_check_interval` attribute values of the cache config object.

When a method defines a consistency check rule by naming a cache config object, the DMCL first checks whether it has information about the cache config object in its memory. If not, it issues a `CHECK_CACHE_CONFIG` administration method to obtain the information. If it has information about the cache config object, the DMCL 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, the DMCL 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 the DMCL executes another `CHECK_CACHE_CONFIG` method to ask Content Server to provide current information about the cache config object. If the interval has not expired, the DMCL uses the information that it has in memory. (For information about how `CHECK_CACHE_CONFIG` behaves, refer to [CHECK_CACHE_CONFIG](#), page 179, in the *Content Server DQL Reference Manual*.)

After the DMCL has current information about the cache config object, it determines whether the cached data is valid. To determine that, the DMCL compares the timestamp on the cached data against the `r_last_changed_date` attribute 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.

Conducting consistency checks

To perform a consistency check on a cached object, the DMCL uses the `i_vstamp` attribute value of the object. If the DMCL 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 `vstamp` values are different, the DMCL refetches the object and resets the time stamp. If they are the same, the DMCL uses the cached copy.

The DMCL does not perform consistency checks on cached query results. If the cached results are out of date, Content Server re-executes the query and replaces the cached results with the newly generated results.

Type and data dictionary cache consistency checks

The type and data dictionary caches are checked for consistency with the repository only once, at session startup. They are not checked each time an instance of the type or data

dictionary information is accessed. Cached data dictionary objects are linked to the cached object types, so they are not checked individually, but along with the types.

The DMCL uses the `type_change_count` and `dd_change_count` attribute values from the `dmi_change_record` object to determine whether the cached types and data dictionary objects are consistent with the repository. If they are consistent, the cached types and data dictionary objects are loaded into memory. If they are not consistent, they are refetched from the repository.

The default consistency check rule

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 the DMCL checks the `i_vstamp` value of the in-memory object against the `i_vstamp` value of the object in the repository.

If a `Query_cmd` method that requests persistent caching does not include an explicit value for the argument defining a consistency check rule, the default consistency rule is `check_never`. This means that the DMCL uses the cached query results.

Cachequery consistency checks

The `Cachequery` method does not have an argument that lets you define a consistency check rule. Cached query results obtained by executing a `Cachequery` method are only updated if the entire set of cached query results (meaning those obtained by `Query_cmd` and `Cachequery`) are refreshed due to a change in the `client_pcache_change` attribute value in the `docbase` config object.

When a client session is started, the DMCL checks the cached value of the `client_pcaching_change` attribute against the repository. If the values are different, the DMCL flushes all the persistent caches, including the object caches and all query caches.

Note that the `client_pcaching_change` value must be changed in the `docbase` config object manually, by a superuser. Additionally, changing its value forces all persistent caches to be flushed, not just the query results obtained through `Cachequery`.

Transaction management

A transaction is a set of repository operations handled as an atomic unit. All operations in the transaction must succeed or none may succeed. A repository connection can

have only one open transaction at any particular time. A transaction is either internal or explicit.

An internal transaction is a transaction managed by Content 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, then 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 Begintran method. It is closed with either a COMMIT statement or Commit method, which saves the changes, or an ABORT statement or Abort method, which closes the transaction without saving the changes. An explicit transaction can include as many operations as desired. However, keep in mind that none of the changes made in an explicit transaction are committed until a COMMIT statement is issued. If an operation fails, the transaction is automatically aborted and all changes made prior to the failure are lost.

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.

Issuing a Begintran method or BEGIN TRAN statement opens an explicit 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 be 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 execute an Assemble method.

Because the Assemble method is the first step in the four-step process to create snapshots, it opens its own transaction. Consequently, you cannot issue the Assemble method when you have an explicit transaction open.

- You cannot use API methods in the transaction if you opened the transaction with the DQL BEGIN[TRAN] statement.

If you want to use methods in an explicit transaction, open the transaction with the Begintran method.

- You cannot execute dump and load operations inside an explicit transaction.
- You cannot issue a CREATE TYPE statement in an explicit transaction.

- With one exception, you cannot issue an ALTER TYPE statement in an explicit transaction. The exception is an ALTER TYPE that lengthens a string attribute.

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 don't occur.

To put a database lock on an object, use the Lock method. 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 attributes or content of the object. It isn't 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.

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 and drops any locks held by that connection and rolls back any changes made in that connection's transaction.

Handling deadlocks in internal transactions

Content 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 tries to read data from a table through an index when another connection is locking the data while it tries to update the index
- Two connections are waiting for locks being held by the each other.

When deadlock occurs, Content Server executes internal deadlock retry logic. The deadlock retry logic tries to execute the operations in the victim's 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.

Handling deadlocks in explicit transactions

Content Server's 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.

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

To test custom deadlock retry logic, Content Server provides an administration method called `SET_APIDEADLOCK`. This method plants a trigger on a particular API method. When the method executes, the server simulates a deadlock, setting the `_isdeadlocked` computed attribute and rolling back any changes made prior to the method's execution. Using `SET_APIDEADLOCK` allows you to test an application's deadlock retry logic in a development environment. For more information about this method, refer to [SET_APIDEADLOCK, page 304](#), of the *Content Server DQL Reference Manual*.

The Data Model

This chapter describes the data model used Content Server. It includes the following topics:

- [Introducing object types and objects, page 57](#), which briefly describes object types and their characteristics
- [Attributes, page 59](#), which introduces attributes
- [Repositories, page 60](#), which describes the structure of a repository
- [Registered tables, page 64](#), which introduces registered tables
- [The data dictionary, page 64](#), which describes what the data dictionary is, its purpose, and what it contains
- [Manipulating object types, page 73](#), which describes what operations can be performed on object types
- [Dropping object types, page 74](#), which describes dropping an object type.
- [Manipulating objects, page 74](#), which contains an overview of how individual objects can be manipulated
- [Destroying objects, page 75](#), which describes how to remove objects from the repository
- [Changing an object's object type, page 76](#), which describes how to change an object from one object type to another object type

Introducing object types and objects

An object type represents a class of objects. Documentum is an object-oriented system. All the 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.

Object types are like templates. When you create an object, you identify which type of object you want to create. Content Server then uses the type definition as a template to create the object.

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

Supertypes, subtypes, and inheritance

Most Documentum 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 attributes of the supertype. The subtype also has the attributes defined specifically for it. For example, the `dm_folder` type is a subtype of `dm_SysObject`. It has all the attributes 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`.

Persistence

Most of the object types in Content Server are persistent. That is, 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 simply to return the results of DQL statements. When the underlying RDBMS returns rows for a SELECT statement, Content 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.

Naming

The names of all system-defined types begin with the prefix `dm`, `dmi`, or `dmr`. The `dm` prefix represents object types that are commonly used and visible to users and applications. The `dmi` prefix represents object types that are used internally by

Documentum Content Server and client products. The dmr prefix represents object types that are generally read only.

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 file or files that make up a document or other SysObject's content are called its primary content files. The primary content files must all have the same format. To create a document that has primary content in a variety of formats, 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. For more information about virtual documents, refer to [Chapter 7, Virtual Documents](#).

Attributes

The attributes that make up a persistent object type's definition are persistent. Their values for individual objects are saved in the repository. The attribute values saved in the repository are called metadata.

An object type's persistent attributes include those that are defined for the type and those that the type inherits from its supertype. ([Supertypes, subtypes, and inheritance](#), page 58, explains supertypes and inheritance.)

In addition to the persistent attributes, many object types also have associated computed attributes. Computed attributes are non-persistent. Their values are computed at runtime when a user requests the attribute and lost when the user closes the session. ([Computed attributes](#), page 16, of the *EMC Documentum Object Reference Manual* lists the computed attributes.)

All attributes share some characteristics. They are all either single-valued or repeating. A single-valued attribute stores one value. A repeating attribute stores multiple values in an indexed list. All attributes have a datatype that determines what kind of values can be stored in the attribute. For example, an attribute with an integer datatype can only store whole numbers. All attributes are either read only or can be read and written.

Persistent attributes have an additional characteristic that defines whether they are global or local attributes. This characteristic is only significant if a repository participates in object replication or is part of a federation.

Object replication creates replica objects, copies of objects that have been replicated between repositories. When users change a global attribute in a replica, the change

actually affects the source object attribute. Content Server automatically refreshes all the replicas of the object containing the attribute. If a repository participates in a federation, changes to global attributes in users and groups are propagated to all member repositories if the change is made through the governing repository, using Documentum Administrator.

A local attribute is an attribute whose value can be different in each repository participating in the replication or federation. If a user changes a local attribute in a replica, the source object is not changed and neither are the other replicas.

Note: It is possible to configure four local attributes of the `dm_user` object to make them behave as global attributes. This is described in the instructions for creating global users in the *Documentum Distributed Configuration Guide*.

For complete information about the characteristics of persistent and computed attributes, refer to [Chapter 1, Object Basics](#), of the *EMC Documentum Object Reference Manual*.

Repositories

A repository is where persistent objects managed by Content Server are stored. A repository stores the object metadata and, sometimes, content files. A Documentum 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. ([Identifiers, page 30](#), in the *EMC Documentum Object Reference* contains information about the identifiers recognized by Content Server.)

Repositories consist of two sets of tables in the underlying RDBMS: object type tables and type index tables.

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 attributes for all instances of the object type. The other table stores the values for repeating attributes for all instances of the object type.

Single-valued attribute tables

The tables that store the values for single-valued attributes 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 attribute and each row represents one instance of the object type. The column values in the row represent the single-valued attribute values for that object.

Repeating attribute tables

The tables that store values for repeating attributes 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 attribute.

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

Table 3-1. Repeating attributes table

<code>r_object_id</code>	<code>ingredients</code>
...	4 eggs
...	1 lb cream cheese
...	2 t vanilla
...	1 c sugar
...	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 attributes, the number of rows in the `_r` table for each object is equal to the number of values in the repeating attribute that has the most values. The columns for repeating attributes having fewer values are filled in with `NULLs`.

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

Table 3-2. Row determination in a repeating attributes table

...	authors	ingredients	testers	ratings
...	yvonne	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 attributes that contain a smaller number of values with NULLs.

Even an object with no values assigned to any of its repeating attributes has at least one row in its type's `_r` table. The row contains a NULL value for each of the repeating attributes. If the object is a `SysObject` or `SysObject` subtype, then it has a minimum of two rows in its type's `_r` table because its `r_version_label` attribute has at least one value—its implicit version label. (Refer to [NULLs, default values, and DQL](#), page 340, for an expanded explanation of how NULLs are handled in Documentum.)

Defining the location and extents of object type tables

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

On some databases, you can change the defaults when you create the repository. (If this is possible in your environment, the instructions are found in your Documentum installation guide, *Installing Content Server*.) By setting `server.ini` 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 extents 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 extents allotted to categories of object types or to specific object types.

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` attribute for each `_s` object type table and one on `r_object_id` and `i_position` for each `_r` object type table. The indexes are used to enhance query performance. Indexes are represented in the repository by objects of type `dmi_index`. The indexes are managed by the RDBMS.

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

Note: Using `MAKE_INDEX` is recommended instead of creating indexes through the RDBMS server because Content Server uses the `dmi_index` table to determine which attributes are indexed.

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 platforms (Windows or UNIX, with Oracle, for example), you can define an alternate location for the indexes during repository creation. Information about doing that, if it is possible on your platform, is found in your *Installing Content Server* manual. After the indexes are created, you can move them manually using the `MOVE_INDEX` administration method.

When you create a custom index, you can define its location.

(The administration methods are available through Documentum Administrator, the `DQL EXECUTE` statement, or the `Apply` method.)

Content storage areas

The content files associated with SysObjects are part of a repository. With two exceptions, content files are stored in directories represented by storage area objects (subtypes of `dm_store`) in the repository and referenced by the content file's content object in the repository. The exceptions are content files stored in turbo storage or blob storage. These content files are stored directly in the repository. Content in turbo storage is stored in an attribute 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 a complete description of the storage implementation, refer to the *Content Server Administrator's Guide*.

Registered tables

Registered tables are RDBMS tables that are not part of the repository but are known to Content Server. They are created by the DQL REGISTER statement and automatically linked to the System cabinet in the repository. 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.

For information about the REGISTER statement, refer to [Register, page 108](#), of the *Content Server DQL Reference Manual*. For information about querying registered tables, refer to [Querying registered tables, page 349](#).

The data dictionary

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

Usage

Content Server stores and maintains the data dictionary information but only uses a small part—the default attribute 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 an attribute. 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 attributes and object types in the data dictionary. All of this information is available to client applications. (For a complete description of the types of information that you can store in the data dictionary, refer to [What the data dictionary can contain, page 66](#).)

Localization

The data dictionary is the mechanism you can use to localize Content 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 attributes in German and English. Then, applications can query for the user's current locale and display the appropriate labels on dialog boxes.

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

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

By default, when Content 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` attribute of the `dm_docbase_config` object.

Modifying the data dictionary

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

- Add additional locales from the set of default locales provided with Content Server or custom locales
- Modify the information in an installed locale by adding to it, deleting it, or changing it

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 modifying the data dictionary, refer to [Populating the data dictionary, page 593](#), of the *Content Server Administrator's Guide*.

Publishing the 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
- `dd attr info` objects
- `dd common info` objects

A `dd type info` object contains the information specific to the object type in a specific locale. A `dd attr info` object contains information specific to the attribute in a specific locale. A `dd common info` object contains the information that applies to both the attribute and type level across all locales for a given object type or attribute. For example, if a site has two locales, German and English 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 attribute—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 attribute because that object stores the information that is common across all locales.

Applications query the `dd common`, `dd type info`, and `dd attr info` objects to retrieve and use data dictionary information. For instructions, refer to [Retrieving data dictionary information, page 71](#).

For instructions on publishing the data dictionary, refer to [Publishing the data dictionary information, page 607](#), in the *Content Server Administrator's Guide*.

What the data dictionary can contain

This section describes the kinds of information that you can put in the data dictionary. For information about adding information to the data dictionary, refer to the *System Administrator's Guide*.

Constraints

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

You can define five kinds of constraints in the data dictionary:

- Unique key
- Primary key
- Foreign key
- Not null
- Check

Unique key

A unique key constraint identifies an attribute or combination of attributes for which every object of that type must have a unique value. The key can be one or more single-valued attributes or one or more repeating attributes. It cannot be a mixture of single-valued and repeating attributes. All the attributes in a unique key must be defined for the same object type.

You can include attributes that allow NULL values because NULL never matches any value, even another NULL. To satisfy a unique key defined by multiple nullable attributes, all the attribute values must be NULL or the set of values across the attributes, as defined in the key, must be unique.

For example, suppose there is a unique key defined for the object type `mydoc`. The key is defined on three attributes, all of which can contain NULLs. The attributes are: A, B, and C. There are four objects of the type with the following values for A, B, and C:

<i>Attribute A</i>	<i>Attribute B</i>	<i>Attribute C</i>
NULL	NULL	1
1	NULL	NULL
NULL	NULL	NULL
NULL	NULL	NULL

If you create another `mydoc` object that has either of the following sets of values for A, B, and C, the uniqueness constraint is violated because these two sets of values already exist in instances of the type:

<i>Attribute A</i>	<i>Attribute B</i>	<i>Attribute C</i>
NULL	NULL	1
1	NULL	NULL

You can define unique key constraints at either the object type level or the attribute level. If the key includes two or more participating attributes, you must define it at the type level. If the key is a single attribute, it is typically defined at the attribute level, although you can define it at the type level if you prefer.

Unique key constraints are inherited. Defining one for a type does not override any inherited by the type. Any defined for a type are applied to its subtypes also.

Primary key

A primary key constraint identifies a non-nullable attribute or combination of non-nullable attributes that must have a unique value for every object of the type.

An object type can have only one primary key constraint defined for it. A type may also have a primary key inherited from its supertype. The attributes in a primary key must be single-valued attributes that are defined for the same object type. Repeating attributes are not allowed.

You can define primary key constraints at either the object type level or the attribute level. If the key includes two or more participating attributes, you must define it at the type level. If the key is a single attribute, it is typically defined at the attribute level, although you can define it at the type level if you prefer.

Primary key constraints are inherited. Defining one for a type does not override any inherited by the type. Any defined for a type are applied to its subtypes also.

Foreign key

A foreign key constraint identifies a relationship between one or more attributes for one object type and one or more attributes in another type. The number and datatypes of the attributes in each set of attributes must match. Additionally, if multiple attributes make up the key, then all must allow NULLs or none can allow NULLs.

You can define foreign key constraints at either the object type level or the attribute level. If the key includes two or more participating attributes, you must define it at the type level. If the key is a single attribute, it is typically defined at the attribute level, although you can define it at the type level if you prefer.

Foreign key constraints are inherited. Defining one for a type does not override any inherited by the type. Any defined for a type are applied to its subtypes also.

You must have at least Sysadmin privileges to create a foreign key.

Documentum uses the terms *parent* and *child* to describe the relationship between the two object types in a foreign key. The type for which the constraint is defined is the child and the referenced type is the parent. For example, in the following statement, `project_record` is the child and `employee` is the parent:

```
CREATE TYPE "project_record"  
("project_lead" string(32),  
"dept_name" string(32),  
"start_date" date)  
FOREIGN KEY ("project_lead", "dept_name")  
REFERENCES "employee" ("emp_name", "dept_name")
```

Both object types must exist in the same repository, and corresponding parent and child attributes must be of the same datatype.

The child's attributes can be one or more single-valued attributes or one or more repeating attributes. You cannot mix single-valued and repeating attributes. The attributes can be inherited attributes, but they must all be defined for the same object type.

The parent's attributes can only be single-valued attributes. The attributes can be inherited attributes, but they must all be defined for the same object type.

Not null

A NOT NULL constraint identifies an attribute that isn't allowed to have a null value. You can only define a NOT NULL constraint at the attribute level. You can define NOT NULL constraints only for single-valued attributes.

Check

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

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

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

Default lifecycles for types

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 business policy, when a user creates an object of that type, the user can simply use the keyword `Default` to identify the lifecycle when attaching the object to the lifecycle. There is no need to know the lifecycle's 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 simply provides an easy way for users to identify the default lifecycle for any particular type, 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 attributes in the `dm_policy` object that defines the lifecycle in the repository. The attributes 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` attribute references one of the type's supertypes and `include_subtypes` is `TRUE`.

For complete information about lifecycles, refer to [Chapter 10, Lifecycles](#).

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's `class_name` attribute and a acronym that represents the component's 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.

Default values for attributes

An attribute's default value is the value Content Server assigns the attribute when new objects of the type are created unless the user explicitly sets the attribute value.

Localized text

The data dictionary's 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 attributes, some help text, and error messages.

Value assistance

Value assistance provides a list of valid values for an attribute. A value assistance specification defines a literal list, a query, or a routine to list possible values for an

attribute. Value assistance is typically used to provide a pick list of values for an attribute associated with a field on a dialog box.

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 attributes, 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 by the application.

The data dictionary and lifecycle states

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 life cycle, 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 could define value assistance to provide users with a pick list of valid version labels at each state of a document's life cycle. Or, you could define check constraints for each state, to ensure that users have entered the correct version label.

Retrieving data dictionary information

You can retrieve data dictionary information using DQL queries or API methods.

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.

Using the API returns the most recent information, including changes, because the API queries the `resync_needed` attribute and implicitly republishes the information if that attribute is `TRUE`. However, using the API only allows you to obtain one data dictionary value with execution of the methods.

Neither DQL or API queries return data dictionary information about new object types or added attributes until that information is published, through an explicit Publish_dd or through the scheduled execution of the Data Dictionary Publisher job.

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 attributes 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
```

For a full description of this keyword, refer to [Special keywords, page 19](#), in the *Content Server DQL Reference Manual*.

To ensure the query returns current data dictionary information, examine the resync_needed attribute. If that attribute is TRUE, the information is not current and you can re-publish before executing the query.

Using the API

To use the API to obtain data dictionary information, you must use an identifier in the format:

```
ttypename[.attr_name]
```

This method is more involved, with more steps than using DQL. Also, if you want locale-specific values, it can only return them from the current session locale.

To use a type identifier:

1. Execute the Type method to obtain the type identifier:

```
type, session, object_type[, attribute][, bus_policy_id, state]
```

For object types, the returned identifier has the format *ttypename*. For attributes, the identifier's format is *ttypename.attr_name*.

The type or attribute may have some information that is overridden when an object is in a specific business state. If you include the optional *bus_policy_id* and *state*

arguments in the Type method, the procedure returns the information defined for the business state instead of the default information.

2. Execute a Dump or Get method, using the type identifier returned by the Type method.

In the Dump output listing of a type's attributes, all of the inherited attributes are listed first, followed by those defined for the type. If you want to determine which are inherited and which are defined, use the start_pos attribute. This attribute contains the position number of the first attribute defined for the type. For example, if start_pos is 47, then all attributes starting with number 47 were defined for the type. (These include any user-defined attributes.)

If you want to retrieve only the information in one data dictionary attribute, use the Get method. For example, the following Get method retrieves the label_text attribute value for the authors attribute:

```
get, session, tdm_document.authors, label_text
```

Manipulating object types

The Documentum object model is extensible. This extensibility allows you to provide users with the customized object types and attributes needed to meet the particular requirements of their jobs.

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 of the following types:

- The SysObject type and its subtypes
- The user type and its subtypes
- The relation type
- A user-defined type that has no supertype and any of its subtypes

New object types are created using the CREATE TYPE statement. ([Create Type, page 71](#), in the *Content Server DQL Reference Manual* contains instructions for using Create Type.)

Altering object types

An object type's definition includes its structure (the attributes 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 attributes. If the object type is a custom type, you can change the structure and the default values. You can add attributes, drop attributes, or change the length definition of character string attributes in custom object types.

Object types are altered using the ALTER TYPE statement. You must be either the type's owner or a superuser to alter a type. Refer to [Alter Type, page 45](#), in the *Content Server DQL Reference Manual* for information about its use and a list of all possible alterations.

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

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.

Content 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. DROP TYPE is described in [Drop Type, page 94](#), in the *Content Server DQL Reference Manual*.

Manipulating objects

Objects are created and manipulated using methods and DQL statements. The ability to create and manipulate objects is controlled by object-level permissions and user privilege levels.

Methods and DQL

Methods are operations that you can perform on an object.

In the DFC, methods are part of the interface for individual classes. Each class 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 DFC and its classes and interfaces, refer to *Developing DFC Applications*.

In the Documentum Client Library (DMCL), methods also apply to object classes, or types, but there is not a one-to-one correspondence. That is, some DMCL methods may be applicable to multiple object types. For example, you can use the Save method to save SysObjects or workflow objects. For information about the methods in the DMCL, refer to the *Content Server API Reference Manual*.

DQL is Documentum's Document Query Language. 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 attribute values or adding content. For complete information about DQL, refer to the *Content Server DQL Reference Manual*.

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

Permissions and privileges

The ability to create objects is controlled by object-level permissions and 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.

The ability to access objects is controlled by object-level access permissions.

User privilege levels and object-level permissions are described in [Privileges and permissions, page 87](#). A complete description of how to assign the privileges and create ACLs is found in the *Content Server Administrator's Guide*.

Destroying objects

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. Refer to [Alias scopes, page 311](#), in the *EMC Documentum Object Reference Manual* for more information about relationships.) 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 (described in [Removing versions, page 122](#)).

By default, destroying an object does not remove the object's 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, you use `dmclean` and `dmfilescan`. You can run these utilities as jobs or manually. The jobs are described in [Chapter 12, Tools and Tracing](#), in the *Content Server Administrator's Guide*. [Using dmclean, page 270](#), and [Using dmfilescan, page 273](#), in the *Content Server Administrator's Guide* describe how to execute the utilities manually.

However, if the content file is stored in a storage area with digital shredding enabled and the content is not shared with another object, then 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 (the version identified by the chronicle ID), Content Server does not actually remove the object from the repository. Instead, it sets the object's `i_is_deleted` attribute 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` attribute 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.

Changing an object's object type

Documentum gives you the ability to change an object's type with some constraints. This feature is particularly 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 attributes. 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.

The DQL `CHANGE...OBJECT[S]` statement lets you make that change. This statement changes one or more objects from their current type to a new type.

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 `dm_document`, then you cannot change an object of type A directly to type B.

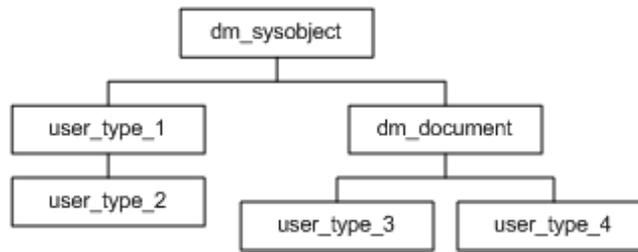
- The object that you want to change cannot be immutable (unchangeable).

For information about immutability and which objects are changeable, refer to [Changeable versions, page 123](#).

[Figure 3–1, page 78](#), shows an example of a type hierarchy. In this example, you can change `user_type_2` to either `user_type_1` or `dm_sysobject`. Similarly, you can change `user_type_1` to either `user_type_2` or `dm_sysobject`, or `dm_sysobject` to either `user_type_1` or `user_type_2`. However, you cannot change `dm_sysobject`, `user_type_1`, or `user_type_2` to `dm_document` or any of the document subtypes. This is because `dm_sysobject` and its user-defined subtypes have a different type identifier than `dm_document` objects.

If you wanted to change `user_type_3` to `user_type_4`, you would have to first change `user_type_3` to `dm_document` and then to `user_type_4`. Changing `user_type_3` directly to `user_type_4` is not allowed because it is a lateral change. Only vertical changes are allowed.

Figure 3-1. Sample hierarchy



Security Services

This chapter describes the security features supported by Content Server. These features maintain system security and the integrity of the repository. They also provide accountability for user actions. The chapter includes the following topics:

- [Security overview, page 79](#)
- [Users and groups , page 82](#)
- [User authentication, page 84](#)
- [Password encryption, page 85](#)
- [Application-level control of SysObjects, page 86](#)
- [Privileges and permissions, page 87](#)
- [Folder security, page 91](#)
- [ACLs, page 91](#)
- [Auditing and tracing, page 93](#)
- [Signature requirement support, page 95](#)
- [Digital shredding, page 104](#)

Security overview

Documentum Content Server security services has multiple standard security features and two features available with a Trusted Content Services license.

Standard security features

The following security features are part of a standard Content Server installation. Some of these are enabled automatically and some are optional. The features are:

- User authentication

- Password encryption
- Application-level control of SysObjects
- User privileges
- Object-level permissions
- Table permits
- Dynamic groups
- ACLs
- Folder security
- Auditing and tracing facilities
- Simple electronic sign-offs and support for digital signatures
- Secure (SSL) communications between Content Server and the client library (DMCL) on client hosts

User authentication occurs automatically, regardless of whether repository security is active. [User authentication, page 84](#), describes that feature in more detail.

Password encryption protects passwords stored in a file. Content Server automatically encrypts the passwords it uses to connect to third-party products such as an LDAP directory server or the RDBMS and the passwords used by internal jobs to connect to repositories. Content Server also supports encryption of other passwords through three API methods and a utility. For information, refer to [Password encryption, page 85](#).

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. [Application-level control of SysObjects, page 86](#), describes this option in detail.

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. Object-level permissions define which users and groups can access a SysObject and which level of access those users have. Table permits are a set of permits applied only to registered tables, RDBMS tables that have been registered with Content Server. Dynamic groups are groups whose membership is dynamic. If a group is dynamic, you can control its membership at runtime. For information about user privileges, object-level permissions, and table permits, refer to [Privileges and permissions, page 87](#). For information about users and groups, including dynamic groups, refer to [Users and groups , page 82](#).

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. [ACLs, page 91](#), describes ACLs.

Folder security is an adjunct to repository security. For information about folder security, refer to [Folder security, page 91](#).

Auditing and tracing are optional features that you can use to monitor the activity in your repository. For information about those features, refer to [Auditing and tracing, page 93](#).

Content Server supports three options for electronic signatures. Support for simple sign-offs, which use the Signoff method, and for digital signatures, which is implemented using third-party software in a client application, are provided as standard features of Content Server. (Support for the third option, using the Addesignature method, is only available with a Trusted Content Services license, and is not available on Linux or HP Itanium platforms.) For information about all three options, refer to [Signature requirement support, page 95](#).

When you install Content Server, the installation procedure creates two service names for Content Server. One represents a native, non-secure 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 if you like. For more information about setting the connection mode for servers, refer to [Setting the secure connection mode, page 109](#), in the *Content Server Administrator's Guide*. For information about how to configure the dmcl.ini to allow clients to request a native or secure connection, refer to [Requesting a native or secure connection, page 168](#), the *Content Server Administrator's Guide*.

Trusted Content Services security features

If you install Content Server with a Trusted Content Services license, the following security options are available:

- Encrypted file store storage areas
- Digital shredding of content files
- Electronic signature support using the Addesignature method

Note: Electronic signatures are not supported on the Linux or Itanium platforms.

- Ability to add, modify, and delete the following types of entries in an ACL:
 - AccessRestriction and ExtendedRestriction
 - RequiredGroup and RequiredGroupSet
 - ApplicationPermit and Application Restriction

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 may be a component of a distributed store. [Encrypted file store storage areas, page 103](#), describes encrypted storage areas in detail.

Digital shredding provides a final, complete way of removing content from a storage area by ensuring that deleted content files may not be recovered by any means. [Digital shredding, page 104](#) provides a brief description of this feature.

Addesignature is the way to implement an electronic signature requirement through Content Server. Addesignature 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 complete details on how this feature works, refer to [Signature requirement support, page 95](#).

The additional types of entries that you can create in an ACL if you have a Trusted Content Services license provide maximum flexibility in configuring access to objects. For example, if an ACL has a RequiredGroup entry, then any user trying to access an object controlled by that ACL must be a member of the group specified in the RequiredGroup entry. For more information about the permit types that you can define with a TCS license, refer to [ACLs, page 91](#).

Users and groups

Users and groups and the object-level permissions assigned to them are the heart of the repository security provided by Content Server.

Users

A user is typically an individual person. To access objects in a repository, a person must be represented by a `dm_user` object in the repository. 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. A user's state is controlled by the `user_state` attribute in the user's `dm_user` object.

Another useful user attribute is the `workflow_disabled` attribute. This attribute controls whether user can receive workflow tasks in their inboxes. If the attribute is set to 1, meaning unavailable, a user cannot receive any workflow tasks. (The attribute is 0, meaning available, by default.)

A user can be a virtual person. That is, you can create a user object for a user who doesn't exist in reality. Doing this may be useful; for example, if you want an application to process certain user requests and want to dedicate an inbox to those requests. The virtual user can be registered to receive events arising from the requests, and the application can read that user's inbox.

Groups

Groups are sets of users, groups, or a mixture of both. They are used to assign permissions or client application roles to multiple users. There are three kinds of groups in a repository: standard groups, role groups, and domain groups.

A standard group consists of a set of users. The users can be individual users or other groups. 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 user 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.

A role group contains a set of users, 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` attribute to `role` and the `group_name` attribute to the role name. A domain group represents a particular client domain. A domain group contains a set of role groups, corresponding to the roles recognized by the client application.

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 is responsible for examining its associated domain group and determining the role group to which the user belongs. The application is also responsible for ensuring that the user performs only the actions allowed for members of that role group. Content Server does not enforce client application roles.

A group, like 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.

Dynamic groups

A dynamic group is a group, of any group class, whose list of members is considered a list of potential members. A setting in the group's 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.

Dynamic groups provide a layer of security by allowing you to dynamically control who can be treated by Content Server as a member of a group. For more information about dynamic groups, refer to [Dynamic groups, page 379](#), in the *Content Server Administrator's Guide*.

Mixing dynamic and non-dynamic groups in group memberships

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.

User authentication

When a user or application attempts to open a repository connection or re-establish a timed out connection, Content Server immediately authenticates the user account. 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, Content Server then 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

Content Server supports user authentication against the operating system, against an LDAP directory server, or using a plug-in module.

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 LDAP directory server, you can configure the directory server to use an external

password checker or to use a secure connection with Content Server. If you choose to use a plug-in module, you can use either of the modules provided with Content Server or write and install a custom module.

Documentum provides one authentication plug-in. The plug-in implements Netegrity SiteMinder and supports Web-based Single Sign-On.

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, his or her user account is deactivated in the repository. For details, refer to [Limiting authentication attempts, page 375](#), in the *Content Server Administrator's Guide*.

For information about all the options and instructions for implementing them, refer to [Chapter 10, Managing User Authentication](#), in the *Content Server Administrator's Guide*.

Password encryption

In Documentum, the passwords used by Content Server to connect to third-party products such as an LDAP directory server or the RDBMS and those used by many internal jobs to connect to a repository are stored in files in the installation. To protect these passwords, Content Server automatically encrypts them. When a method includes one of these encrypted passwords in its arguments, the DMCL automatically decrypts the password before passing the arguments to Content Server.

In addition to the automatic password encryption, Content Server provides an API method, `Encryptpass`, that allows you to use encryption in your applications and scripts. Use `Encryptpass` to encrypt passwords used to connect to a repository. All the API methods that accept a repository password accept a password encrypted using the `Encryptpass` method. The DMCL will automatically perform the decryption.

Passwords encrypted using `Encryptpass` are encrypted using the AEK (Administration Encryption Key). The AEK is installed during Content Server installation. After encrypting a password, Content 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, refer to [Managing encrypted passwords, page 371](#), in the *Content Server Administrator's Guide*. For more information about `Encryptpass`, refer to its description in the *Content Server API Reference Manual*.

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. User access to controlled objects must be limited to approved client applications. Documentum Content Server supports this requirement by supporting application-level control of SysObjects by client 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's `a_controlling_app` attribute to its application code to identify the object as belonging to it. Once set, the attribute 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 `application_code` attribute in either the `apiconfig` or `sessionconfig` object when the application is started. (Setting the attribute in the `apiconfig`, rather than the `sessionconfig`, provides performance benefits.) The `application_code` attribute is a repeating attribute. On start-up, an application can set `application_code` to multiple application codes if users are allowed to modify objects controlled by multiple applications through that particular application.

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

Privileges and permissions

Content Server supports a set of user privileges, object-level permissions, and table permits that determine what operations a user can perform on a particular object.

User privileges are always enforced whether repository security is turned on or not. Object-level permissions and table permits are only enforced when repository security is on. Repository security is controlled by the `security_mode` attribute in the `docbase` config object. The attribute is set to `ACL`, which turns on security, when a repository is created. Consequently, unless you have explicitly turned security off by setting `security_mode` to `none`, object-level permissions and table permits are always enforced.

This section briefly describes the user privileges, object-level permissions, and table permits. For a complete discussion and instructions on assigning privileges and permissions, refer to the security information in the *Content Server Administrator's Guide*.

User privileges

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.

Basic user privileges

[Table 4-1, page 87](#), lists the basic user privileges.

Table 4-1. User privilege names and levels

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 them 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 Content Server as the server always have Superuser privileges.

Extended user privileges

Table 4–2, page 88 lists the extended user privileges.

Table 4-2. Extended user privileges

Level	Name	Description
8	Config Audit	User can execute the Audit and Unaudit 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 may grant and revoke extended user privileges, but they may not grant or revoke these privileges for themselves.

Object-level permissions

Object-level permissions are access permissions assigned to every SysObject (and SysObject subtype) in the repository. There are two kinds of object-level permissions: base permissions and extended permissions. Table 4–3, page 89, lists the base permissions.

Table 4-3. Base object-level permissions

Level	Permission	Description
1	None	No access is permitted.
2	Browse	The user can look at attribute values but not at associated content.
3	Read	The user can read content but not update.
4	Relate	The user can attach an annotation to the object.
5	Version	The user can version the object.
6	Write	The user can write and update the object.
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.

Table 4-4, page 89, lists the extended permissions.

Table 4-4. Extended object-level permissions

Permission	Description
Change Location	<p>In conjunction with the appropriate base permission level, allows the user to move an object from one folder to another.</p> <p>All users having at least Browse permission on an object are granted Change Location permission by default for that object.</p> <p>Note: Browse permission is not adequate to move an object. For a description of privileges necessary to link or unlink an object, refer to the Link and Unlink method descriptions in the <i>Content Server API Reference Manual</i>.</p>
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.

Permission	Description
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.

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

Object-level permissions are defined as entries in ACL objects. Each SysObject (or SysObject subtype) object has an associated ACL. The entries in the ACL identify users and groups and define their object-level permissions to the object with which the ACL is associated. ACLs are described in more detail in [ACLs, page 91](#).

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 also. Superusers have Read permission and all extended permissions except Delete Object by default on any object.

Table permits

The table permits control access to the RDBMS tables represented by registered tables in the repository. 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.

There are five levels of table permits, described in [Table 4-5, page 90](#).

Table 4-5. Table permits

Level	Permit	Description
0	None	No access is permitted
1	Select	The user can retrieve data from the table.

Level	Permit	Description
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 not hierarchical. For example, assigning the permit to insert does not confer the permit to update. To assign more than one permit, you add together the integers representing the permits you want to assign and set the appropriate attribute to the total. For example, the following DMCL Set method sets the group permit on a document to both insert and update:

```
dmAPISet ("set,s0,0900002e4311125e,group_table_permit","6")
```

Folder security

Folder security is a supplemental level of repository security. When folder security is turned on in addition to repository security, the server performs the object-level permission checks and for some operations, also checks and applies permissions on the folder in which an object is stored or on the object's primary folder.

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.

For a complete list of the extra checks imposed by folder security, refer to the security information in the *Content Server Administrator's Guide*.

ACLs

Access control lists, or ACLs, are the mechanism that Content Server uses to impose object-level permissions on SysObjects. Each SysObject has an ACL, identified in the `acl_name` and `acl_domain` attributes of the object. The entries in the ACL define who can access the object and the level of access accorded to that user or group.

The ACL itself is stored in the repository as an object of type `dm_acl`. An ACL's entries are recorded in repeating attributes in the object.

After an ACL is assigned to an object, the ACL is not unchangeable. 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 create them through the API or DQL also. For instructions on creating ACLs, refer to [Managing ACLs, page 387](#), in the *Content Server Administrator's Guide*. For information about assigning ACLs, refer to [Assigning ACLs, page 147](#).

ACL entries

The entries in the ACL 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. Creating, modifying, or deleting AccessPermit and ExtendedPermit entries is supported by all Content Servers.

The remaining entry types provide extended capabilities for defining access. For example, an AccessRestriction entry restricts a user or group's access to a specified level even if that user or group is granted a higher level by another entry. (For a full description of all the permit types, refer to [The ACL object type, page 388](#), in the *Content Server Administrator's Guide*.) You must have installed Content Server with a Trusted Content Services license to create, modify, or delete any entry other than an AccessPermit or ExtendedPermit entry.

Note: A Content Server enforces all ACL entries regardless of whether the server was installed with a Trusted Content Services license or not. The TCS license on affects the ability to create, modify, or delete entries.

Categories of ACLs

ACLs are either external or internal ACLs. They are also further categorized as public or private.

External ACLs are ACLs 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 Content 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, Content 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 Content 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.

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 about aliases, refer to [Appendix A, Aliases](#).)

Auditing and tracing

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

Auditing

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

By default, Content Server always audits the following system events:

- All executions of an Audit or Unaudit method
- User login failure
- All executions of a Signoff, Addsignature, or Addsignature method.

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 to 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

The record of audited events is 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 attribute values in the audited object. (For information about viewing an audit trail, refer to [Viewing audit trails, page 412](#), of the *Content Server Administrator's Guide*.)

An Audit method is used to store auditing requests in the repository. The Audit method arguments identify the event to be audited, the target of the event, and the names of any attributes whose values you want to audit. For example, you can issue an Audit method to request auditing of checkin events on a particular document. The event can be a system event or an application event. The information in the Audit method arguments is stored in the repository in registry objects. Each registry object represents one auditing request.

Issuing an Audit method to request auditing of 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 method.

For complete information about auditing, how to initiate it, and the information stored in an audit trail object, refer to [Auditing, page 406](#), of the *Content Server Administrator's Guide*.

Tracing

Content Server supports multiple tracing facilities. You can turn on tracing using a Trace method or using the `SET_OPTIONS` or `MODIFY_TRACE` administration methods. The tools in the administration tool suite, described in the *Content Server Administrator's Guide*, also generate trace files for their operations. Jobs that you create can also generate tracing information if you set the `method_trace_level` argument for the job.

For information about tracing, refer to:

- [Trace, page 456](#), in the *Content Server API Reference Manual*
- [SET_OPTIONS, page 312](#), in the *Content Server DQL Reference Manual*

- [MODIFY_TRACE](#), page 269, in the *Content Server DQL Reference Manual*
- [Chapter 12, Tools and Tracing](#), in the *Content Server Administrator's Guide*

Signature requirement support

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

Content Server supports signature requirements with three options:

- Electronic signature
- Digital signature
- Simple sign-offs

Electronic signatures are generated and managed by Content Server. The feature is supported by two API methods: `Addsignature` and `Verifysignature`. Use this option if you require a rigorous signature implementation to meet regulatory requirements. You must have a Trusted Content Services license to use this option. For detailed information, refer to [Electronic signatures](#), page 95.

Note: Electronic signatures are not supported on the Linux or Itanium platforms.

Digital signatures are electronic signatures in formats such as PDKS #7, XML signature, or PDF signature. Digital signatures are generated by third-party products called when an `Addsignature` method is executed. Use this option if you want to implement strict signature support in a client application. For more information, refer to [Digital signatures](#), page 101.

Simple sign-offs are the least rigorous way to supply an electronic signature. Simple sign-offs are implemented using the `Signoff` method. This method authenticates a user signing off a document and creates an audit trail entry for the `dm_signoff` event. [Signoff method usage](#), page 102, describes this option in detail.

Electronic signatures

Note: This feature is not supported on Linux or HP Itanium platforms. On supported platforms, it requires a Trusted Content Services license.

Electronic signatures are generated by Content Server when an application or user issues an `Addsignature` method. Using `Addsignature` is a rigorous way to fulfill a signature requirement. Signatures generated by `Addsignature` are recorded in a

formal signature page and added to the content of the signed object. The `Addesignature` method is audited automatically, and the resulting audit trail entry is itself signed by Content Server. The auditing feature cannot be turned off. If an object requires multiple signatures, before allowing the addition of a signature, Content Server verifies the preceding signature. Content Server also authenticates the user signing the object.

All the work of generating the signature page and handling the content is performed by Content 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, Content 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. Documentum 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. (The default template and method are described in detail in [The default signature page template and signature method, page 98](#).) 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. For instructions, refer to [Customizing electronic signatures, page 431](#), in the *Content Server Administrator's Guide*.

What Addesignature does

When an application or user issues an Addesignature method, Content 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

5. Retrieves all the audit trail entries for the previous dm_addesignature events on this content.

6. Verifies that the most recent audit trail entry is signed (by Content Server) and that the signature is valid

7. Verifies that the entries have consecutive signature numbers

8. Verifies that the hash in the audit trail entry matches the hash of the document content

9. Copies the content to be signed to a temporary directory location and calls the signature creation method. The signature creation method

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

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

12. 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, Content Server appends the original, unsigned content to the document as a rendition with the page modifier set to dm_sig_source.

13. 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. For instructions on turning on tracing, refer to [Tracing electronic signature operations, page 440](#), in the *Content Server Administrator's Guide*.

The default signature page template and signature method

Documentum provides a default signature page template and a default signature creation method with Content Server so that you can use the electronic signature feature with no additional configuration. The only requirement to use 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.

Default signature page template

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

In the repository, the 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 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.

Default signature creation method

The default signature creation method is a Docbasic method named esign_pdf.ebs, stored in %DM_HOME%\bin (\$DM_HOME/bin). The method uses the PDF Fusion library to generate signature pages. The PDF Fusion library and license is installed during Content Server installation. The Fusion libraries are installed in %DM_HOME%\fusion (\$DM_HOME/fusion). The license is installed in the Windows directory on Windows hosts and in \$DOCUMENTUM/share/temp on UNIX platforms.

The signature creation method uses the location object named SigManifest to locate the Fusion library. The location object is created during repository configuration.

The signature creation method checks the number of signatures supported by the template page. If the maximum number is not exceeded, the method generates a signature page and adds that page to the content file stored in the temporary location by Content Server. The method does not read the content from the repository or store the signed content in the repository.

How content is handled by default

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, Content 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.

Audit trail entries

Content 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.)

Content Server uses the generic string attributes in the audit trail entry to record information about the signature. [Table 4-6, page 99](#), lists the use of those attributes for a `dm_addesignature` event.

Table 4-6. Generic string attribute use for `dm_addesignature` events

Attribute	Stores
<code>string_1</code>	Name of the user who signed the object
<code>string_2</code>	The justification for the signature

Attribute	Stores
string_3	<p>The signature's 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 <code>pre_signatureHash</code> argument of the <code>Addsignature</code> method.</p> <p>The information is formatted in the following manner:</p> <pre>sig_number/method_name/pre_signature hash argument</pre>
string_4	<p>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:</p> <pre>hash_algorithm/format_name/hash</pre>
string_5	<p>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:</p> <pre>hash_algorithm/format_name/hash</pre> <p>If the signed content was added to the document as primary content, then value in <code>string_5</code> is the same as the <code>string_4</code> value.</p>

What you can customize

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 may also create a custom signature page template for use by the custom signature creation method; however, using a template is not required.

For instructions on all customizations to electronic signatures, refer to [Customizing electronic signatures, page 431](#), in the *Content Server Administrator's Guide*.

Verifying signatures

Electronic signatures added by Addesignature are verified by the Verifiesignature 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 Verifyaudit method to verify the Content 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.

General usage notes

Here are some general notes about working with electronically signed documents:

- Users can modify a signed document's attributes without invalidating the signatures.
- If the signed document was created on a Macintosh machine, modifying the resource fork does not invalidate the signatures.
- Addesignature cannot be executed against an object that is checked out of the repository.
- Checking out a signed document and then checking it in as the same version invalidates the signatures on that version and prohibits subsequent signings.
- If you dump and load a signed document, the signatures are not valid in the target repository.
- If you replicate a signed document, executions of Addesignature or Verifiesignature against the replica will act on the source document.
- Using Addesignature to sign a document requires at least Relate permission on the document.
- Using Verifiesignature to verify a signature requires at least Browse permission on the signed document.

Digital signatures

Digital signatures are electronic signatures in formats such as PKCS #7, XML Signature, or PDF Signature. Signatures in these formats are implemented and managed by the client application. The 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 a 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.

Note: If you wish assistance in creating, implementing, or debugging a digital signature implementation in an application, you must contact Documentum Professional Services or Documentum Developer Support.

Content Server supports digital signatures with an attribute on SysObjects and the Adddigsignature method. The attribute is a Boolean attribute called a_is_signed. The Adddigsignature 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.

An application using digital signatures typically implements the following steps for the signatures:

1. Obtain the user's signature.
2. Extract the signature from the document and verify it.
3. If the verification succeeds, set the a_is_signed attribute to T.
4. Check the document in to the repository.
5. Issue the Adddigsignature method to generate the audit trail entry.

It is possible to require Content Server to sign the generated audit trail entries. Because the Adddigsignature method is audited by default, there is no explicit registry object for the event. However, if you want Content Server to sign audit trail entries for dm_adddigsignature events, you can issue an explicit Audit method for the event, setting the sign_event argument to TRUE in the Audit method.

For more information about Content Server signatures on audit trail entries, refer to [Signing audit trail entries, page 410](#), in the *Content Server Administrator's Guide*. For information about using Audit, refer to the method description, [Audit, page 115](#), in the *Content Server API Reference Manual*.

Signoff method usage

Simple sign-offs are the least rigorous way to enforce a signature requirement. 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 implemented using a Signoff method. The method accepts a user authentication name and password as arguments. When the method is executed, Content Server calls a signature validation program to authenticate the user. If authentication succeeds, Content 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 Signoff 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 instructions, refer to [Customizing simple signoffs, page 441](#), in the *Content Server Administrator's Guide*.

For a description of the Signoff method, refer to the Javadocs or to [Signoff, page 449](#), in the *Content Server API Reference Manual*.

Encrypted file store storage areas

Note: Encrypted file store storage areas are only available if you have installed Content Server with a Trusted Content Services license.

An encrypted file store storage area is a file store storage area that contains encrypted content files. 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 non-encrypted.

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

When you store content in a encrypted file store storage area, the encryption occurs automatically. Content is encrypted by Content 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` attribute of the storage area object. It is encrypted using the repository encryption key. (For information about the repository encryption key, refer to [Repository encryption keys, page 445](#), in the *Content Server Administrator's Guide*.)

Similarly, the decryption occurs automatically also, when the content is fetched from the storage area for a user.

Encrypted content may be full-text indexed. However, the index itself is not encrypted. If you are storing non-indexable content in an encrypted storage area and indexing renditions of the content, the renditions are not encrypted either 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. For more information, refer to [Dumping a repository, page 48](#), in the *Content Server Administrator's Guide*.

Digital shredding

Digital shredding is an optional feature available for file store storage areas if you have installed Content Server with a Trusted Content Services license. 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 utilizes 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. It is not supported for these storage areas if they are components of a distributed storage area. You may also enable shredding for file store storage areas that are the targets of linked store storage areas.

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.

Server Internationalization

This chapter describes Documentum's approach to server internationalization, or how Content Server handles code pages. It discusses Unicode, which is the foundation of internationalization at Documentum, and issues that arise as you plan for server internationalization.

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. Content Server uses a data dictionary to provide localized values for applications. For information about the data dictionary and localization, refer to [Chapter 3, The Data Model](#).

The chapter contains the following topics:

- [What is internationalization?](#), page 105
- [Content files and metadata](#), page 106
- [Configuration requirements for internationalization](#), page 107
- [Where ASCII must be used](#), page 111
- [User names, email addresses, and group names](#), page 111
- [Lifecycles](#), page 112
- [Docbasic](#), page 112
- [Federations](#), page 112
- [Object replication](#), page 113
- [Other cross-repository operations](#), page 113
- [Dump and load operations and the session code page](#), page 113

What is internationalization?

Internationalization means that Content Server's features and design do not make assumptions based on a single language or *locale*. (A locale represents a specific geographic region or language group.) Instead, Content Server can store metadata and content files from all languages and locales in a single repository and provide support for storing all languages in all supported Content Servers. For example, Content Server

is certified on English, Japanese, and Korean versions of different operating systems. Content files and metadata from all languages can be stored on each of those operating systems. This supports client internationalization and localization.

Documentum does not localize products that have administrative interfaces only. These products include Content Server and Documentum Administrator. Their interfaces are in English. However, the data dictionary provides support for localizing attribute labels. For more information on the data dictionary, refer to [The data dictionary, page 64](#).

Content 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.

Content 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 an NCS to Unicode and back without loss. Only common data can be transcoded from one NCS to another. Characters that are present in one NCS cannot be transcoded to another NCS in which they are not available.

Content Server's use of Unicode enables the following features:

- Ability to store metadata using non-English characters
- Ability to store metadata in multiple languages
- Ability to manage multilingual Web and enterprise content

For more information about Unicode, UTF-8, and National Character Sets, refer to the Unicode Consortium's Web site at <http://www.unicode.org/>.

Content files and metadata

A repository stores content files on a file system and metadata in the database.

All content files are transferred to and from a repository as binary files. You can store text, graphics, music, and any other content files in the repository. For files that contain text content, you can store content files in any format, regardless of encoding. For example, your repository can contain content files in English, Chinese, Russian, Greek, German, Italian, Finnish, and Arabic. Note, however, that XML content may be best stored in a single encoding. For information on XML content, refer to *Managing XML Content in Documentum*.

What metadata you can store depends on the code page of the 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.

Client applications may use National Character Sets. When a client application with the `client_codepage` set to an NCS sends metadata to Content Server, the server transcodes the data to the UTF-8 encoding of Unicode and then stores the metadata in the codepage used by the repository's database.

All code pages supported by EMC Documentum 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.

Configuration requirements for internationalization

Before you install the server, it is important to set the 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 Content Server, refer to the *Content Server Installation Guide*.

During the 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's operating system. These parameters are used by the server in managing data, user authentication, and other functions.

Documentum has recommended locales for the server host and recommended code pages for the server host and database.

Values set during installation

Some locales and code pages are set during Content Server installation and repository configuration. The following sections describe what is set during installation. For a table of the default values for these, refer to [Table 3–1, page 87](#), in the *Content Server Administrator's Guide*.

The server config object

The server config object describes a Content Server and contains information that the server uses to define its operations and operating environment.

- `locale_name`

This is the locale of the server host, as defined by the host's 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`

This is the default code page used by clients connecting to the server. The value is determined programmatically and set during server installation. It is strongly recommended that you do not reset the value. The `client_codepage` key in the `dmcl.ini` file overrides the `default_client_codepage` attribute.

- `server_os_codepage`

This is the code page used by the server host. Content 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. It is strongly recommended that you do not reset the value.

Values set during sessions

During an API or repository session, values are set in the API config object and session config object that control interactions between the server and the client.

In an API session, set the `session_codepage` attribute in the session config object early in the session and do not reset it.

The server does not support multiple server sessions in different code pages during the same API session.

The api config object

An api config object describes the configuration of an API session. It is created when a client issues a `dmAPIInit` call. The values reflect the information found in the `dmcl.ini` file on the client host. Some of the values are then used in the session config object when a client opens a repository session.

- `client_codepage`

The value of the `client_codepage` attribute is taken from the `client_codepage` key in the `dmcl.ini` file on the client host. This code page is the preferred code page for repository sessions started in the API session. The value of `client_codepage` overrides the value of the `default_client_codepage` attribute in the server config object.

- `client_locale`

This is the client's preferred locale for repository sessions started in the API session.

The 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 attribute values are taken from values in the `api config` object, the `server config` object, and the `connection config` object.

- `session_codepage`

This attribute is obtained from the `api config` object's `client_codepage` attribute. It is the code page used by a client application connecting to the server from the client host.

- `session_locale`

This is the locale of the repository session. The value is obtained from the `client_locale` attribute of the `api config` object. If `client_locale` is unset, the value is determined programmatically from the locale of the client's host machine.

How values are set

The values of the `client_codepage` and `client_locale` attributes in the `api config` object determine the values of `session_codepage` and `session_locale` in the session config object. The DMCL determines these values as follows:

1. Use the values supplied programmatically by an explicit `Set API` on the `api config` object or `session config` object.
2. If such an API is not called, examine the settings of `client_codepage` and `client_locale` in the `dmcl.ini` file on the client host.
3. If the values are not set in the `dmcl.ini` file
 4. On Windows, checks the value of the Registry key `sLanguage`
 5. On UNIX, checks the value of the environment variable `LANG`

On Windows, Content Server chooses the values of `session_locale` and `session_codepage` based on the value of `sLanguage`:

Value of <code>sLanguage</code>	Value of <code>session_locale</code>	Value of <code>session_codepage</code>
ENU	en	ISO_8859-1
DEU	de	ISO_8859-1
ESN	es	ISO_8859-1
ITA	it	ISO_8859-1
FRA	fr	ISO_8859-1
JPN	ja	Shift_JIS
KOR	ko	EUC-KR

On UNIX, Content Server chooses the values of `session_locale` and `session_codepage` based on the value of `LANG`.

If the value of `LANG` exactly matches one of the values in the first column below, then `session_locale` and `session_codepage` are set as in the second and third columns:

Value of <code>LANG</code>	Value of <code>session_locale</code>	Value of <code>session_codepage</code>
C	en	ISO_8859-1
de	de	ISO_8859-1
es	es	ISO_8859-1
it	it	ISO_8859-1
fr	fr	ISO_8859-1
ja	ja	EUC-JP
ko	ko	EUC-KR

If the value of `LANG` is in the format `language[_territory][.codepage][@modifier]` and the `codepage` component is specified, then if the code page matches a supported code page, the value of `session_codepage` is set as follows:

Value of code page	Value of <code>session_codepage</code>
ISO8859_1	ISO_8859-1
ISO88591	ISO_8859-1
eucKR	EUC-KR
eucJP	EUC-JP

Value of code page	Value of session_codepage
SJIS	Shift_JIS
PCK	Shift_JIS

Where ASCII must be used

Some objects, names, directories, and attribute values must contain only ASCII characters. These include:

- Content 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 Content Server is installed
- Location object names
- The value of the file_system_path attribute of a location object
- Mount point object names
- Format names
- Format DOS extensions
- All file store names
- Object type names and attribute 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

User names, email addresses, and group names

There are code page-based requirements for the following attribute 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's configuration. If the repository is a standalone repository, the values in the attributes must be compatible with the code page defined in the server's `server_os_codepage` attribute. (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 attribute must be compatible with the `server_os_codepage`. However, if the repositories have different code pages identified in `server_os_codepage`, then the values in the attributes listed above must consist of only ASCII characters.

Lifecycles

The scripts that you use as actions in lifecycle states must consist of only ASCII characters.

Docbasic

Docbasic does not support Unicode. For all Docbasic server methods, the code page in which the method itself 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 Content Server host (the `server_os_codepage`).

Docbasic scripts run on client machines must be in the code page of the client operating system.

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 attribute values must use only ASCII characters:

- `user_name`

- user_os_name
- user_address
- group_address

ACLs can use Unicode characters in ACL names.

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's database uses Unicode. If the French repository's database uses Latin-1, replication fails.

In mixed code page environments, the source and target folder names must consist of only ASCII characters. The folders contained by the source folder are not required to be named with only ASCII characters.

When you create a replication job, set the code page to UTF8 if the source and target repositories are version 4.2 or better. If one of the repositories is pre-4.2, set the code page to the code page of the source repository.

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.

Dump and load operations and the session code page

If you use dump and load operations, the session code page in use during the load operation must be the same as the session code page used during the dump operation. If the session code pages are not the same, the metadata is corrupted during the load operation.

The dump file header does not indicate the session code page in which the dump file was created. If you do not know the session code page in use when a dump file was created, do not load the dump file.

The session code page is the code page specified in the `dmcl.ini` file on the host from which you connect to the repository to run the dump operation, or it may have been taken implicitly from the environment. Examine the value of the `client_codepage` key in `dmcl.ini` to determine the session code page. If the key is not set, the `client_codepage` key defaults to the operating system's code page.

You can modify the `dmcl.ini` on the host from which you connect to the repository or you can set the session code page explicitly if you want to use a specific code page. If you are connecting from the Content Server host, do not modify the `dmcl.ini` file.

Valid code pages for the session code page are US-ASCII, UTF-8, ISO_8859-1, Shift-JS, EUC-JP, and EUC-KR. Do not use a code page that is incompatible with the RDBMS code page. For example, do not dump a Japanese repository using the ISO-8859-1 (Latin-1) code page.

It is strongly recommended that you use UTF-8 for the session code page when you create the dump file. However, if the code pages of the source and target repositories are the same, no metadata corruption occurs if the session code page matches that of the repositories. For example, if you dump a Latin-1 repository using a session in Latin-1 and then load the data into a different Latin-1 repository, no metadata corruption occurs.

To set the session code page:

1. Start IAPI.
2. Connect to the repository.
3. Issue these commands

```
set,c,sessionconfig,session_codepage  
codepage
```

where *codepage* is the code page in which you want to dump the repository.

Issue the dump command after you set the session code page. For more information on how to use dump and load, refer to [Chapter 2, Content Repositories](#), in the *Content Server Administrator's Guide*.

Content Management Services

This chapter describes the support provided by Content Server for objects with content. For information about the underlying infrastructure and its management (storage areas, retention policies, digital shredding, and so forth), refer to the *Content Server Administrator's Guide*. This chapter includes the following topics:

- [Introducing SysObjects, page 115](#)
- [Documents, page 116](#)
- [Versioning, page 118](#)
- [Immutability, page 123](#)
- [Concurrent access control, page 126](#)
- [Document retention and deletion, page 128](#)
- [Documents and lifecycles, page 131](#)
- [Creating SysObjects, page 132](#)
- [Modifying SysObjects, page 140](#)
- [Managing content across repositories, page 151](#)
- [Managing translations, page 153](#)
- [Working with annotations, page 154](#)
- [User-defined relationships, page 157](#)

Introducing SysObjects

SysObjects are the supertype, directly or indirectly, of all object types in the hierarchy that can have content. The SysObject type's defined attributes store information about the object's version, the content file associated with the object, the security permissions on the object and other information important for managing content.

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

Documents

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 Documentum, documents are represented by `dm_document` objects, a subtype of `SysObjects`. You can use a document object to represent an entire document or only some portion of a document. For example, a document can contain text, graphics, or tables.

Simple and virtual documents

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 the primary content files in a simple document must have the same file format.

A *virtual document* is a document that includes one or more components structured as an ordered hierarchy. A *component* can be a simple document or another virtual document. A virtual document can have any number of components, nested to any level.

Using virtual documents lets you combine documents with a variety of formats into one document. It also allows you to use one document in a variety of larger documents. For example, you can place a graphic in a simple document and then add that document as a component to multiple virtual documents.

Working with virtual documents is described in [Chapter 7, Virtual Documents](#).

Content objects

A content object is the connection between a document object and the file that actually stores the document's content. Every content file in the repository, whether in a repository storage area or external storage, has an associated content object. The attributes 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.

Content 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` attribute in the content object. The `parent_id` attribute records the object IDs of all documents to which the content belongs. There is only one content object for each content file in the repository.

Page numbers

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.

Renditions

A *rendition* of a document is a content file that differs from the source document's content file only in its format. Renditions are created by Content Server using supported converters or by Documentum Media Transformation Services™. Media Transformation Services is an optional product that lets you manage rich media such as jpeg and various audio and video formats. With Media Transformation Services, you can create thumbnails and other renditions for the rich media formats. [Chapter 8, Renditions](#), contains an in-depth description of renditions generated by Content Server and briefly describes renditions generated by Media Transformation Services. Renditions generated by Media Transformation Services are described in detail in the Media Transformation Services documentation.

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 Content Server or AutoRenderPro™ are always connected through a content object. For these renditions, the rendition attribute in the content object is set to indicate that the content file represented by the content object is a rendition. The page attribute 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. Which is used typically depends on the transformation profile used to transform the source content file. 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.

For information about creating renditions using converters and managing renditions, refer to [Chapter 8, Renditions](#). For information about using Documentum Media

Transformation Services to create renditions, refer to *Administering Documentum Media Transformation Services*.

Primary content pages and renditions

When you create a rendition, the rendition is associated with the primary content file through the page number of the primary content. (Refer to [Page numbers, page 117](#) for information about page numbers.)

Translations

Content Server contains support for managing translations of original documents using relationships. For details about setting up a translation relationship, refer to [Managing translations, page 153](#).

Versioning

Content Server provides comprehensive versioning services for all SysObjects except folders and cabinets and their subtypes. (Those SysObject subtypes cannot be versioned.)

Versioning creates a historical record of a document. Your business rules may require you to keep copies of all old versions of a document. Each time you check in or branch a document or other SysObject, Content Server creates a new version of the object without overwriting the previous version. All the versions are stored in a hierarchy called a version tree. Each version on the tree has its own numeric version label. The server automatically provides numeric version labels and keeps track of the current version on the tree.

Version labels

Every SysObject object or SysObject subtype in the repository has a repeating attribute called `r_version_label` that stores the object's implicit version label and any number of symbolic version labels.

Implicit version labels

The implicit version label is a numeric label. It 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 implicit 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 implicit 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 implicit 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, you can provide labels that are meaningful to your application and work environment.

Symbolic labels are stored starting in the second position (`r_version_label[1]`) in the `r_version_label` attribute. To define a symbolic label, simply define it in the argument list when you check in or save the document. For example, the following DMCL API command assigns the symbolic version label `inprint` to a document:

```
dmAPIGet ("checkin,s0,0900000182643c5a,,inprint")
```

An alternative way to define a symbolic label is to use a 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 to move a symbolic label from one document version to another. (For detailed information about using a Mark method, refer to [Mark](#), page 310, in the *Content Server API Reference Manual* or consult the Javadocs.)

A document can have any number of symbolic version labels. Symbolic labels are case sensitive and must be unique within a version tree. (Refer to [Version trees](#), page 120, for a description of version trees.)

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:

```
dmAPIGet ("checkin,s0,0900000182643c5,,inprint,CURRENT")
```

If you remove a version that carries the CURRENT label, the server automatically reassigns the label to the parent of the removed version. (Refer to [Removing versions, page 122](#), for more information about removing versions.)

Uniqueness

Because both implicit and symbolic version labels are used to access a version of a document, Documentum ensures that the labels are unique across all versions of the document. The server enforces unique implicit version labels by always generating an incremental and unique sequence number for the implicit labels.

Content Server also enforces unique symbolic labels. If a symbolic version label specified with a Checkin, Save, or Mark method matches the symbolic label already assigned to another version of the same object, then the existing label is removed and the specified label is applied to the version indicated by the Checkin, Save, or Mark method.

Note: Remember that 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.

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.

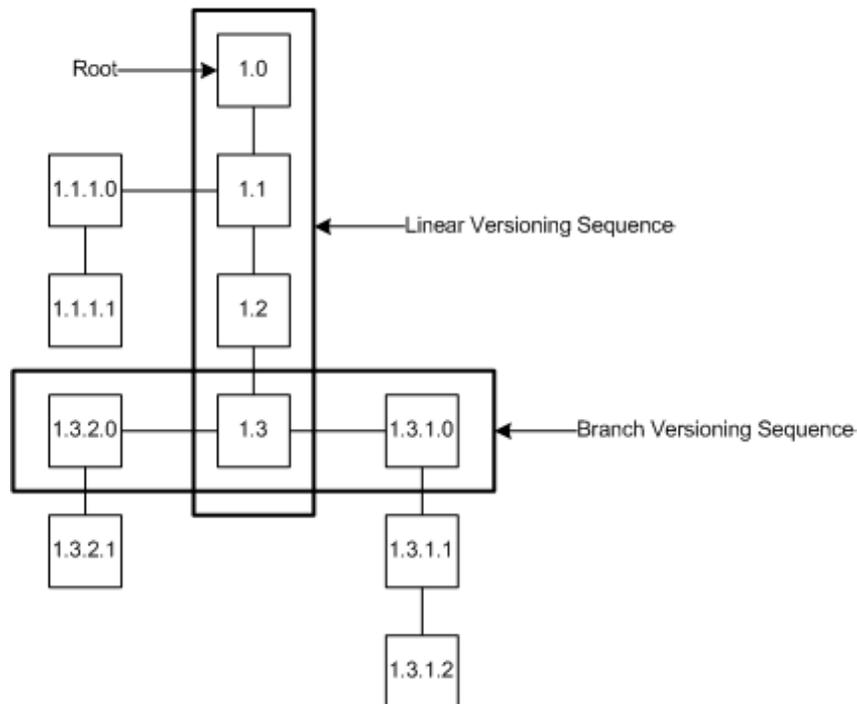
To identify which version tree a document belongs to, the server uses the document's `i_chronicle_id` attribute value. This attribute contains the object ID of the document's original version, the root of the version tree. Each time you create a new version, the server copies the `i_chronicle_id` value 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 a document's place on a version tree, the server uses the document's implicit version label.

Branching

A version tree is often a linear sequence of versions arising from one document. However, you can also create branches. [Figure 6-1, page 121](#), shows a version tree that contains branches.

Figure 6-1. A version tree with branches



The implicit 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 example, version 1.3 is the origin of two branches. These branches begin with the implicit version labels 1.3.1.0 and 1.3.2.0. If we were to create a branch off version 1.3.1.2, 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 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 object ID of 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.

Removing versions

Content Server provides two ways to remove a version of a document. If you want to remove only one version, use a Destroy method. If you want to remove more than one version, use a 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.

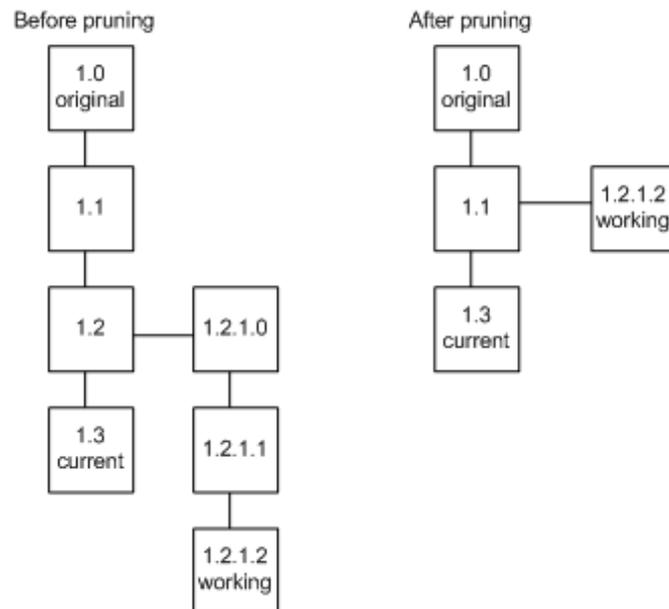
To prune an entire version tree, identify the first version of the object in the method's arguments. (The object ID of the first version of an object is found in the `i_chronicle_id` attribute of each subsequent version.) Query this attribute if you need to obtain the object ID of an object's first version.

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 shown in [Figure 6-1, page 121](#), 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` attribute of any remaining version to the appropriate parent.

For example, look at [Figure 6-2, page 123](#). Suppose the version tree shown on the left 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 on the right. Notice that the remaining versions have not been renumbered.

Figure 6-2. Before and after pruning

Changeable versions

You can modify the most recent version on any branch of a version tree. For instance, in [Figure 6–1, page 121](#), 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 those older versions.) [Immutability, page 123](#), is described in detail in the next section.

Immutability

Immutability is a characteristic that defines an object as unchangeable. An object is marked immutable when the object is versioned or a Freeze method is executed against the object or the object is associated with a retention policy that designates controlled documents as immutable.

Effects of a Checkin or Branch method

When a user creates a new version of a document (or any SysObject or SysObject subtype), Content Server sets the `r_immutable_flag` attribute to TRUE in the old version. Users can no longer change the old version's content or most of its attribute values.

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 attributes. The content, primary storage location and the frozen attributes 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 a Halt method.

When you freeze an object, the server sets the following attributes of the object to TRUE:

- `r_immutable_flag`

This attribute indicates whether the object is changeable. If set to TRUE, you cannot change the object's content, primary storage location, or most of its attributes. ([Attributes that remain changeable, page 125](#), lists the attributes that can be changed in a frozen object.)

- `r_frozen_flag`

This attribute indicates whether the `r_immutable_flag` attribute was set to TRUE by an explicit Freeze method call.

If the object is a virtual document, other attributes are also set. Refer to [Freezing a document, page 179](#), for details.

A Freeze method has an optional argument that directs the server to freeze the object and the components of any snapshot associated with the object if the argument is set to TRUE.

To unfreeze an object, use an 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 optional flag that unfreezes the components of a snapshot associated with the object if the argument is set to TRUE. (For the details of how unfreezing affects a virtual document, refer to [Unfreezing a document, page 179](#).)

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's `r_immutable_flag` attribute is set to `TRUE`.

Attributes that remain changeable

Some attributes are changeable even when an object's `r_immutable_flag` attribute is set to `TRUE`. Users or applications can change the following attributes:

- `r_version_label` (but only symbolic labels, not the implicit label)
- `i_folder_id` (the object can be linked or unlinked to folders and cabinets)
- the security attributes (`acl_domain`, `acl_name`, `owner_name`, `group_name`, `owner_permit`, `group_permit`, `world_permit`)
- `a_special_app`
- `a_compound_architecture`
- `a_full_text` (requires Sysadmin or Superuser privileges)
- `a_storage_type`

The server can change the following attributes:

- | | |
|--------------------------------|------------------------------------|
| • <code>a_archive</code> | • <code>r_current_state</code> |
| • <code>i_isdeleted</code> | • <code>r_frozen_flag</code> |
| • <code>i_reference_cnt</code> | • <code>r_frzn_assembly_cnt</code> |
| • <code>i_vstamp</code> | • <code>r_immutable_flag</code> |
| • <code>r_access_date</code> | • <code>r_policy_id</code> |
| • <code>r_alias_set_id</code> | • <code>r_resume_state</code> |

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`. (Data dictionary attributes are set using the `ALTER TYPE` statement. For instructions on using `ALTER TYPE`, refer to [Alter Type, page 45](#), in the *Content Server DQL Reference Manual*.)

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. Content Server provides three locking strategies for SysObjects:

- Database-level locking
- Repository-level locking
- Optimistic Locking

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 a DQL BEGINTRAN statement or a Begintran method. 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 Lock method.

Database locks provide a way to ensure that deadlock doesn't occur in explicit transactions and that Save operations don't fail due to version mismatch errors.

If you use database locks, using repository locks also 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.

Repository-level locking

Repository-level locking occurs when a user or application checks out a document or object. When a Checkout occurs, Content Server sets the object's `r_lock_owner`, `r_lock_date`, and `r_lock_machine` attributes. 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 be a superuser. (Refer to [Object-level permissions, page 88](#), for an introduction to the object-level permissions. User privileges are described in the *Content Server Administrator's Guide*.)

Repository locks are released when you issue Checkin methods. A Checkin 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 overwrites the current version of an object with the changes you have made, have an optional argument that directs the server to hold the repository lock.

An Unlock method also removes repository locks. This method cancels a checkout. Any changes you made to the document are not saved to the repository.

If a user is dropped from the system, any repository locks held by that user are also dropped.

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's `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's `i_vstamp` attribute 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.

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.

Content Server supports two ways to apply retention:

- Retention policies

Using retention policies requires a Retention Policies Services license. If Content Server is installed with that license, you can define and apply retention policies through 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. [Retention policies, page 128](#), briefly describes retention policies. For more information about retention policies and their use, refer to Documentum Administrator online help.

- 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.

Using a content-addressed storage area requires a Content Services for EMC Centera license. (This license is not supported on HP Itanium.) For more information about content-addressed storage areas, refer to [Content-addressed storage, page 227](#), in the *Content Server Administrator's Guide*.

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 the `i_retain_until` attribute of any document under its control is set to five years from the date at which the policy was applied. If a policy specifies a date, the `i_retain_until` attribute is set to the date.

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.

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 Content 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.

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.

For more information about how retention periods in a content-addressed storage area are defined and how they are implemented internally, refer to [Defining storage area retention requirements, page 257](#), in the *Content Server Administrator's Guide*.

If a retention policy and storage-based retention apply

If a document is assigned to a retention policy and the document's 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's content address is set to the date furthest in the future.

Similarly, the attribute `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. Content Server will not allow the

document to be deleted (without using a forced deletion) until May 31, 2013. The `i_retain_until` attribute is set to May 31, 2013.

Deleting documents under retention

Deleting documents associated with an active retention policy or with unexpired retention periods in a content-addressed storage area requires use of 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.

For more information about privileged deletions, refer to the Documentum Administrator online help.

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. For instructions on enabling its use for a forced deletion, refer to [Enabling forced deletion in content-addressed storage areas, page 268](#), in the *Content Server Administrator's Guide*.)

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.

Deleting old versions and unneeded renditions

As documents are checked in and out of 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. [Removing versions, page 122](#), describes using destroy or prune in detail. [Version Management, page 536](#), in the Content Server Administrator's Guide, describes the tool in detail.

You can remove unneeded renditions using the Rendition Management administration tool. [Rendition Manager, page 520](#), in the Content Server Administrator's Guide, describes the tool in detail.

Documents and lifecycles

Business policies define life cycles for documents and other SysObjects. They consist 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's object type. Only a user with the Change State extended permission can move the document from one state to another.

For instructions about creating lifecycles, attaching documents to lifecycles, and moving documents through the life cycle, refer to [Chapter 10, Lifecycles](#).

Creating SysObjects

SysObjects, represented by the `dm_sysobject` object type and its subtypes, are the only types in the object type hierarchy that accept content. To create a SysObject such as a document, with content, a user or application must perform the following steps:

1. Create the object.
2. Set its attributes.

An object's attributes define several important characteristics of the object, such as where it is stored in the repository and what permissions are associated with the object. [Setting the object's attributes, page 132](#), provides some guidelines for this step.

3. Add the content.
4. If the content is to be stored in a content-addressed storage area, set the content-related attributes in the content object. (Documentum clients perform this step silently when needed. No action is required by users.)
5. Save or import the object into the repository.

Creating the object

End users typically use an Documentum client application, such as WebPublisher or Webtop, to create documents or to import documents created in an external editor. Applications use the DFC to create objects. For information about using a Documentum client to create documents, refer to the documentation for that particular client. For information about instantiating an object using DFC, refer to the DFC documentation.

It is also possible to create an object using the DQL `CREATE...OBJECT` statement. However, if you use DQL, there are some constraints on adding content to the object when you execute the statement. Refer to [Create...Object, page 67](#) in the *DQL Reference Manual* for details.

Setting the object's attributes

By default, the server sets read-only SysObject attributes, some of the read and write attributes, such as the security attributes, and the `a_full_text` attribute. You may want to explicitly set some of the attributes, bypassing the defaults. Additionally, you may want to set other attributes. For example, you may want to set the `object_name`, `title`, and `subject` attributes.

How you set the attribute values depends on what interface you are using to create the object. In applications, use the appropriate DFC method for each attribute. Client applications such as Webpublisher or Webtop typically provide a dialog that allows users to set properties (attributes) of a selected object. In DQL, use the set and append clauses of the CREATE...OBJECT statement.

This section provides some information and guidelines about setting some common attributes, or properties, of the object, including:

- owner_name
- keywords
- default_folder
- a_full_text
- language_code

owner_name attribute

The owner_name attribute 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 attribute. To identify another user as the owner, you must be a superuser. To identify a group as the owner of an object, you must either be a superuser or you must own the object and be a member of the group to which you are assigning ownership.

keywords attribute

Each SysObject and SysObject subtype has a repeating attribute called keywords that holds user-defined string values of up to 32 characters each. Providing one or more values for this attribute gives users an easy way to search for the document using a DQL query statement. For example, the following SELECT statement retrieves the object ID and name of all documents that have Breads defined in keywords:

```
SELECT "r_object_id", "object_name" FROM "dm_document"  
WHERE ANY "keywords" IN ('Breads')
```

default_folder attribute

The default_folder attribute records the name of an object's primary location. 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 repository. Although this location is often referred to as the object's primary cabinet, it can be either a cabinet or a folder.

The default primary location for a new document (or any other SysObject) a user creates, is the user's home cabinet. A different location can be specified:

- A user can identify the desired location when saving or importing the object into the repository using a Documentum client application.
- You can include the LINK clause in the CREATE...OBJECT statement if creating the object using DQL.
- An application can set the default_folder attribute or call a link method to link the object to the desired cabinet .

Note: To make an application portable, link methods accept aliases for folder path specifications. Content Server resolves the alias to an appropriate path at the time the application executes. For information about using aliases in link methods, refer to [Resolving aliases in Link and Unlink methods, page 314](#).

After you define a primary location for a object, it is not necessary to define the location again each time you save the object.

a_full_text attribute

All SysObjects are full-text indexed. The a_full_text attribute controls whether content and metadata are indexed or only the metadata. By default, the attribute is T (TRUE) and both the content and metadata (attribute values) are indexed. If an object's a_full_text attribute is F (FALSE), then only the object's metadata is indexed.

Content Server sets a_full_text to T automatically for all SysObjects and SysObject subtypes. You must have Sysadmin or Superuser privileges to change the value to F.

language_code attribute

The language_code attribute stores a code that identifies the language and country of origin for an object. This value is useful if your business has sites across the world. If you want to set the language_code attribute, [Appendix B, Language and Country Codes](#), in the *EMC Documentum Object Reference Manual* provides a list of recommended language codes.

Adding content

When users create a SysObject using a Documentum client, adding content is a seamless operation. Creating a new document using WebPublisher or 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. Which method is used to add the content to the object depends on whether the content is a file or data block.

The first content file

When you add the first primary content file to an object, you must identify the content's format. All methods that add content provide an argument that identifies the content format. In DQL, the CREATE_OBJECT statement allows you to identify the format in the SETFILE clause. For example, the following CREATE...OBJECT statement creates a new document and adds the content file called Bread Tips to the document:

```
CREATE "dm_document" OBJECT
SET "object_name" = 'Bread Tips',
SET "authors" = 'jenny',
SETFILE 'c:\files\yeasted_breads\breadtips.doc'
with content_format = 'mswm'
```

The format is recorded in the a_format_type attribute of the object.

You can also identify the format before adding the content by setting the a_format_type attribute explicitly. If you do, it is not necessary to define the format when you add the first content file.

Additional primary content files

You can add additional files as primary content when you create an object or you can add them after you save the object. However, all additional primary content files must have the same format as the first primary content file added to the document.

If you are using the API, you must issue one method for each content file you want to add before saving the object.

If you are using DQL, you can include multiple SETFILE clauses in the CREATE...OBJECT statement. For example:

```
CREATE "dm_document" OBJECT
SET "object_name" = 'Bread Tips',
```

```
SET "authors" = 'jenny',  
SETFILE 'c:\files\yeasted_breads\breadtips.doc'  
with content_format='mswm',  
SETFILE 'c:\files\yeasted_breads\bibliography.doc'  
with page_no=1
```

For all subsequent primary content files, specify the content's page number (or position) in the object's list of primary content files. Do not include the format.

Macintosh files

You cannot use DQL to add a file created on a Macintosh machine to an object. You must use an API method. Macintosh-created files have two parts: a data fork (the actual text of the file) and a resource fork. The API methods that add content to objects include an optional argument that ensures that both parts are included as the object's content.

Renditions

You can also add renditions of primary content. Renditions are typically copies of the content in a different format. For information about creating and adding renditions, refer to [Chapter 8, Renditions](#).

Assigning content to storage

Documentum 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 a site's 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. A detailed description of the implementation and use the options is found in [Allocating content to storage areas](#), page 237, in the *Content Server Administrator's Guide*.

Content assignment policies

Note: Content assignment policies are a feature of Content Storage Services. This set of services is separately licensed. A Content Server must be installed with a Content Storage Services license to use this feature.

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 a 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 policies can only assign content to file store storage areas or content-addressed storage areas. Policies are enforced by DFC-based client applications (5.2.5 SP2 and higher), and are applied to all new content files, whether created by a save or import into the repository or a checked in operation.

For complete information about the behavior and implementation of content assignment policies and creating them, refer to [Using content assignment policies, page 237](#), in the *Content Server Administrator's Guide*.

Default storage allocation

The default storage algorithm uses values in a document's 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's supertypes
- A content file does not satisfy any of the conditions in the applicable policy
- Content is saved with a retention date.

For a description of how the default storage algorithm behaves, refer to [Using the default storage algorithm, page 241](#), in *Content Server Administrator's Guide*.

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.

Assigning an ACL

An ACL, or access control list, specifies access permissions for an object. Each 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. For information about setting permissions for a document, refer to [Assigning ACLs, page 147](#).

Setting content attributes 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 attributes 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 attributes. 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 attributes. The name is placed in content_attr_name and the value is stored in the attribute corresponding to the field's 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, then its value currently stored in content_attr_value would simply be overwritten.

In a ca store object, the attributes 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 a `Setcontentattrs` method is executed, the metadata name and value pairs are stored first in the content object attributes and then the plugin 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 the content requires a retention period, you must include the metadata field identified in the storage object's `a_retention_attr_name` attribute in the list of name and value pairs in the `Setcontentattr` method's parameter argument. The value for that field 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, 2006. The `Setcontentattrs` parameter argument would include the following name and value pair:

```
'retain_date=DATE(01/01/2006)'
```

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 Content Server can interpret as a number of seconds. If you include characters that cannot be translated to a number of seconds, Content 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` to set the content object attributes and a `PUSH_CONTENT_ATTRS` to copy the metadata to the storage system.

`Setcontentattrs` must be executed after the content is added to the `SysObject` and before the object is saved to the repository. `SET_CONTENT_ATTRS` and `PUSH_CONTENT_ATTRS` must be executed after the object is saved to the repository.

For information about configuring a storage area to require a retention period for content stored in that area, refer to [Defining storage area retention requirements, page 257](#), in the *Content Server Administrator's Guide*.

Saving the new object

Saving an object for the first time creates an object of the particular type in the repository. For example, saving a document for the first time creates a document object in the repository. If you added content to the object, saving the object also stores the content file in the specified storage area and creates a content object for the file if none existed previously.

Saving a SysObject for the first time also sets the object's implicit and symbolic version labels. The default value for the implicit version label is 1.0. The default value for the symbolic label is CURRENT.

If you are using DQL, the save operation occurs automatically when the CREATE...OBJECT statement is executed successfully. You cannot override the version label defaults.

If you are using the API, use a Save method to save the object. You can override the symbolic version label default using a Save method argument. For example, the following method call saves the Book Proposal document with the version label working:

```
dmAPIExec ("save, s0, book_proposal_doc_id, , working")
```

You can include multiple symbolic labels.

If you include any version labels, include the extra comma before the label specification. The comma is a placeholder for an argument that is not used when saving a new object.

Modifying SysObjects

The ability to modify a SysObject is controlled by object-level permissions, user privileges, and whether the object is under the control of a retention policy. Each SysObject has an associated ACL object that defines the access permissions for that object. Users with Superuser privileges are subject to object-level permissions. However, because a superuser always has at least Read permission on SysObjects and the ability to modify ACLs, a superuser can always access a SysObject.

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. If that is so, then users can change only some of the document's attributes. (For a list of the changeable attributes in immutable documents, refer to [Attributes that remain changeable, page 125.](#))

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. The attributes that can be changed are described in [Attributes that remain changeable, page 125](#).

Before a user or application can modify a SysObject, the object must be obtained from the repository. After the object is modified, the changes must be written back to the repository.

Getting a document from the repository

When you want to work with a SysObject, you must to retrieve the object 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's metadata from the repository. Retrieving the object's content requires a `getfile` or `getcontent` method. However, you must execute a lock, checkout, or fetch before retrieving the content files.

A lock 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.

A checkout method 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. The checkout method also offers you two alternatives for saving the document when you are done. You need `Version` or `Write` permission to use the checkout method.

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; it only guarantees that one user cannot overwrite the changes made by another. Consequently, it is possible to fetch a document, make changes, and then 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.

(For a full description of locks and locking strategies, refer to [Concurrent access control, page 126](#).)

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's current primary content, use a `getfile` or `getcontent` to retrieve the content file.

Modifying single-valued attributes

If you modify the value of a single-valued attribute, the new value overwrites the old value.

In the DFC, most attributes have 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.

Setting the `a_full_text` attribute

The `a_full_text` attribute is set to `TRUE` by Content Server when a `SysObject` is saved for the first time. If you have `Sysadmin` or `Superuser` privileges, you can reset `a_full_text` after the document is saved.

Setting the `a_content_type` attribute

An object's content format is set the first time you add primary content to the document. However, if you discover that an object's `a_content_type` attribute is set incorrectly, it is not necessary to re-add the content. You can check out the object, reset the attribute, and save (or check in) the object.

Modifying repeating attributes

You can modify a repeating attribute by adding additional values, replacing current values, or removing values.

Adding values

To add values to a repeating attribute, use either an `append` method or a `set` method. Use an `append` to add one or more values to the existing values in the attribute. The new values are always added to the end of the list of values in the attribute.

You can also use a set method. However, if you use a set method, you must specify the index position of the new value in the method's argument. If you specify the index position of an existing value, the method will overwrite the existing value with the new value. Consequently, if you want to add a new value without substituting the new value for an old value, an append method is the recommended way to do so.

Replacing a value

Use a set method to replace an existing value. The set methods require you to specify the index position of the value you are replacing. When executed, they replace the value at the specified index position with the new value.

The set methods are best used when you want to replace an existing value. If you only want to add a new value, use an append method.

Removing a value

To remove a repeating attribute value, use a remove method. You must identify the index position of the value you want to remove. The index positions of any values after the removed value are adjusted appropriately.

Performance tip for repeating attributes

How long it takes the server to append or insert a value for a repeating attribute increases in direct proportion to the number of values in the attribute. Consequently, if you want to define a repeating attribute for a type and you expect that attribute 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 that joins the type and the table.

Adding content

When you add primary content to an object, you can append the new file to the end of the object's list of content files or you can insert the file into the list. [Table 6-1, page 144](#), lists the DFC methods available for adding content to a SysObject. These methods are defined for the IDfSysObject interface. (The javadocs provide more information about these methods.) Use the method appropriate for the operation being performed. The

content can be a file or a block of data, but it must reside on the same machine as the client application.

Table 6-1. Method choices for adding content

Method name	When to use
appendFile	Use to append an external file to a document's list of content files.
appendContent	Use to append a block of data in memory to a document's list of content files.
insertFile	Use to insert or replace content when the new content is an external file.
insertContent	Use to insert or replace content when the new content is a block of data in memory.
setFileEx	Use to add an external file as primary content. Supports arguments defining format, page number, and Macintosh files. You may use setFileEx as an alternative to insertFile to replace content.
setFile	Use to replace the first page of existing primary content
setContentEx	Use to add a block of data in memory as primary content. Supports arguments defining format, page number, and Macintosh files. You may use this as an alternative to insertContent to replace content.

Although the DQL CREATE...OBJECT and UPDATE...OBJECT statements support a clause that allows you to add content, using DQL is not recommended because it bypasses any TBOs or SBOs that you may have associated with the object or operation. Using DQL also requires superuser privileges. However, you must use DQL if the content file resides on the server host machine or a disk that is visible to the server host but not the client machine. For information about this statement, refer to the *DQL Reference Manual*.

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's page number, rather than its format.

The page number must be the next number in the object's 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.

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 document referenced as `mySysObject` and that 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)
```

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.

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's `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.

Writing changes to the repository

The following methods write changes to the repository:

- `checkin`
- `checkinEx`
- `save`
- `saveLock`

Checkin and Checkinapp 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.

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 a checked out or fetched objects. A `saveLock` method works only on checked out objects.

If the document has been signed using `Addsignature`, using `Save` to overwrite the signed version invalidates the signatures and will prohibit the addition of signatures on future versions.

Assigning ACLs

An ACL is assigned to each SysObject when the SysObject is created. The ACL can remain with the object for the life of the object, or you can replace the ACL as the object moves through its life. Alternatively, you can change the entries in the ACL associated with an object, rather than assigning it a new ACL.

When you create a document or other SysObject, you can:

- Assign a default ACL (either explicitly or allow the server to choose)
- Assign an existing non-default ACL
- Generate a custom ACL for the object

This section does not describe how to create an ACL, only how to assign an existing ACL to a document. For information about creating new ACLs, refer to [Creating ACLs, page 398](#), in the *Content Server Administrator's Guide*.

Assigning a default ACL

There are three ACLs that can be used as default ACLs:

- The ACL associated with the object's primary folder

An object's 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's 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's 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.

When a new object is created, Content Server assigns one of these defaults to the object if the object's creator does not explicitly assign an ACL to the object. Which default is assigned by Content Server is controlled by the value in the `default_acl` attribute in the server's server config object. In a newly configured repository, that attribute is set to the value identifying the user's ACL as the default ACL. You can change the setting through Documentum Administrator.

To assign a default ACL that is different from the default identified in the `default_acl` attribute, the object's creator or the application must issue a `Useacl` method.

Assigning an existing non-default ACL

If you are a document's owner or a superuser, you can assign to the document any private ACL that you own or any public ACL, including any system ACL.

In an application, if the application is designed to run in multiple contexts with each having differing access requirements, assign a template ACL. When the application executes, the template is instantiated as a system ACL when it is assigned to an object. 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, Content 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.

Generating custom ACLs

Custom ACLs are created by the server when you use a `grantPermit` or `revokePermit` method against a `SysObject` to define access control permissions for the object.

Note: You must have installed Content Server with a Trusted Content Services license to grant or revoke any of the following permit types:

- `AccessRestriction` or `ExtendedRestriction`
- `RequiredGroup` or `RequiredGroupSet`
- `ApplicationPermit` or `ApplicationRestriction`

A custom ACL's name is created by the server and always begins with `dm_`. Generally, a custom ACL is only assigned to one object; however, you can use a `set` method to assign a custom ACL to multiple objects. Any custom ACL that the server creates for you is owned by you.

There are four common situations that generate a custom ACL:

- [Granting permissions to a new object without assigning an ACL, page 149](#)
- [Using `grantPermit` to modify the ACL assigned to a new object, page 149](#)
- [Using `grantPermit` when no default ACL is assigned, page 149](#)
- [Using `grantPermit` or `revokePermit` to modify the current ACL, page 149](#)

Granting permissions to a new object without assigning an ACL

The server creates a custom ACL when you create a SysObject and grant permissions to it but do not explicitly associate an ACL with the object.

The server bases the new ACL on the default ACL defined at the server level (the value of the default_acl attribute of the server config object). It copies that ACL, makes the indicated changes, and then assigns the custom ACL to the object.

Using grantPermit to modify the ACL assigned to a new object

The server creates a custom ACL when you create a SysObject, associate an ACL with the object, and then use a grantPermit method to modify the access control entries in the ACL. 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 specified in the grantPermit method 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.

Using grantPermit or revokePermit to modify the current ACL

If you fetch an existing document and use a grantPermit or revokePermit method to 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.

Removing permissions

At times, you may need to remove a user's access or extended permissions to a document. For example, an employee may leave a project or be transferred to another location. A variety of situations can make it necessary to remove someone's permissions.

You must be owner of the object, a superuser, or have Change Permit permission to change the entries in an object's ACL.

You must have installed Content Server with a Trusted Content Services license to revoke any of the following permit types:

- AccessRestriction or ExtendedRestriction
- RequiredGroup or RequiredGroupSet
- ApplicationPermit or ApplicationRestriction

When you remove a user's access or extended permissions, you can either:

- Remove permissions to one document
- Remove permissions to all documents using a particular ACL

Use a revokePermit method to remove object-level permissions. That method is defined for both the IDfACL and IDfSysObject interfaces.

Note: The dmcl Revoke method is supported also, but the recommended way to manipulate ACLs in 5.3 and forward is using the DFC methods.

Each execution of revokePermit removes a specific entry. To preserve backwards compatibility with pre-5.3 versions, 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. (For more information about this behavior, refer to [Revoke, page 400](#), in the *Content Server API Reference Manual*. The DFC revokePermit and DMCL Revoke method behave similarly.)

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 to the group engr is retained.

Removing permissions to a single document

To remove permissions to a document, call the IDfSysobject revoke method against the document. If you are using the dmcl Revoke method, identify the document as the target of the operation in the method arguments.

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.

Removing permissions to all documents

To remove permissions to all documents associated with the ACL, you must alter the ACL. To do that, call the IDfACL revoke method against the ACL. If you are using the dmcl Revoke method, identify the ACL as the target of the operation in the method arguments.

Content Server modifies the specified ACL. Consequently, the changes affect all documents that use that ACL.

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. If you want 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. However, users typically replace an ACL using facilities provided by a client interface.

Managing content across repositories

In a multi-repository 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.

Like other users, applications can also work with remote objects. After an application opens a session with a repository, it can work with remote objects directly, by opening

a subconnection, or it can indirectly reference the remote objects through reference links. Or, depending on the purpose, the user or application might choose to work with a replica.

Reference links and replicas are the Content Server features that provide the capability of working across repositories.

Reference links

A reference link is a pointer in one repository to an object in another repository. Reference links are created automatically by the following operations:

- Linking a remote object to a local folder or cabinet
- Checking out a remote object
- Adding a remote object to a local virtual document

A reference link is a combination of a mirror object and a dm_reference object.

Mirror objects

A mirror object is an object in one repository that mirrors an object in another repository. The term *mirror object* describes the object's 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's attribute data. When the system creates a mirror object, it does not copy the object's 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.

Only a limited number of operations can affect mirror objects. You can link them to local folders or cabinets, and you can retrieve their attribute values.

The majority of API methods must execute against the actual object represented by the mirror object. With the exception of Getfile, API methods require you to specify an indirect reference in their arguments when you are working on a remote object.

For information about indirect references, refer to [Indirect references](#), page 30, in the *EMC Documentum Object Reference Manual*.

Reference objects

Every mirror object has an associated reference object. Reference objects are the internal links between mirror objects and their source objects in remote repositories. Reference objects are persistent. They are stored in the same repository as the mirror object and are managed by Content Server. Users never see reference objects.

Applications can create reference links by creating `dm_reference` objects directly if needed. When they do, the system automatically creates the associated mirror object. For information about how to create and manipulate reference objects in an application, refer to [Appendix B, Writing Distributed Applications](#).

Replicas

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. ([Object replication, page 44](#), in the *Distributed Configuration Guide* explains object replication in detail.)

Like mirror objects, a replica has an associated reference object that points back to the source of the replica. When a user performs an operation on a replica, depending on the operation, it may affect the source object or the replica. [Replica objects, page 52](#), in the *Distributed Configuration Guide* describes replicas in detail and which operations on replicas affect the replica and which affect the source.

Managing translations

Documents are often translated into multiple languages. Content 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's 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. Documentum provides a recommended set of language and country codes in [Appendix B, Language and Country Codes](#), of the *EMC Documentum Object Reference Manual*.

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.

To create a translation relationship between two documents:

1. Use a Create method to create a dm_relation object.
2. Set the parent_id attribute of the relation object to the object ID of the original document.
3. Identify the translation by setting the child_id and optionally, the child_label attribute in the relation object.

If you set only the child_id, set it to the object ID of the translation.

If you set both attributes, the child_id attribute must be set to the chronicle ID of the version tree that contains the translation, and child_label must be set 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's chronicle ID and child_label to APPROVED.

4. Identify the relationship by setting the relation_name attribute to DM_TRANSLATION_OF.
5. Set any of the optional attributes that you want to set.
[Relation, page 371](#), in the *EMC Documentum Object Reference Manual* lists the attributes defined for the relation object type.
6. Save the relation object.

Working with annotations

Annotations are comments that a user attaches to a document (or any other SysObject or SysObject subtype). Throughout a document's development, and often after it is published, people may want to record editorial suggestions and comments. For example, several managers may 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 `Addnote` 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's `parent_id` attribute contains the document's object ID and its `child_id` attribute contains the note object's ID. The `relation_name` attribute contains `dm_annotation`, which is the name of the relation type object that describes the annotation relationship. (Refer to [Relationships, page 34](#), for a complete description of relation objects, relation type objects, and their attributes.)

Creating annotations

To create an annotation for an object:

- You must have at least `Relate` permission for the object you are annotating.
- You must own the note object that represents the annotation or you must have at least `Write` permission for the note.

To create an annotation for a document:

1. Create the file for the annotation.
2. Create the note object.
3. Attach the file to the note object.

Refer to [Table 6–1, page 144](#), for information about the methods that add content to an object.

4. Use an `addNote` method to attach the annotation to the desired document.
5. Save the note object.

These steps are described in detail in the following sections.

The annotation file

The content files of a note object contain the comments you want to make about a document. (Like other `SysObject` subtypes, a note object can have multiple content files.)

The files can have any format. Their format is not required to match the format of the documents to which you are attaching them. Also, multiple annotations attached to the same document are not required to have the same format. However, if you put multiple content files in the same note object, those files must have the same format.

Attaching the annotation to the document

Annotations are attached to a document using an `addNote` method. Attaching an annotation to a document creates a relation object that links the note object and the document and describes the nature of their relationship. You must own the note object or have at least `Write` permission to it and must have at least `Relate` permission to the document.

The *keepPermanent* argument sets the `permanent_link` attribute in the relation object that represents the relationship between the note and the document. If you set this to `TRUE`, when the document is versioned, saved as new, or branched, the annotation is copied and attached to the new version or copy. By default, this flag is `FALSE`.

Detaching annotations from a document

To detach an annotation from a document, the following conditions must be true:

- You must have at least `Write` permission to the annotation.
- You must have `Relate` permission to the document to which it is attached.

To detach annotations from the document, use a `removeNote` method. Detaching an annotation from a document destroys the relation object that described the relationship between the annotation and the document.

Deleting annotations from the repository

To delete an annotation (note object) from the repository, you must be a system administrator, a superuser, or the owner of the annotation.

To delete a single annotation, use a `destroy` method. Destroying a note object automatically destroys any relation objects that reference the note object.

To delete orphaned annotations (note objects that are no longer attached to any object), use `dmclean`, a utility that performs repository clean-up operations. You must be a system administrator or a superuser to run `dmclean`.

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 saveasnew 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.

User-defined relationships

A relationship is an association between two objects in the repository. Some system-defined relationships are installed with Content Server. (For example, annotations are implemented as a system-defined relationship between a `SysObject`, generally a document, and a note object.) You may also define your own kinds of relationships.

Before you can connect two objects in a user-defined relationship, the relationship must be described in the repository. Relationships are described by `dm_relation_type` objects. (Refer to [Relationships, page 34](#), in the *EMC Documentum Object Reference Manual* for full information about relationships and about creating relation type objects.)

Each instance of a particular relationship is described by a `dm_relation` object. 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.

User-defined relationships are not managed by Content 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.

Creating a relationship between two objects

To create a relationship between two objects, you must create the relation object that describes their relationship. A relation object represents an instance of the relationship described by a relation type object.

To create a relationship between two objects:

1. Create a `dm_relation` object.
2. Set the object's attributes.

You must set the following attributes:

- `relation_name`

This identifies the type of relationship between the two objects. It must match the value of the `relation_name` attribute of an existing relation type object.

- `parent_id`

This identifies the object that is considered the parent in the relationship. Use the object's object ID.

- `child_id`

This identifies the object that is considered the child in the relationship. Use the object's object ID.

Additionally, you may want to set the optional attribute, `child_label`. Setting the `child_label` attribute defines which version of the child participates in the relationship. The version is not required to be the same version you identified in the `child_id` attribute. If the object specified in `child_id` does not carry the correct version label, the server searches the version tree containing the version identified in the `child_id` attribute for the version specified in `child_label` and uses that version in the relationship.

Note: If the applications that will manage and manipulate the objects in the relationship are running against the DMCL or a pre-5.3 DFC, you may also want to set the `permanent_link` attribute. In the DMCL or a pre-5.3 DFC, this attribute controls whether the relationship is maintained when the parent object is copied or versioned. (In a 5.3 DFC, this behavior is controlled by attributes in the relationship's definition instead of by the `permanent_link` attribute at the relation instance level.

For more information, refer to [The permanent_link and copy_child attributes, page 36](#), in the *EMC Documentum Object Reference Manual*.)

Refer to [Relation, page 371](#), in the *EMC Documentum Object Reference Manual* for a complete list of the attributes defined for the dm_relation type.

3. Save the object.

Destroying a relationship between objects

To destroy a user-defined relationship between two objects, destroy the relation object representing their relationship. Destroying a relation object does not remove the participating objects from the repository. It only removes the connection between the objects that was established by the relation object.

The security level defined for a relationship (in the relation type object) determines who can destroy the relation objects that represent instances of the relationship. (For information about security levels, refer to [Security and relationships, page 35](#), in the *EMC Documentum Object Reference Manual*.)

Use the Destroy method to remove a relation object from the repository. Specify the relation object's object ID.

Relationships and object operations

This section describes how relationships and common operations interact.

Checkin, Saveasnew, and Branch

In the 5.3 DFC, how a relationship is handled when the parent object is checked in, saved as new, or branched depends on the setting of the two attributes, permanent_link and copy_child, in the relationship's definition (the dm_relation_type object defining the relationship type). For information about how the settings of these attributes are used, refer to [The permanent_link and copy_child attributes, page 36](#), in the *EMC Documentum Object Reference Manual*.

In the DMCL and in a pre-5.3 DFC, the permanent_link attribute in the dm_relation object itself determines how the relationship is handled when the parent is versioned or copied. If permanent_link is set to T, when the parent object is versioned or copied, a new relation object is created that associates the new copy or version with the child object.

Save

Saving an object does not change or remove any of its user-defined relationships.

However, saving an object does not automatically keep its annotations, a system-defined relationship. Refer to [Working with annotations, page 154](#), for information about this.

Destroy

When you destroy an object, the server also destroys all the relation objects that reference the destroyed object, regardless of whether the object participates in the relationship as parent or child.

Content Server behavior when you attempt to destroy an object that participates in a user-defined relationship is determined by the settings of the `direction_kind` and `integrity_kind` attributes in the relationship's definition (the `dm_relation_type` object). For details, refer to [Defining delete behavior, page 38](#), in the *EMC Documentum Object Reference Manual*.

Object replication

If a replicated object is a parent in any user-defined relationship in the source repository, the child object and the associated relation and relation type objects are replicated.

If the replicated object has annotations, the annotations are replicated.

Virtual Documents

This chapter describes virtual documents and how to work with them. 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 the API.

The chapter covers the following topics:

- [Introducing virtual documents, page 162](#), defines a virtual document.
- [Early and late binding, page 166](#), defines binding of components to virtual documents.
- [Defining component assembly behavior, page 167](#), describes how you can affect the assembly behavior of components.
- [Defining copy behavior, page 169](#), describes how to define copy behavior for a virtual document's descendants.
- [Creating virtual documents, page 169](#), describes how to create virtual documents.
- [Modifying virtual documents, page 171](#), describes how change a virtual document.
- [Assembling a virtual document, page 173](#), provides instructions for assembling a virtual document.
- [snapshots, page 164](#), defines a snapshot and provides instructions for creating one.
- [Freezing virtual documents and snapshots, page 178](#), describes how freezing and unfreezing affects a virtual document.
- [Obtaining information about virtual documents, page 180](#), describes how to query a virtual document and how to obtain a path through a virtual document to a particular component.

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 [Chapter 7, Content Management](#), before you begin working with virtual documents.

Introducing virtual documents

A virtual document is a container document. Most commonly, it contains other documents, but it can contain SysObjects or any SysObject subtype except folders, cabinets or subtypes of folders and cabinets. If the contained object is a document, that document can be a simple document or another virtual document. Documentum imposes no limits on the depth of nesting in a virtual document.

Virtual documents are a way to combine documents whose contents have a variety of formats into one document. For example, suppose you want to publish a document with text content, a document with graphic content, and a spreadsheet as one document. By making the three documents components of one virtual document, you can publish them as one document.

You can associate a particular version of a component with the virtual document or you can associate the component's entire 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. (For more information about, refer to [Early and late binding, page 166.](#))

Virtual document architecture

Documentum uses two object types and several attributes defined for the SysObject type to support virtual documents.

Object types

Content Server uses two object types to store information about virtual documents:

- Containment object type
- Assembly object type

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. (For information about the attributes controlling assembly behavior, refer to [Defining component assembly behavior, page 167.](#))

The attributes of containment objects are set by Appendpart, Insertpart, and Updatepart methods.

Assembly objects make up snapshots. A snapshot is a record of a virtual document at a particular time. It records the exact components of the virtual document at the time the snapshot was created. Each assembly object in a snapshot represents one component of the virtual document. For more information about snapshots, refer to [Snapshots](#), page 175.

SysObject attributes

Several attributes defined for the SysObject type support virtual documents. The two that you encounter most frequently are:

- `r_is_virtual_doc`

The `r_is_virtual_doc` attribute is a Boolean attribute that determines whether Documentum client applications treat the object as a virtual document. If the attribute is set to TRUE, the client applications always open the document in the Virtual Document Manager. The default for this attribute is FALSE. It is set to TRUE using the `Setdoc` method and can be set for any SysObject subtype except folders, cabinets, and their subtypes.

- `r_link_cnt`

When an object is a virtual document, the `r_link_cnt` attribute records how many direct components belong to the virtual document. Each time you add a component to a document, the value of this attribute is incremented by 1.

Component ordering

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 use an `Appendpart` or `Insertpart` method to add components to a virtual document.

If you bypass the automatic numbering provided by the server, you can use your own numbers. `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.

Versioning

You can version a virtual document and manage its versions just as you do a simple document. However, deleting a virtual document version also removes the containment objects and any assembly objects associated with that version.

For information about versioning documents and managing versions, refer to [Versioning](#), page 118.

Referential integrity and freezing

By default, Content 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` attribute in the server's server config object. By default, this attribute is `TRUE`, which prohibits users from destroying any object contained in a virtual document.

If you set this attribute 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 be a system administrator or superuser to set the `compound_integrity` attribute.

Conditional assembly

Assembling a virtual document selects a set of the document's components for publication or some other operation, such as viewing or copying. When assembling a virtual document, you can identify which components to include. You can include all the components or only some of them. If a component's 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's descendants may also be included. Whether descendants are included is controlled by two attributes in the containment objects. For information about the attributes, refer to [Defining component assembly behavior](#), page 167.

For information about the process of assembling a virtual document, refer to [Assembling a virtual document](#), page 173.

snapshots

After you assemble a virtual document, you can record what components were selected in a snapshot. A snapshot is a record of the virtual document. If you frequently publish a particular set of components, creating a snapshot provides faster access to that set than assembling them every time. For information about creating and working with snapshots, refer to [Snapshots](#), page 175.

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, like the content files belonging to a simple document, all content files associated with a virtual document must have the same format.

XML support

XML documents are supported as virtual documents in Content Server. When you import or create an XML document using the DFC (Documentum Foundation Classes), 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 attributes of containment objects: `a_contain_type` and `a_contain_desc`. The Documentum DFC and Documentum Desktop (which uses the DFC) use the `a_contain_type` attribute to indicate whether the reference is an entity or link. They use the `a_contain_desc` to record the actual identification string for the child.

These two attributes are also defined for the `dm_assembly` type, so applications can correctly create and handle virtual document snapshots using the DFC.

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. For information about creating XML applications, refer to the *XML Application Development Guide*. You can find this document through Documentum's Support web site. (Instructions for obtaining documents from the Web site are found in the Preface.)

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.

Early and late binding

Attaching a specific version of a component to a virtual document is referred to as binding. Early binding occurs if you bind a specific version of a component to a document when you add the component to the document. Late binding occurs if you bind a specific version of a component to a document when you assemble the document.

Early binding

Early binding ensures that all future snapshots of a document include the same version of the component.

To use early binding, identify the version using either an implicit or symbolic version label when you add the component to the virtual document. Using an implicit version label establishes an absolute link between the component and the document. Using a symbolic label establishes a symbolic link between the two.

Absolute links

An absolute link results when you implement early binding using the component's implicit version label. The implicit version label is the version label that the server automatically assigns to an object version when you create the version. An implicit version label does not change—it remains with that version for the life of the version.

When you create an absolute link, you link a specific version of a component to the virtual document. To alter the version of the component contained in the virtual document, you must remove the old version from the virtual document and add the new one.

Symbolic links

A symbolic link results when you implement early binding using a component's symbolic version label. The symbolic label is a version label that is generally user-defined. (Content Server can assign only one symbolic label, the label CURRENT.)

Symbolic labels can be moved from version to version of an object. Consequently, if you use symbolic linking, the virtual document always contains the version of the component that has the particular symbolic label. For example, linking components to a virtual document using the CURRENT label is an easy way to ensure that the virtual document always contains the current versions of its components.

Note: Using the CURRENT label does not guarantee that the virtual document has the most recent version, just that it contains the version defined in your system as the current version. For more information about symbolic labels, refer to [Symbolic version labels, page 119](#).

Late binding

Late binding occurs when you do not identify a version label when adding a component to a virtual document. If you do not identify a version label, the server associates the component's entire version tree with the document. The decision about which version to include in the document is deferred until the document is assembled. Late binding's flexibility is a powerful feature of virtual documents.

Defining component assembly behavior

There are two attributes in containment objects that control how components that are themselves virtual documents behave when the components are selected for a snapshot. The attributes are `use_node_ver_label` and `follow_assembly`. They are set by arguments in `Appendpart`, `Insertpart`, and `Updatepart` methods.

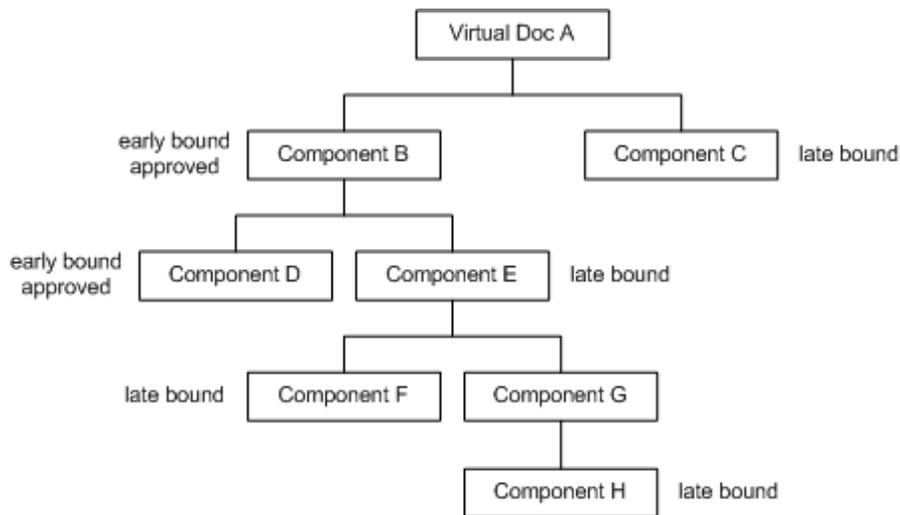
`use_node_ver_label`

The `use_node_ver_label` attribute 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's 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's 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.

To illustrate how `use_node_ver_label` works, let's use the virtual document Figure 2-1. 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 7-1. A sample virtual document showing node bindings

Component B is early bound—the specified version is the one carrying the version label approved. Because Component B is early bound and `use_node_ver_label` set to `TRUE`, when the server determines which versions of Component B's 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 Component E's 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 for which `use_node_ver_label` is 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.

follow_assembly

Follow_assembly determines whether the server selects a component's descendants using the containment objects or a snapshot associated with the component.

If you set follow_assembly to TRUE, the server selects a component's 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 attribute.

If follow_assembly is FALSE or a component does not have a snapshot, the server uses the containment objects to determine the component's descendants.

Defining copy behavior

When users copy 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 attribute in a component's containment object. It is an integer attribute 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 the user's access permissions).

Creating virtual documents

To create a virtual document using the API:

1. Obtain the object that you want to use as a virtual document.
2. Set the object's r_is_virtual_doc attribute.
3. Add components to the object.
4. Save or check in the object.

Obtaining the object

There are three ways to obtain an object that you want to use as a virtual document:

- Create a new SysObject or SysObject subtype.

Remember that folders and cabinets cannot be virtual documents.

- Fetch an object from the repository.
- Check out an object from the repository.

Setting the r_is_virtual_doc attribute

The r_is_virtual_doc attribute identifies the object as a virtual document to Documentum client applications. When the attribute is TRUE, the Documentum client applications always open the object in the Virtual Document Manager.

If users are never going to open or work with the document, setting this attribute is not necessary. However, setting it ensures that if users do work with the document, the document behaves appropriately.

Set the attribute using a Setdoc method.

Adding components

Two methods add components to a virtual document: Appendpart and Insertpart. Appendpart adds components to the end of the ordered list of components that make up the virtual document. Insertpart inserts components into the ordered list of components at any location.

Typically, you use Appendpart when you are creating a virtual document by adding one or more components to a virtual document that has no existing components. If the virtual document already has components, use Appendpart only if you want to add a new component at the end of the current list of components.

Use Insertpart when you want to insert a component into an existing list of components. Insertpart lets you control where the component is inserted.

Appendpart assigns order numbers automatically. If you want to bypass the server's automatic numbering, you must use Insertpart to add components because only Insertpart lets you control the order number.

Note: Neither Appendpart or Insertpart sets the r_is_virtual_doc attribute. They only increment the r_link_cnt attribute.

You cannot add components to a frozen virtual document. (Refer to [Freezing virtual documents and snapshots](#), page 178, for more information.)

Saving the changes

How you save the changes to the repository depends on how you obtained the virtual document:

- If you created a new object, use a Save method to put the object in the repository.
Saving a new object requires Write permission on the folder or cabinet where you are storing the object if the repository is running under folder security.
- If you used a Fetch method to obtain the object, use a Save method to save the changes to the repository.
You must have Write permission on the virtual document to save the changes you made.
- If you used a Checkout method to obtain the object, use a Checkin method to save your changes to the repository.
You must have at least Version permission on the virtual document to use Checkin. If the repository is running under folder security, you must also have Write permission on the object's primary cabinet or folder.

Modifying virtual documents

If a virtual document is not frozen, you can modify it by adding or removing components, changing component order, or changing the document's assembly or copy behavior.

To save your changes to the repository, you must have either Version or Write permission on the virtual document. Version permission is required if you intend to use a Checkin method to save your changes. Write permission is required if you intend to use a Save method to save your changes. If the repository is running under folder security, you also need Write permission on the virtual document's primary cabinet or folder.

Adding components

To add components, use an Appendpart or Insertpart method. Appendpart adds components at the end of the document's list of components. Because it assigns ordering numbers automatically, you cannot use Appendpart if you are managing the order numbers.

Insertpart inserts components anywhere in the list of components. It allows you to specify the order number of the new components. Use Insertpart if you want to add a component between two existing components or if you are managing the order numbers.

Removing components

To remove a component from a virtual document, use a Removepart method.

Changing the component order

To change the component order in a virtual document, use a Removepart method and an Insertpart or Appendpart method.

To change a component's order:

1. Remove the component from the virtual document.
2. Insert the component in the new position or append the component at the end of the list of components.

Modifying assembly behavior

If a component is a virtual document, you can determine how the server chooses the component's descendants during assembly. The server's behavior is controlled by two attributes in the component's containment object: `use_node_ver_label` and `follow_assembly`. The values of these attributes are set initially when the component is added to the document. To change their values, and the server's subsequent behavior, use an Updatepart attribute. For information about how these attributes affect assembly behavior, refer to [Defining component assembly behavior, page 167](#).

Modifying copy behavior

How a component is handled when the containing document is copied is determined by the `copy_child` attribute in the component's associated containment object. This attribute is set initially when the component is added to the virtual document. To reset this attribute, use an Updatepart method.

For information about valid settings for the attribute, refer to [Defining copy behavior](#), page 169.

Assembling a virtual document

Typically, users work on individual parts of a virtual document, retrieving them from the repository as needed. However, eventually, they need to work with the parts as one document. The process of selecting individual components to produce a virtual document is called assembling a virtual document.

In the context of an application, assembling a virtual document has two steps:

1. Use a `SELECT` statement to retrieve the object IDs of the components from the repository.

Which component objects are selected depends on how the objects are bound to the virtual document, the criteria specified in the `SELECT` statement (including the `IN DOCUMENT` clause's late binding condition, if any), and, for those components that are themselves virtual documents, how their assembly behavior is defined. (Refer to [Defining component assembly behavior](#), page 167, for information about defining assembly behavior for virtual documents.)

2. Use the object IDs in the client application to get the components from the repository.

After you have obtained the components from the repository, the application can manipulate the components as needed.

Selecting components

The server uses the `SELECT` statement you define, in conjunction with the values of two attributes in the components' containment objects (`use_node_ver_label` and `follow_assembly`), to determine which components to include in the virtual document.

Using the `SELECT` statement's `SEARCH` and `WHERE` clauses and the `WITH` option in the `IN DOCUMENT` clause, you can assemble documents based on current business rules, needs, or conditions.

For example, perhaps your company has an instruction manual that contains both general information pertinent to all operating systems and information specific to particular operating systems. You can put both the general information and the operating system-specific information in one virtual document and use conditional assembly to assemble manuals that are operating system specific.

The following SELECT statements use a WHERE clause to assemble two operating-system specific manuals, one UNIX-specific, and the other VMS-specific:

```
SELECT "r_object_id" FROM "dm_document"  
IN DOCUMENT ID('0900001204800001') DESCEND  
WHERE ANY "keywords" = 'UNIX'
```

```
SELECT "r_object_id" FROM "dm_document"  
IN DOCUMENT ID('0900001204800001') DESCEND  
WHERE ANY "keywords" = 'VMS'
```

Notice that the virtual document identified in both IN DOCUMENT clauses is the same. Each SELECT searches the same virtual document. However, the conditions imposed by the WHERE clause restrict the returned components to only those that have the keyword UNIX or the keyword VMS defined for them. ([keywords attribute, page 133](#), has information about associating keywords with individual documents.)

The `use_node_ver_label` and `follow_assembly` attributes affect any components that are themselves virtual documents. Both control how Content Server chooses the descendants of such components for inclusion. For information about how [Defining component assembly behavior, page 167](#).

SELECT processing

This section describes the algorithm Content Server uses to process a SELECT statement to assemble a virtual document. The information is useful, as it will help you to write a SELECT statement that chooses exactly the components you want.

Content Server uses the following algorithm to process a SELECT statement:

1. The server applies the criteria specified in the SEARCH and WHERE clauses to the document specified in the IN DOCUMENT clause. The order of application depends on how you write the query. By default, the SEARCH clause is applied first. When a document meets the criteria in the first clause applied to it, the server tests the document against the criteria in the second clause. If the document does not meet the criteria in both clauses, the SELECT returns no results.

Note: Refer to [Full-text searching and virtual documents, page 348](#), for information about full-text indexing and virtual documents

2. The server applies the criteria specified in the SEARCH and WHERE clauses to each direct component of the virtual document. The order of application depends on how you write the query. By default, the SEARCH clause is applied first. When a component meets the criteria in the first clause applied to it, the server tests it against the criteria in the second clause. If a component does not meet the criteria in both clauses, it is not a candidate for inclusion.

If a component is late bound, the SEARCH and WHERE clauses are applied to each version of the component. Those versions that meet the criteria in both clauses are candidates for inclusion.

3. The binding condition in the WITH option is applied to any versions of late-bound components that passed Step 2.

It is possible for more than one version to meet the condition specified by the WITH option. In these cases, the server uses the NODESORT BY option to select a particular version. If NODESORT BY option is not specified, the server includes the version having the lowest object ID by default.

4. If the DESCEND keyword is specified, the server examines the descendants of each included component that is a virtual document. It applies the criteria specified in the SEARCH and WHERE clauses first.

For late-bound descendants, the SEARCH and WHERE clauses are applied to each version of the component. Those versions that meet the criteria are candidates for inclusion.

5. For late-bound descendants, the server selects the version to include from the subset that passed Step 4. The decision is based on the values of use_node_ver_label in the containment objects and the binding condition specified in the WITH option of the IN DOCUMENT clause.

The resulting set of components comprises the assembled document.

The WITH option and the SEARCH and WHERE clauses are optional. For example, if you do not specify the WITH option and the search encounters any late-bound components, the server takes the version having the lowest object ID or, if NODESORT BY is specified, whichever is first in the sorted order.

If you include an IN ASSEMBLY clause instead of an IN DOCUMENT clause, the server applies the SEARCH and WHERE clauses, as described above, to the components found in the specified snapshot. Similarly, if you include the USING ASSEMBLIES option in the IN DOCUMENT clause or if the component has follow_assembly set to TRUE, when the server finds a component that has a snapshot, it applies the SEARCH and WHERE clauses to the objects in the component's snapshot rather than recursively searching the component's hierarchy.

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 a virtual document's components. Creating a snapshot of that subset of components lets you assemble the set more 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.

Creating snapshots

To create a snapshot for a virtual document, you must have at least Version permission for the virtual document. Creating a snapshot is a four-step process. Those steps are:

1. Use an Assemble method to select the components for the snapshot and place them in a collection.

Assemble methods generate a SELECT statement from the values you provide in the argument list.

Assemble arguments also define the document to which the snapshot is assigned and how many components are processed in each iteration of the Next method ().

2. Execute a Getlastcoll method to obtain the ID of the collection holding the components.

3. A Getlastcoll method returns the collection ID of the most recently generated collection.

4. Execute a Next method to generate assembly objects for the components.

The interrupt_freq argument in the Assemble method determines how many components are processed in each iteration of the Next method.

5. When the Next method returns a NULL value, execute a Close method to close the collection.

A Next method returns NULL when assembly objects have been created for all the objects in the collection. When that occurs, use a Close method to close the collection and complete the snapshot's creation. If you close the collection before all the components have been processed (that is, before assembly objects have been created for all of them), the snapshot is not created.

Because an Assemble method opens and manages its own transaction, you cannot issue an Assemble method or create a snapshot while an explicit transaction is open. (An explicit transaction is a transaction that a user opens with a Begintran method call or a BEGIN TRAN DQL statement.)

For an example of code that creates a snapshot, refer to [Assemble, page 100](#), in the *Content Server API Reference Manual*.

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.

Adding new assembly objects

To add an assembly object to a snapshot:

1. Create an assembly object.
2. Set the object's attributes.

You must set the following attributes:

- `book_id`, which identifies the topmost virtual document containing this component. Use the document's object ID.
- `parent_id`, which identifies the virtual document that directly contains this component. Use the document's object ID.
- `component_id`, which identifies the component. Use the component's 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 attribute 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.

3. Save the new object.

The Save operation fails if any of the objects identified by `book_id`, `component_id`, or `comp_chronicle_id` do not exist.

You can add components that are not actually part of the virtual document to the document's snapshot. However, doing so does not add the component to the virtual document in the repository. That is, the virtual document's `r_link_cnt` attribute is not incremented and a containment object is not created for the component.

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's `book_id` attribute) to delete the 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. To delete a snapshot, use the information in [Deleting a snapshot, page 178](#).

Changing an assembly object

You can change the values in the attributes 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's IN DOCUMENT clause.)

To change an assembly object:

1. Fetch the assembly object from the repository.
2. Make the changes.
3. Save the assembly object.

Deleting a snapshot

Use a 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.

Freezing virtual documents and snapshots

Freeze methods allow you to freeze a virtual document and a snapshot associated with that document. When you issue a Freeze method against a virtual document, if there is a snapshot associated with that document, you have the option to freeze the snapshot also. If a virtual document has multiple snapshots, only the snapshot actually associated with the virtual document itself is frozen. (The other snapshots, associated with simple documents, are not frozen.)

Users cannot modify the content or attributes of a frozen virtual document or the frozen snapshot components. Nor can they add or remove snapshot components.

However, 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's content files or attributes.

Freezing a document

Use a Freeze method to freeze a virtual document or another document with an associated snapshot. Freezing sets the following attributes of the document to TRUE:

- `r_immutable_flag`

This attribute indicates that the document is unchangeable.

- `r_frozen_flag`

This attribute indicates that the `r_immutable_flag` was set by a Freeze method (instead of a Checkin method).

If you chose to freeze an associated snapshot, the `r_has_frzn_assembly` attribute is also set to TRUE.

Freezing a snapshot sets the following attributes for each component in the snapshot:

- `r_immutable_flag`
- `r_frzn_assembly_cnt`

The `r_frzn_assembly` count attribute contains a count of the number of frozen snapshots that contain this component. If this attribute is greater than zero, you cannot delete or modify the object.

When you execute a Freeze method to freeze a snapshot, the document with which the snapshot is associated is also frozen automatically. To freeze only the snapshot and not the document, execute a Freeze method and include the argument to freeze the snapshot. Then execute an Unfreeze method to unfreeze only the document.

Unfreezing a document

Unfreezing a document makes the document changeable again.

Unfreezing a virtual document sets the following attributes 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 attribute to FALSE. The document remains unchangeable even though it is unfrozen.

- `r_frozen_flag`

If you chose to unfreeze the document's snapshot, the server also sets the `r_has_frzn_assembly` attribute to FALSE.

Unfreezing a snapshot resets the following attributes 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 attribute.

- `r_frzn_assembly_cnt`

This attribute, which contains a count of the number of frozen snapshots that contain this component, is decremented by 1.

Obtaining information about virtual documents

This section describes how to query a virtual document and how to obtain a path through a virtual document to a particular component.

Querying virtual documents

To query a virtual document, use DQL just as you would to obtain information about any other object. Documentum provides an extension to the SELECT statement that lets you query virtual documents to get information about their components. This extension is the IN DOCUMENT clause. Used in conjunction with the keyword DESCEND, this clause lets you:

- Identify all components contained directly or indirectly in a virtual document
- Assemble a virtual document

Use the IN DOCUMENT clause with the ID scalar function to identify a particular virtual document in your query. (The ID function is described in the *Content Server DQL Reference Manual*.) The keyword DESCEND directs the server to search the virtual document's full hierarchy.

Note: The server can search only the descendants of components that reside in the local repository. If any components are reference links, the server cannot search the descendants of the referenced documents.

For example, suppose you want to find every direct component of a virtual document. The following SELECT statement does this:

```
SELECT "r_object_id","object_name" FROM "dm_sysobject"  
IN DOCUMENT ID('virtual_doc_id')
```

This second example returns every component including both those that the document contains directly and those that it contains indirectly.

```
SELECT "r_object_id" FROM "dm_sysobject"
IN DOCUMENT ID('virtual_doc_id') DESCEND
```

The VERSION clause lets you find the components of a specific version of a virtual document. The server searches the version tree that contains the object specified in *virtual_doc_id* and uses (if found) the version specified in the VERSION clause. For example:

```
SELECT "r_object_id" FROM "dm_sysobject"
IN DOCUMENT ID('virtual_doc_id') VERSION '1.3'
```

Obtaining a path to a particular component

If you are writing Web-based applications, the ability to determine a path to a document within a virtual document is very useful. One attribute (*path_name*) and two methods (*Vdmpath* and *Vdmpathdql*) provide this information.

The *path_name* attribute

The *path_name* attribute is defined for the assembly object type. When you create a snapshot for a virtual document, the processing automatically sets each assembly object's *path_name* attribute to a list of the nodes traversed to arrive at the component represented by the assembly object. The list starts with the top containing virtual document and works down to the component. Each node is represented in the path by its object name, and the nodes are separated with forward slashes.

For example, suppose that *Mydoc* is a virtual document and that it has two directly contained components, *BrotherDoc* and *SisterDoc*. Suppose also that *BrotherDoc* has two components, *Nephew1Doc* and *Nephew2Doc*.

If a snapshot is created for *Mydoc* that includes all the components, each component will have an assembly object. The *path_name* attribute values for these assembly objects would be:

Component	Assembly object <i>path_name</i> value
Mydoc	Mydoc
BrotherDoc	Mydoc/BrotherDoc
Nephew1Doc	Mydoc/BrotherDoc/Nephew1Doc

Component	Assembly object path_name value
Nephew2Doc	Mydoc/BrotherDoc/Nephew2Doc
SisterDoc	Mydoc/SisterDoc

The path_name attribute is set during the execution of the Next method during assembly processing. (Refer to [Creating snapshots, page 176](#), for details about assembly processing.) If the path is too long for the attribute's length, the path is truncated from the end of the path.

Because a component can belong to multiple virtual documents, there may be multiple assembly objects that reference a component. Use the assembly object's book_id attribute to identify the virtual document in which the path in path_name is found.

Vdmpath and Vdmpathdql methods

Vdmpath and Vdmpathdql methods return the paths to a document as a collection. Both methods have arguments that tell the method you want only the paths found in a particular virtual document or only the shortest path to the document.

The Vdmpathdql method provides the greatest flexibility in defining the selection criteria of late-bound versions found in the paths. Vdmpathdql also searches all components in the paths for which the user has at least Browse permission.

With Vdmpath, you can only identify version labels as the selection criteria for late-bound components in the paths. Additionally, Vdmpath searches only the components to which World has at least Browse permission.

For details about the syntax and return values of these methods, refer to [Vdmpath, page 501](#), and [Vdmpathdql, page 506](#).

Renditions

This chapter describes how Documentum stores a content file in a variety of formats, called renditions. Renditions can be generated through converters supported by Content Server or through operations of Documentum Media Transformation Services, an optional product.

The following topics are included in this chapter:

- [What a rendition is, page 183](#), describes what a rendition is.
- [Creating renditions, page 184](#), describes how to create a rendition in an alternate format.
- [Rendition formats, page 185](#), tells you how to determine what the valid formats are.
- [Rendition characteristics, page 186](#), describes the resolution characteristics, encapsulation characteristics, and transformation loss characteristics of renditions, and describes how to compose a full format specification.
- [Adding and removing renditions, page 190](#), describes how to add and remove a rendition that you create outside of the Documentum system.
- [Determining the keep_flag setting, page 190](#)
- [Implementing an alternate converter, page 194](#), describes how to implement a format converter that you have purchased and how you can manage transformations explicitly by providing your own program to perform the transformation.

What a rendition is

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 Content Server or through Documentum Media Transformation Services, an optional product that handles rich media formats such as jpeg and audio and video formats.

Converter support

Content Server support for content converters allows you to:

- Transform one word processing file format to another word processing file format.
- Transform one graphic image format to another graphic image format.
- Transform one kind of format to another kind of format—for example, changing a raster image format to a page description language format.

All the work of transforming formats is carried out by one of the converters supported by Content Server. Some of these converters are supplied with Content Server, others must be purchased separately. If you want to use a converter that you have written or one that is not on our current list of supported converters, you can do so. For information about using a converter that you have written or that is not on our supported list, refer to [Implementing an alternate converter, page 194](#).

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

Media Transformation Services renditions

If you have installed Documentum Media Transformation Services, each time a user creates and saves a document with a rich media format, Media Server creates two renditions of the content:

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

In addition, the user or an application can issue a `TRANSCODE_CONTENT` administration method to request additional renditions. Information about Documentum Media Transformation Services and how to create renditions using `TRANSCODE_CONTENT` is found in *Administering Documentum Media Transformation Services*. Reference information for `TRANSCODE_CONTENT` is found in the *Content Server DQL Reference Manual*.

Creating renditions

Note: This section describes only how to create a rendition using a converter. For instructions on creating renditions through Documentum Media Transformation Services, refer to the *Administering Documentum Media Transformation Services* manual.

Creating a rendition using a converter is as simple as using a `Getfile` method. When you used `Setfile` to associate a particular content file with a document, you specified a format for that file. To create a rendition, you specify an alternate format when you issue a

Getfile method to access the content file. Content Server automatically transforms the content from the original format to the alternate format.

To illustrate, look at the following two commands. The first command associates the content file `brd_rcps` (for bread recipes) with the document specified by the `doc_id`. This content file has the format of plain ASCII text. The second command gets the content file for the user and creates a new rendition at the same time, by specifying the format maker (for FrameMaker files) in the Getfile command.

```
dmAPISet("setfile,s0,doc_id,c:\brd_rcps.txt,text")
. . .
dmAPIGet("getfile,s0,doc_id,c:\brd_rcps.mak,maker")
```

Note: The above examples assume that the user is working on a Windows platform..

Content Server's 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 file's current 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.

Rendition formats

A rendition's 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 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.

Content Server is installed with a wide range of formats. Installing Documentum Media Transformation Services provides an additional set of rich media formats. You can modify or delete the installed formats or add new formats. Refer to the *Content*

Server Administrator's Guide for instructions on obtaining a list of formats and how to modify or add a format.

Rendition characteristics

Documentum stores a set of information about each rendition in the content object. The set 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.

Generally, when you specify a format in a method command, you specify only its file format. However, in some instances you may want to specify a full format. This section contains information about the parts of a full format specification. For instructions about actually specifying a full format, refer to [Reading and composing a full format specification, page 189](#).

Resolution characteristics

For each rendition, the server stores information about the rendition's resolution characteristics. This information consists of one or more resolution names and their associated values and a file format specification. [Table 8-1, page 186](#), shows the three valid resolution names.

Table 8-1. Resolution names

Name	Description	Example
PX	Pixel extent (height and width in pixels)	PX:2250:3300
PZ	Pixel depth in bits/pixel	PZ:1
RS	Resolution in pixels/inch	RS:300

The resolution information is specified as a string having the following format:

```
resolution_name:value[:value]
{.resolution_name:value[:value]}.file_format
```

Note that only the PX resolution name allows you to specify more than one value.

Using the table above, the resolution information for a scanned image, stored in tiff format might look like:

```
RS:300.PX2250:3300.tiff
```

The names and their values can appear in any order in the string. The file format specification must appear last.

Encapsulation characteristics

There are some data file formats that re-encode data without changing the data's logical structure or content. These formats encapsulate the data. An example of this type of format is the .Z Lempel-Ziv data compression format used by the UNIX compress utility. Another encapsulation format is provided by the UNIX uuencode utility, which represents binary data as ASCII characters.

Encapsulations are independent of the type of data being encoded.

Each type of encapsulation is represented by a name, stored as an ASCII text string. [Table 8-2, page 187](#), shows some sample encapsulation names.

Table 8-2. Encapsulation names

Name	Description
Z	Data compressed with UNIX compress utility
U	Binary data encoded as ASCII with UNIX uuencode utility
CR	ASCII with CR (carriage return) as the end of the line (instead of a line feed LF)

To indicate encapsulation, the server appends the encapsulation name to the end of the file format specification using a plus (+) sign. For example, the following designation represents a troff file that has been compressed with UNIX compress utility and then encoded as ASCII with the UNIX uuencode utility:

```
troff+Z+U
```

Transformation loss characteristics

When you create a rendition, the server uses an evaluation algorithm to chain together transformations to convert from one format to another. This algorithm uses a list of transformations and their associated loss values to calculate the best conversion

available. The loss values are part of the convert.tbl file. [Table 8-3, page 188](#), shows some of the transformation loss information in this file.

Table 8-3. Sample transformations and their loss values

Transformation name	Description	Loss index
t	No loss conversion	00001
rcv	Raster conversion (pixels->pixels)	00001
pgn	Pagination	00001
ret	Retrieve from stored remotely	00001
z	Uncompress	00001
bad	Not as good as other options	00005
pdl	Page description language (structured->PDL)	00011
pl	Extract first page	00101
pg	Extract page	00111
scv	Scan conversion (PDL->pixels)	00101
rpr	Raster processing (pixels->pixels)	00101
lcv	Low-loss WP conversion (structured->structured)	00011
cnv	Generic WP conversion (structured->structured)	00101
xcv	High-loss WP conversion (structured->structured)	01001
txt	High-loss WP text extraction (structured->structured)	01006
ocr	Character recognition (pixels->text)	10001
scr	Scan conversion + OCR	10001

The server stores the transformation path for the rendition with the content object. A transformation path has the following format:

```
T:transformation_name[:transformation_name]
```

For example, perhaps a particular text rendition is the result of the following transformations:

page extraction (paper)->scanned (pixels)->character recognition (text)

Its transformation path would be:

```
T:pg:scv:ocr
```

By looking at the transformation path, you can estimate how close the rendition is to the original document. You do this by adding the loss value for each part of the path. The higher the resulting figure, the farther the rendition is from the original. Note that a transformation loss of 1 indicates that there is no loss. (Refer to [Reading and composing a full format specification, page 189](#), for information about how to query the content object to see the transformation paths.)

Reading and composing a full format specification

A full format specification has the format:

```
T:transformation_path.resolution_characteristics.  
file_format+encapsulation_characteristics
```

The server stores the full format specifications in the `dmr_content` type in the `full_format` attribute. To see the specifications for the renditions of a particular document, use the following SELECT statement:

```
SELECT "full_format" FROM "dmr_content"  
WHERE "parent_id" = 'document_id'
```

The `document_id` is the object ID of the document object.

If you want to find the format specification for a particular content file, use this SELECT, substituting the appropriate content ID:

```
SELECT "full_format" FROM "dmr_content"  
WHERE "r_object_id" = 'content_id'
```

Generally, when you retrieve a content file, you specify only the file format in which you want to see the data and the server picks the optimal transformation path and rendition characteristics. However, if you like, you can specify exact rendition characteristics and the transformation path. To do this, you specify a full format specification in the argument list when you issue the `Getfile` method to retrieve the content file.

To illustrate, the following command retrieves a file using a full format specification:

```
dmAPIGet("getfile,s0,obj_id,c:\recipe,  
T:pdl:scv.RS:300.PX:2250:3300.PZ:1.tiff")
```

In some instances, you may want to specify a range value for the resolution attributes instead of a specific figure. For example, a range might be more appropriate when requesting a low-resolution thumbnail sketch. To specify a resolution range, use the following syntax in the resolution specification:

```
resolution_name:value[value_range]
```

For example:

```
RS:9[8-10]
```

And:

```
PX:150[50-150]:150[50-150]
```

If you specify range values for the PX (height and width) resolution characteristic, the values are not used independently of each other. That is, the ratio between the height and width that is established by the specified value is maintained if a range value is used instead. To illustrate, the previous example specified both the height and width as 150. The specified height and width are equal. Consequently, if the value of the height changes, the system will change the value of the width to equal the value of the height (and vice versa). The system maintains the given proportion between the height and width.

Adding and removing renditions

At times you may want to use a rendition that can't be generated by a Documentum server. In such cases, you can create the file outside of Documentum and add it to the document using an `Addrendition` method.

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

Determining the `keep_flag` setting

When a rendition is added using an `Addrendition` method, a method argument called `keep_flag` allows you to define how you want to handle the rendition when the containing object is versioned. If you set the `keep_flag` argument to `T` (`TRUE`) in the method arguments, the rendition is kept with the object when the object is versioned. If the argument is set to `F` (`FALSE`), the rendition is not carried forward with the new version.

If the `keep_flag` argument is set to `T`, the rendition attribute in the content object representing the rendition is set to 3. To determine whether the `keep_flag` is set to `T` or `F`, examine that attribute. If it is set to 3, the flag was set to `T`. If the attribute is set to either 1 or 2, the flag was set to `F` when the rendition was added to the object.

Supported conversions on Windows platforms

Content Server supports conversions between the three types of ASCII text files. [Table 8-4, page 191](#), lists the acceptable ASCII text input formats and the obtainable output formats.

Table 8-4. Supported input and output formats for automatic conversion

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

Supported conversions on UNIX platforms

On UNIX, Content Server supports format conversion by using the converters in the `$DM_HOME/convert` directory.

The Content Server currently provides the following converters:

- PBMPLUS image conversion (refer to [PBM image converters, page 192](#))
- Miscellaneous file format conversions (refer to [Miscellaneous converters, page 193](#))

The convert directory contains the following directories:

filtrix	sandpiper
kurzweil	scripts
pmbplus	soundkit
pdf2text	troff
psify	

You can also purchase and install document converters. The following section, [Implementing an alternate converter, page 194](#), contains instructions for this. Documentum 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., 260 Petaluma Avenue, Sebastopol, CA 95472, and purchase a separate license.

You can also purchase and install Frame converters from Adobe Systems Inc., 345 Park Avenue, San Jose, CA 95110-2704. If you install the Frame converters in the Content Server's bin path, the converters are incorporated automatically when you start the Documentum system. The server assumes that the conversion package is found in the UNIX bin path of the server account and that this account has the FMHOME environment variable set to the FrameMaker home.

PBM image converters

To transform images, the servers 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.

[Table 8-5, page 192](#), lists the acceptable input formats and [Table 8-6, page 193](#), lists the acceptable output formats for the PBMPLUS package.

Table 8-5. Acceptable input formats for PBMPLUS converters

Input format	Description
gem	Digital Research image file
gif	General Interchange Format
macp	MacPaint file
pcx	PCPaint file (Windows)
pict	Macintosh standard graphics file

Input format	Description
rast	SUN raster image file
tiff	TIFF graphic file
xbm	xbitmap file (x.11 Windowing system definition)

Table 8-6. Output formats of the PBMPLUS converters

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

Miscellaneous converters

The Content Server also uses UNIX utilities to provide some miscellaneous conversion capabilities. These utilities include tools for converting to and from 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.

[Table 8-7, page 194](#), lists the acceptable input formats and [Table 8-8, page 194](#), lists the acceptable output formats for these tools.

Table 8-7. Acceptable input formats for UNIX conversion utilities

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

Table 8-8. Output formats of UNIX conversion utilities

Output format	Description
crtext	ASCII text file with carriage return line feed (for PCs)
ps	PostScript file
text	UNIX text file

Implementing an alternate converter

Perhaps you want to use a converter that is not on our list of currently supported converters. Or maybe you would like to write your own converter and use that instead of Documentum's transformation engine. Either of these options is possible.

Server config objects have an attribute called `user_converter_location`. If you want to use an alternate converter, set this attribute to the name of the location object that identifies the directory where the executable or script for the alternate converter is found.

If `user_converter_location` has a value, whenever a `Getfile` or `Print` command requires a transformation, the server attempts to use the converter script specified in `user_converter_location` before invoking the built-in transformation engine. The converter script is called through an operating system system API call. The syntax for this is:

```
system("converter repository user ticket document_id object_type  
format output_path")
```

The arguments have the following meanings:

<i>repository</i>	repository to which the server is connected
<i>user</i>	Username of the currently connected user
<i>ticket</i>	Ticket value obtained by a Getlogin method
<i>document_id</i>	Object ID of the document that you want to convert
<i>object_type</i>	The object type of the document
<i>src_format</i>	The current format of the document
<i>new_format</i>	Format to which you want to convert the document
<i>output_path</i>	The directory where you want the convertor to put the converted file.

The server issues the system call, substituting the value specified in `user_converter_location` for the converter argument and providing all the values for the other arguments as well. Your alternate converter script may or may not use the values in the other arguments. The arguments are intended to provide enough information for the alternate converter so it can make a decision about whether it can perform the requested transformation.

Content Server expects the converter script to return `ENOSYS` (as defined by your operating system) if the converter cannot handle the transformation. If the converter is successful, the server expects the converter to return the converted file's file path to standard output and to exit with 0.

Workflows

This chapter describes workflows, part of the process management services of Content Server. Workflows allow you to automate business processes. The following topics are included:

- [Introducing workflows, page 197](#)
- [Workflow definitions, page 201](#)
- [Validation and installation, page 226](#)
- [Architecture of workflow execution , page 228](#)
- [Package notes, page 232](#)
- [Attachments, page 233](#)
- [The workflow supervisor, page 233](#)
- [The workflow agent, page 234](#)
- [The enable_workitem_mgmt key, page 234](#)
- [Instance states, page 235](#)
- [Starting a workflow, page 238](#)
- [How execution proceeds, page 238](#)
- [User time and cost reporting, page 253](#)
- [Reporting on completed workflows, page 254](#)
- [Changing workflow, activity instance, and work item states, page 255](#)
- [Modifying a workflow definition, page 257](#)
- [Destroying process and activity definitions, page 260](#)
- [Distributed workflow, page 260](#)

Introducing workflows

A workflow formalizes a business process such as an insurance claims procedure or an engineering development process. The business process is formalized as a workflow definition stored in the repository. When users want to perform the business process,

they create a runtime instance of the workflow, based on the definition. Users can repeatedly perform the business process, and because it is based on a stored definition, the essential process is the same each time. Additionally, separating a workflow's definition from its runtime instantiation allows multiple workflows based on the same definition to be run concurrently.

Workflows can describe simple or complex business processes. A workflow's 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. Or, 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.

Implementation

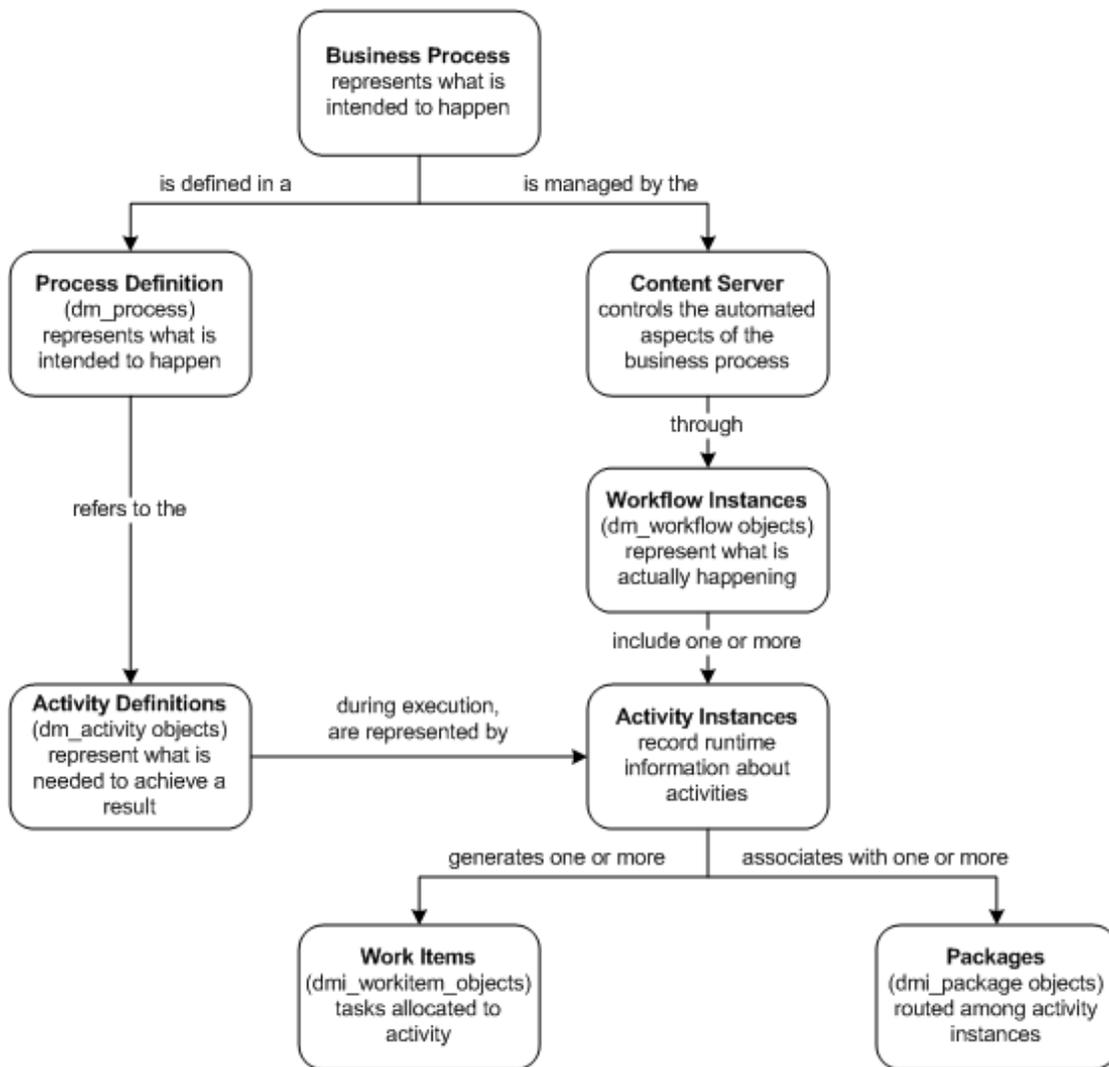
A workflow is implemented as two separate parts: a workflow definition and a runtime instantiation.

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 you start 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 attributes 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.

[Figure 9-1, page 199](#), illustrates how the components of a workflow definition and runtime instance work together.

Figure 9-1. Components of a workflow



For more information about workflow definitions, including activity, port, and package definitions, refer to [Workflow definitions, page 201](#).

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 Content 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 Desktop Client (through the File menu) and Webtop (through the Tools menu).

For more information about using aliases in workflows, refer to [Using aliases as performer names, page 212](#).

Business Process Manager and Workflow Manager

Content Server supports two user interfaces for creating and managing workflows: Workflow Manager and Business Process Manager.

Workflow Manager (WFM) supports basic workflow functionality. Business Process Manager (BPM) supports the basic functionality and additional features not supported by WFM, but requires an additional license. The features available using BPM that are not available using WFM are:

- Global package definitions
- Enhanced workflow timer capabilities
- Work queues, to help manage work items
- XPath specifications in activity transition conditions
- Email templates for workflow events
- Ability to add attachments to a running workflow

This chapter describes basic functionality. The additional features supported by BPM are called out and described in the appropriate sections. However, complete descriptions of their use and implementation are found in the Business Process Manager documentation.

Workflow definitions

A workflow definition consists of one process definition and a set of activity definitions. This section provides some basic information about the components of a definition.

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 attributes that identify the activities that make up the business process and a set of attributes that define the links connecting the activities. It also has attributes that define some behaviors for the workflow when an instance is running.

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's structure.

The activities included in a process definition are identified in the definition as either Begin activities, the End activity or Step activities. Begin activities are the first activities in the workflow. A process definition must have at least one beginning activity. An End activity is the last activity in the workflow. A process definition can have only one ending activity. Step activities are the intermediate activities between the beginning and the end. A process definition can have any number of Step activities.

Multiple use

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.

Whether an activity may be used more than once in a workflow is determined by the activity's definition. For more information, refer to [Repeatable activities, page 205](#).

How activities are referenced in workflows

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` attribute. 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` attribute). However, if the activity is used more than once in the workflow, providing a unique name for each use is required.

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.

Note: Input ports on Begin activities and output ports on End activities are not allowed to participate in links.

Each link in a process definition has a unique name.

Activity definitions

Activity definitions describe tasks in a workflow. Documentum implements activity definitions as `dm_activity` objects. The attributes 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 attributes that define the ports for the activities and the packages that each port can handle.

Manual and automatic activities

An activity can be either manual or automatic. A manual activity represents a task performed by an actual person or members of a group. An automatic activity represents a task whose work is performed, on behalf of a user, by a script defined in a method object. The script can be executed on the user's behalf by Content Server, the dmbasic method server, or the Java method server.

Manual activities can allow delegation or extension. Automatic activities cannot. (Delegation and extension are described in [Delegation and extension, page 204](#).)

Any user can create a manual activity. You must have Sysadmin or Superuser privileges to create an automatic activity. (For instructions on creating a method for an automatic activity refer to [Implementing a method, page 134](#), in the *Content Server Administrator's Guide*.)

Activity priorities

Priority values are used to designate the execution priority of an activity. When you create a workflow definition in either WFM or BPM, you can set a priority for each activity in the workflow. The value chosen is recorded in the process definition, in the `r_act_priority` attribute. [Use of the priority defined in the process definition, page 203](#), describes how this priority value is used.

Work items generated by activities whose performer is any member of a work queue are assigned a priority value at runtime, when the work item is generated. [Use of the work queue priority values, page 204](#), describes priorities assigned in work queues.

Note: A work queue can be chosen as an activity performer only if the workflow definition was created in BPM.

Use of the priority defined in the process definition

The priority value recorded in the `r_act_priority` attribute in the process definition is only applied to automatic tasks. Content Server ignores the value for manual tasks.

The workflow agent (the internal server facility that controls execution of automatic activities) uses these priority values to determine the order of execution for automatic activities. When an automatic activity is instantiated, Content 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.

You can change the priority value of a particular work item generated from an activity at runtime using a `Setpriority` method.

Use of the work queue priority values

In BPM, you can set up work queues to automate the distribution of manual tasks to appropriate performers. (For more information about work queues, refer to the BPM documentation or online Help.) Every work item on a work queue is governed by a workqueue policy object. The workqueue 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.

The priority assigned by a workqueue policy does not affect or interact with a priority value assigned to an activity in the process definition. Workqueue 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 Content Server only for execution of automatic activities.

For more information about how the workqueue policy is handled at runtime, refer to BPM documentation.

Delegation and extension

Delegation and extension are features that you can set for manual activities. Delegation allows the server or the activity's performer to delegate the work to another performer. Extension allows the activity's performer to identify a second performer for the activity after he or she completes the activity the first time.

Delegation

If delegation is allowed, it can occur automatically or be forced manually.

Automatic delegation occurs when the server checks the availability of an activity's 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` attribute of the original performer's user object.

If there is no user identified in `user_delegation` or that user is not available, automatic delegation fails. When delegation fails, Content Server reassigns the work item based on the value in the `control_flag` attribute 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 a Delegate method is explicitly issued. Typically, only the work item's performer, the workflow supervisor, or a Superuser can execute a Delegate method. However, if the `enable_workitem_mgmt` key in the `server.ini` file is set to T (TRUE), any user can issue a Delegate method to delegate any work item.

If delegation is disallowed, automatic delegation is prohibited. However, the workflow supervisor or a Superuser can delegate the work item manually.

Extension

If extension is allowed, when the original performers complete an activity's 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's 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 a 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. ([Table 9-1, page 206](#), describes the user categories for performers.)

Repeatable activities

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` attribute controls this feature. It is TRUE by default. To constrain an activity's use to only once in a workflow's structure, the attribute must be set to FALSE.

Performer choices

When you define a performer for an activity, you must first choose a performer category. Depending on the chosen category, you may 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

[Performer categories, page 206](#), describes the choices for performer categories. [Defining the actual performer, page 211](#), describes the options for when the actual performer is chosen.

Performer categories

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. [Table 9-1, page 206](#), lists the performer categories and whether they are supported for manual or automatic activities or both. Internally, each choice is represented by an integer value stored in the activity's `performer_type` attribute.

Table 9-1. Performer categories for activities

User category (integer value)	How performers are selected	Valid category for
Workflow supervisor (0)	The server gets the supervisor's name from the workflow instance and assigns a new work item to the supervisor. If the supervisor has <code>workflow_disabled</code> set to <code>TRUE</code> , the server gives the work item to the supervisor's delegated user.	manual and automatic activities
Repository owner (1)	The server assigns a new work item to the repository owner. If the owner has <code>workflow_disabled</code> set to <code>TRUE</code> , the server gives the work item to the owner's delegated user. If the activity is an automatic activity, you must be a superuser to assign this performer.	manual and automatic activities

User category (integer value)	How performers are selected	Valid category for
Last performer (2)	<p>The server gets the performer from the last finished activity that satisfied the trigger condition of the current activity.</p> <p>If that performer has <code>workflow_disabled</code> set to <code>TRUE</code>, the server gives the work item to the user's delegated user.</p> <p>If the activity is an automatic activity, you must be a superuser to assign this performer.</p>	manual activities only
A user (3)	<p>The server assigns a new work item to the chosen user. Valid performers for this category are a user, an alias representing a user, or the keyword <code>dm_world</code>.</p> <p>If the user has <code>workflow_disabled</code> set to <code>TRUE</code>, the server gives the work item to the user's delegated user.</p> <p>If you specify <code>dm_world</code>, you must define the activity as an activity whose performer is chosen by either the workflow initiator or another activity's performer.</p> <p>If the activity is an automatic activity, you must be a superuser to assign this performer.</p>	manual and automatic activities
All members of a group (4)	<p>The server assigns a separate work item for each group member. Valid performers for this category are a group or an alias for a group name.</p> <p>The server does not give a work item to any group member who has <code>workflow_disabled</code> set to <code>TRUE</code>, nor does it give the work item to the group member's delegated user.</p>	manual activities only

User category (integer value)	How performers are selected	Valid category for
Single user in a group (5)	<p>The server assigns a new work item to every group member and allows any group member to acquire it. The server changes the work item's performer_name to the person who first acquires the work item and prevents anyone else from acquiring the work item.</p> <p>The server does not give a work item to any group member who has workflow_disabled set to TRUE, nor does it give the work item to the group member's delegated user.</p> <p>Valid performers for this category are a group name or an alias for a group name in the performer_name attribute.</p>	manual activities only
Single user in a group who is least loaded (6)	<p>The server determines which user in a group has the least workload by querying the dmi_workitem table and assigns a new work item to that user. Workload is measured as the number of dormant and active work items.</p> <p>The server does not give a work item to any group member who has workflow_disabled set to TRUE, nor does it give the work item to the group member's delegated user.</p> <p>Valid performers for this category are a group name or an alias for a group name in the performer_name attribute.</p>	manual activities only

User category (integer value)	How performers are selected	Valid category for
Some users in a group or some users in the repository (8)	<p>The server assigns a work item to each of the users in the group or repository who are chosen as performers. If a group name is chosen as a user, the server assigns one work item to the group and the first group member that acquires the work item becomes the performer.</p> <p>The server does not give a work item to any group member who has <code>workflow_disabled</code> set to <code>TRUE</code>, nor does it give the work item to the group member's delegated user.</p> <p>Valid performers for this category are user names, a group name, an alias that resolves to a group name, or the keyword <code>dm_world</code>.</p>	manual activities only
Some users in a group or some users in the repository, sequentially (9)	<p>The server assigns the work item to the first user in the group or repository who is chosen as a performer. When that user completes the work item, the server creates another work item for the next user in the list of chosen users. This continues until all chosen users have completed their work items.</p> <p>If a group name is chosen as a user, the server assigns one work item to the group and the first group member that acquires the work item becomes the performer.</p> <p>The server does not give a work item to any group member who has <code>workflow_disabled</code> set to <code>TRUE</code>, nor does it give the work item to the group member's delegated user.</p> <p>Valid performers for this category are user names, a group name, an alias that resolves to a group name, or the keyword <code>dm_world</code>.</p>	manual activities only

User category (integer value)	How performers are selected	Valid category for
	Activities with this performer type cannot be extensible.	
A user from a work queue (10)	<p>The server assigns a new work item to the work queue and by default, allows any of the queue users to acquire it.</p> <p>If the package associated with the work item has a required skill level set, then the work item may only be acquired by queue users who have a matching or higher skill level.</p> <p>This category of performer type is only available if you are using BPM to create and manage the workflow.</p>	manual activities only

Categories for manual activities

The performer of a manual activity can be selected from any of the user categories. If you want to define the actual performer when you create the activity, you can choose any of the user categories listed in [Table 9–1, page 206](#). If you want the actual performer to be selected at runtime, you must choose from categories 3 through 10.

Note: For categories 4, 5, and 6, Content Server requires a performer name or alias to successfully validate the activity definition. However, if you want the group to be chosen at runtime by the performer of a previous activity, the performer name value is ignored. Workflow Manager and BPM provide a dummy value (the user’s default group) for the performer in such cases.

Categories for automatic activities

The performer for automatic activities must resolve to single user. This requirement limits your choices for automatic activities to the following user categories:

- The workflow supervisor
- The repository owner
- A particular user

If you select either the workflow supervisor and repository owner, the server determines the actual user at runtime.

If you choose a particular user, you can define the actual user when you create the activity or use an alias, to allow the selection to occur at runtime.

Defining the actual performer

If you selected Workflow supervisor (0), Repository owner (1), or Previous performer (2) as the performer category, the actual user is defined by the category. For example, an executing workflow has only one workflow supervisor and the repository in which it executes has only one owner. It is not necessary to define the actual person (performer_name) either when you create the activity or later, at runtime. The server determines the actual performer when the activity is started.

If you selected any performer category from 3 through 10, you must select the actual performer from that category or configure the activity to allow selection at one of the following times:

- When the workflow is started
- When the activity is started
- When a previous activity is completed

Defining the actual performer in an activity definition is the least flexible structure. Allowing the performer of a previous activity to choose an activity's performer is the most flexible structure. Letting a previous performer choose an activity's performer lets decisions about performers be based on current circumstances and business rules.

At workflow initiation

This option allows the user who starts the workflow to choose a performer or performers for the activity. For this option, you must specify an alias for the performer when defining the activity. When the workflow is started, WFM or BPM will prompt the workflow's initiator to provide a user or group name for the alias.

At activity initiation

To configure actual performer selection at the time the activity is started, you must specify an alias for the performer name and tell Content Server in which alias set or sets to search for the alias.

When the activity is started, Content Server searches the specified alias set or sets for the alias and assigns the work item to the performer name that is matched with the alias.

At the completion of a previous activity

This option allows a performer of a previous activity to choose, at runtime, the activity's performer.

Choosing the same performer set for multiple manual activities

When you create a workflow in WFM or BPM, you can select multiple activities and make a performer choice that is applied to all the selected activities.

Using aliases as performer names

You use an alias in place of a performer name in an activity definition when you want the performer to be chosen by the workflow's initiator or by Content Server, when the activity is started. Using aliases creates a flexible activity definition that can be used in a variety of contexts.

For example, suppose you are creating a workflow with the following activities: Write, Review, Revise, Approve, and Publish. The Write, Review, and Revise performers will probably be different for different documents. By using an alias instead of an actual user or group name in those activities, you can ensure that the correct performer is selected each time the workflow is run.

The format of an alias specification is:

```
%[alias_set_name.]alias_name
```

alias_name is the alias for the activity's performer. The alias can represent a user or a group. For example, possible alias names might be Writer or Reviewer or Engineers.

alias_set_name is the name of an alias set object. Including an alias set name is optional. If you include it, the server looks for a match for the alias name value only within the specified alias set when it resolves the alias.

(For more information about aliases, refer to [Appendix A, Aliases](#).)

Constraints on aliases as performer names

You cannot use an alias as the performer name if you are choosing any of the following user categories:

- Workflow supervisor
- Repository owner
- Performer of previous activity

- Some users in the repository

Note: While you can use an alias if the performer category is 8 (Some users in a Group or Some Users in the Repository), Content Server assumes that you are selecting some users in a group, not the repository, and tries to resolve the alias to a group name.

Alias resolution in workflows

An alias is resolved by searching for the alias in an alias set. An alias set is an object whose attributes identify one or more aliases and, optionally, their associated values. If you are creating an activity whose performer is chosen by the workflow initiator, WFM or BPM creates the alias set for you and associates it with the workflow definition. That alias set contains the alias you specify. When a user starts a workflow using that definition, the user is prompted to provide an actual name for the alias.

If the performer is to be chosen when the activity starts, you instruct Content Server which alias set to search for the alias. The options are:

- Search a specific alias set that you identify when choosing the performer
- Search the alias set associated with a document in a specified or unspecified incoming package
- Search the alias set associated with the previous activity's performer

The alias sets represented by these options must contain both the aliases and their corresponding actual names. When the activity starts, the server will search the alias set to locate the alias and its corresponding performer name.

The server records the kind of resolution you choose in the `resolve_type` attribute of the activity definition. If you choose to use the alias set associated with a component package, you can specify a particular package. The package name is recorded in the `resolve_pkg_name` attribute of the activity definition. If you do not identify a particular package, at runtime, Content Server examines the components of each incoming package in the order in which the packages are defined in the activity's `r_package_name` attribute until a match is found.

If you choose to use the alias set associated with a previous performer, the server searches the alias set defined for the performer of the previous activity. If no match is found, the server searches the alias set defined for that performer's default group.

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 attributes. At runtime, the actual message is constructed by substituting the actual attribute 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 run time 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` attribute of the activity definition. The text can be up to 255 characters and can contain references to the following object types and attributes:

- `dm_workflow`, any attribute
- `dmi_workitem`, any attribute

At runtime, references to `dmi_workitem` are interpreted as references to the work item associated with the current task.

- `dmi_queue_item`. any attribute 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 attribute

The format of the object type and attribute references must be:

```
{object_type_name.attribute_name}
```

The server uses the following rules when resolving the string:

- The server does not place quotes around resolved object type and attribute references.
- If the referenced attribute is a repeating attribute, 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 attribute reference contains an error, for example if the object type or attribute 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` attribute of the task's associated queue item object. Once the server has created the queue item, the value of the `task_subject` attribute in the queue item will not change, even if the values in any referenced attributes change.

Starting conditions

A starting condition defines the starting criteria for an activity. At runtime, the server will not start an activity until the activity's 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.

For Step and End activities, the trigger condition must be a value between one and the total number of input ports. For Begin activities, the value is 0 if the activity has no input ports. If the Begin activity has input ports, then the trigger condition must be between one and the total number of input ports (just like Step and End activities).

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.

If you include a trigger event in the starting condition, the server must find the event queued to the workflow before starting the activity. The same event can be used as a trigger for multiple activities, however, the application must queue the event once for each activity. (The server examines the `dmi_queue_item` objects looking for the event.)

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 attributes in activity definitions. ([How activities accept packages, page 249](#), describes how the implementation actually moves packages from one activity to the next.)

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. There are three types of ports:

- Input
- Output
- Revert

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.

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.

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.

A Begin activity must have at least one output port but an input port is optional. If you include an input port on a Begin activity, your application must manufacture and pass the package to the port at runtime, using an `addPackage` method. Revert ports are also optional for begin activities.

All Step activities must have at least one input and one output port. However, because each port can participate in only one link, the actual number of input and output ports required by each activity in your workflow will depend on the structure of your workflow. For example, if Activity A sends packages to Activity B and Activity C, then Activity A requires two output ports, one to link with Activity B and one to link with Activity C. If Activities B and C only accept packages from Activity A, they will require only one input port each, to complete the link with Activity A.

An End activity must have at least one input port but an output port is optional. And End activity cannot have a revert port.

Package definitions

Documents are moved through a workflow as packages moving from activity to activity through the ports. Packages are defined in attributes 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, Content 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 an activity's route cases, the schema is used to validate the path. (Route cases are defined for automatic transitions for manual activities. For information about these, refer to [Transition types, page 219](#).) The XML file and the schema are associated using a relationship, represented by an object of type `dmc_wf_package_schema`. The `dmc_wf_package_schema` object type is a subtype of `dm_relation`.

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.

Empty packages

In BPM, you can define a package with no contents. This lets you design workflows that allow an activity performer to designate the contents of the outgoing package at the time he or she completes the activity. For more information about empty packages, refer to the BPM documentation or online help.

Scope of a package definition

If you create the workflow using Workflow Manager, a package definition is associated with the input and output port connected by the selected link (flow). In Workflow Manager, you must define the package or packages for each link in the workflow.

If you are using Business Process Manager (BPM) to create the workflow, a package definition is global. When you define a package in BPM, the definition is assigned to all input and output ports in all activities in the workflow. It is not necessary to define packages for each link individually.

Note: BPM allows you to choose, for each activity, whether to make the package visible or invisible to that activity. So, even though packages are globally assigned, if a package is not needed for a particular activity, you can make it invisible to that activity. When the activity starts, the package is ignored—none of the generated tasks will reference that package. For more information about defining packages in BPM, refer to the Business Process Manager documentation or BPM online help.

Required Skill Levels for Packages

Different tasks often require different skill sets from the users who perform the task. If you are creating manual tasks with a specific user or group designated as the performer, you can ensure that the designated performer has the required skill set. If the designated user is a work queue, it may be more difficult to ensure that the work queue user who acquires or is assigned the task has the required skill set. To ensure that tasks on work queues are acquired by users with the appropriate skill set, BPM allows you to set a required skill level for a package at runtime. When a work queue user pulls a task from the work queue, the user can only pull tasks whose packages have either no skill set defined or those whose required skill set matches the skill level defined in the user's user profile.

This feature is implemented using an automatic activity provided with BPM. For more information, refer to the BPM documentation.

Package compatibility

The package definitions associated with two ports connected by a link must be compatible. For example, suppose you define a link between Activity A and Activity B, with ActA_OP1 (Activity A output port 1) as the source port and ActB_IP2 (Activity B input port 2) as the destination port in the link. In this case, the package definitions defined for ActA_OP1 must be compatible with the package definitions defined for ActB_IP2.

If you define multiple packages for a pair of ports, all the packages in the output port must be compatible with all the packages in the input port. The validation procedure compares each pair of definitions in the linked ports for compatibility. For example, suppose the package definitions for OP1 are ADef1 and ADef2 and the package definitions for IP2 are BDef1 and BDef2. The validation checks the following pairs for compatibility:

ADef1 and BDef1
ADef1 and BDef2
ADef2 and BDef1
ADef2 and BDef2

If any pair fails the compatibility test, the validation fails.

Because package compatibility is checked across links, compatibility is not validated until you attempt to validate the process definition. To avoid errors at that point, be sure to plan carefully when you design the workflow.

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, then 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.

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
- The transition type

(To learn more about how activity transitions work, refer to [When the activity is complete](#), page 249.)

Number of completed tasks as transition trigger

Starting an activity may generate multiple tasks. For some activities you may want all the generated tasks to be completed before moving to the next activity. In other cases, you may only need some of the tasks to be completed before moving to the next activity. For example, suppose the performer for a review activity is a group that has five users. When the activity starts, five tasks are generated, one for each group member. However, the author only needs three reviews. Therefore, when you define the activity, you can specify that only three of the five tasks must be completed to trigger the transition.

By default, all generated tasks must be completed. Both WFM and BPM allow you to change that default unless the activity's performer category is category 9 (Some Users in a Group or Repository, Sequentially).

If the number of completed tasks you specify is greater than the total number of work items for an activity, Content Server requires all work items for that activity to complete before triggering the transition.

The number of completed tasks required to trigger a transition is recorded in the `transition_eval_cnt` attribute of the activity's definition.

Transition types

An activity's transition type defines how the output ports are selected when the activity is complete. There are three types of transitions:

- Prescribed

If an activity's transition type is prescribed, the server delivers packages to all the output ports. This is the default transition type.

If the activity's user category for the performer is 9, Some Users in a Group or in a Repository Sequentially, and the activity contains a revert link so that a performer

can reject the activity back to a previous performer in the sequence, the activity cannot use a prescribed transition. It must use a manual or automatic transition.

- Manual

If the activity's transition type is manual, the activity performers must indicate at runtime which output ports receive packages.

If you choose a manual transition type, you can also decide

- How many activities the performer can choose
- What occurs if there are multiple performers and some performers select a previous activity (a revert port) and some select a next activity (a forward port)

[Limiting output choices in manual transitions, page 220](#) and [Setting preferences for output port use in manual transitions, page 220](#) describes these secondary choices.

- Automatic

If the activity's transition type is automatic, you must define one or more route cases for the transition. Each route case is an expression that evaluates to TRUE or FALSE. The server evaluates the route cases and selects the ports to receive packages based on which route case is TRUE.

For information about route cases, refer to [Route cases for automatic transitions, page 221](#).

Limiting output choices in manual transitions

If an activity's transition type is "manual", the activity's performer chooses the next activity or activities. Both WFM and BPM allow you to limit the number of ports that a performer can select. The number you specify is recorded in the activity's `transition_max_output_cnt` attribute.

When the performer completes the activity and selects the ports, Content Server verifies that the number of output ports identified by the user does not exceed the number defined in the `transition_max_output_cnt` attribute. If the number of selected ports exceeds the value in the attribute, the operation fails with an error.

Setting preferences for output port use in manual transitions

When activities with a manual transition type have more than one performer, the choices for the next activity may be contradictory. Some performers may choose a previous activity (a revert port) and others may choose a next activity (a forward port). For such activities, you must define how the server should respond to that situation.

The following options are supported:

- Trigger all selected ports
- Give priority to revert ports

If both revert and forward ports are selected, only the revert ports are triggered. If there are no revert ports selected, the forward ports are triggered.

- Give priority to forward ports

If both revert and forward ports are selected, only the forward ports are triggered. If there are no forward ports selected, the revert ports are triggered.

- Trigger a selected revert port immediately

If any performer selects a revert port, the port is triggered immediately. The current activity is finished even though there may be uncompleted work items.

- Trigger a selected forward port immediately

If any performer selects a forward port, the port is triggered immediately. The current activity is finished even though there may be uncompleted work items.

The option you choose is recorded in the `transition_flag` attribute of the activity definition.

Route cases for automatic transitions

A route case represents one routing condition and one or more associated ports. A routing condition is one or more Boolean expressions that are ANDed together. When an activity whose transition type is automatic is completed, the server tests each of the activity's route cases. It activates the port or ports associated with the first route case that returns TRUE.

The server uses the following logic to test route cases:

```
If (route case condition #0) then
  Select port, ...
Else if (route case condition #1) then
  Select port, ...
Else
  Select port, ...
```

Route case conditions must be Boolean expressions. They are typically used to check attributes of the package's components, the containing workflow, or the last completed work item. If the route case condition includes a reference to a repeating attribute, the attribute must have at least one value or the condition generates an error when evaluated.

The expression can reference attributes from the workflow object, the work item object, or any package, visible or invisible, associated with the activity. If you are using BPM to define the route cases, the expression can be an XPath expression.

You can also define an exceptional route case, which is a route case that has no routing condition and applies only when all other route cases fail. An activity can only have one exceptional route case.

If an XPath expression is included and the referenced XML file has an associated schema, the XPath is validated against that schema when the route case is added to the activity definition. (Schemas are associated with XML files in packages at the time the package is defined.)

How the route cases are stored internally and which methods are used to add the route cases to the activity definition depends on whether any of an activity's route cases include an XPath expression.

Implementation without an XPath expression

If the route cases you define for an activity's transition do not include an XPath expression, Content Server adds the route cases using an `addRouteCase` method. Internally, this creates a `dm_cond_expr` object and one or more `dm_func_expr` objects. The `r_condition_id` attribute in the activity definition is set to the object ID of the `dm_cond_expr` object.

Implementation with an XPath expression

If any of the route cases defined for an activity's transition include an XPath expression, Content Server adds the route cases using the `addConditionRouteCase` method. Internally, this creates one or more `dmc_composite_predicate` objects and, for each composite predicate object, one or more `dmc_transition_condition` objects. The `r_predicate_id` attribute, a repeating attribute, is set to the object IDs of the composite predicate objects.

Each composite predicate object represents one route case defined for the activity's transition. For example, suppose you define the following as the activity's route cases:

```
If dm_workflow.r_due_date>"TODAY" select portA
  else if dm_workflow.r_due_date<"TODAY" select portB
  else if dm_workflow.r_due_date="TODAY" select portC
```

Each if condition represents one route case. Content Server would create three composite predicate objects, one for each.

Each composite predicate object points to one or more transition conditions. Each transition condition object represents one comparison expression in the route case. For example, if the route case has two comparison expressions, then the composite predicate object will point to two transition conditions. To illustrate, suppose the route case condition is:

```
dmi_workitem.r_due_date>"TODAY" and dm_workflow.supervisor_name="JohnDoe"
```

This creates two transition condition objects. One records the first comparison expression:

```
dmi_workitem.r_due_date>"TODAY"
```

The other records the second comparison operation:

```
dm_workflow.supervisor_name="JohnDoe"
```

The `predicate_id` attribute of the composite predicate object points to the object IDs of these two transition conditions. The results of the comparison expressions are always ANDed together to obtain a final Boolean result for the full route case condition.

Compatibility of the implementations

An activity definition may not both the `r_condition_id` and the `r_predicate_id` attributes set at the same time. If you wish to add an XPath expression to an existing set of route case conditions that does not currently contain an XPath expression, you must first execute a `removeRouteCase` method to remove all route cases and then redefine the full set of route cases, to add the new XPath addition.

You cannot modify an existing set of route cases that includes an XPath expression using WFM unless you must first remove all the existing route cases. After you remove the existing route cases, you can then redefine the route cases in WFM (although you will not be allowed to add any that include an XPath expression).

How the associated ports are recorded

When route cases are saved to the activity definition, Content Server also sets the activity's `r_condition_name` and `r_condition_port` attributes. These are repeating attributes. The `r_condition_name` attribute is set to names assigned by Content Server, one value for each route case. In `r_condition_port`, Content Server records the ports to be selected when the route case in the corresponding index position in `r_condition_name` evaluates to TRUE.

The ports associated with each route case are recorded in the `r_condition_port` attribute in the activity definition.

Warning and suspend timers

Content 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.

- 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.

Warning timers

There are two types of warning timers:

- 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.

Who receives the messages varies depends on the configured action for the timer. If there is no action defined, the message is sent to the task performer in addition to the workflow supervisor.

Note: Actions for timers may only be configured using BPM. Workflow Manager does not provide facilities for configuring an action.

If you are defining the activity in Workflow Manager, the timer is configured to deliver the warning one time. If you are defining the activity in Business Process Manager, you can configure the timer to deliver the message once or repeatedly. For example, in BPM, you can instruct Content Server to send a message to the workflow supervisor if an activity is not started within 60 minutes of a workflow's start and to continue sending the message every 10 minutes thereafter, until the activity is started.

Timers are created at runtime if they are defined for an activity. Pre-timers are instantiated when a workflow is started. Post-timers are instantiated when the activity starts. They are stored in the repository as `dmi_wf_timer` objects.

The attributes of `dmi_wf_timer` objects identify the workflow, the activity, and the type of the timer. They also reference a module config object that identifies the action or actions to perform when the timer is triggered. Actions are defined in business object modules that reside in the Java method server. Each business object module is represented in the repository as a module config object.

Warning timers are executed by the dm_WfmsTimer job. This job is installed in the inactive state. If you plan to use warning timers, make sure that the job is activated before a workflow with an activity with a warning timer is started. (Refer to the *Content Server Administrator's Guide* for instructions about activating a job.) For information about the execution of a warning timer, refer to [How activity timers work](#), page 251.

Suspend timers

Note: Suspend timers are only supported if you are using BPM to manage the workflow or if the workflow application is built using a 5.3 DFC or client library.

Suspend timers are created when an activity is halted if the user halting the activity defines a time period for the halt. For example, a workflow supervisor may wish to halt an activity for 60 minutes (1 hour). When the supervisor halts the activity, he or she also specifies that the halt should last only 60 minutes. Content Server responds by creating a wf timer object. The attributes of the wf timer object identify the timer as a suspend timer and the date and time at which the activity should be resumed. In this example, that date and time is 60 minutes after activity is halted. At the expiration of 60 minutes, the timer is triggered and the activity is automatically resumed.

Programmatically, you can specify a suspension interval in the haltEx method defined for the IDfWorkflow interface. You can also specify a suspension interval when you halt an activity through Webtop.

Suspend timers are executed by the dm_WFSuspendTimer job. This job is installed in the inactive state. If you plan to use a suspend timer, make sure the job is activated. For activation instructions, refer to the *Content Server Administrator's Guide*. For information about how a suspend timer is executed, refer to [How activity timers work](#), page 251.

Package control

Package control is a specific constraint on Content 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 attribute of the generated package or wf attachment object. If package control is enabled, Content Server sets the r_component_name attribute to a single blank even if the component names are specified in the methods.

Package control is enabled at either the repository level or within the individual workflows. 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` attribute to 1. (To enable or disable package control at the repository level, refer to [Configuring repository-level package name control, page 80](#) in the *Content Server Administrator's Guide*.)

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. ([Validation and installation, page 226](#), describes how to validate and install process and activity definitions.)

Validation and installation

Activity and process definitions must be validated and installed before users can start a workflow based on the definitions.

[Validating process and activity definitions, page 226](#), describes validation. Installing definitions is described in [Installing process and activity definitions, page 228](#).

Validating process and activity definitions

Validating activity and process definitions ensures that the workflow will function correctly when used.

You can validate activity definitions individually, before you validate the process definition, or concurrently with the process definition. You cannot validate a process definition that contains unvalidated activities unless you validate the activities

concurrently. If you validate only the process, the activities must be in either the validated or installed state.

To validate an activity or process definition requires either:

- Relate permission on the process or activity definition
- Sysadmin or Superuser privileges

Content Server uses the Validate method to validate process and activity definitions.

What validation checks

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` attributes 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

Validating port connectability

The output port and input port referenced by a link must be connectable. When you validate a process definition, the server checks the connectability of each port pair referenced by a link in the process.

To check connectability, validation verifies that:

- Both ports handle the same number of packages

If the numbers are the same, the method proceeds. Otherwise, it reports the incompatibility.

- 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 the rules of package compatibility, refer to [Package compatibility, page 218.](#))

Installing 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, refer to [Reinstalling after making changes, page 258](#), for instructions. Do not use the information in this section.

Installing a process or activity definition makes the definition available for use by users. The 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

Content Server uses an Install method to install activity and process definitions.

Architecture of workflow execution

Workflow execution is implemented with the following object types:

- dm_workflow
- dmi_workitem
- dmi_package
- dmi_queue_item
- dmi_wf_timer

Workflow objects

Workflow objects represent an instance of a workflow definition. 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 attribute).

This person functions much like the owner of an object, with the ability to change the workflow's attributes and change its state. (For more information about the workflow supervisor, refer to [The workflow supervisor, page 233](#).)

Activity instances

A workflow object contains attributes that describe the activities in the workflow. These attributes are set automatically, based on the workflow definition, when the workflow object is created. They are repeating attributes, and the values at the same index position across the attributes represent one *activity instance*.

The attributes 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 attributes change to reflect the status of the activities at any given time in the execution. (The description of the `dm_workflow` object type in [Workflow, page 526](#), in the *EMC Documentum Object Reference Manual* provides a full list of the attributes that make up an activity instance.)

Work 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's performer (either a person or an invoked method).

Work items are instances of the `dmi_workitem` object type. A work item object contains attributes that identify the activity that generated the work item and the user or method who will perform the work, record the state of the work item, and record information for its management.

The majority of the attributes are set automatically, when the server creates the work item. A few are set at runtime. For example, if the activity's performer executes a `Repeat` method to give the activity to a second round of performers, the work item's `r_ext_performer` attribute is set.

For a complete description of the `dmi_workitem` object type, refer to [Work Item, page 531](#), of the *EMC Documentum Object Reference Manual*.

Work items and queue items

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. A work item object's `r_queue_item_id` attribute identifies its peer queue item, and the `item_id` attribute in the queue item object identifies its underlying, associated work item. Work items for automatic activities do not have peer queue item objects.

For a description of the queue item object type attributes and how work items use them, refer to [Queue Item, page 352](#), in the *EMC Documentum Object Reference Manual*.

How manual activity work items are handled

The first operation that must occur on a work item is acquisition. A work item must be acquired by a user before the user can perform the work represented by the work item. The user acquiring the work item is typically the work item's designated performer. However, the user may also be the workflow's supervisor or a user with Sysadmin or Superuser privileges. However, if workflow security is disabled, any user can acquire any work item. (For information about workflow security, refer to [The `enable_workitem_mgmt` key, page 234](#).)

Users typically acquire a work item by selecting and opening the associated Inbox task. Internally, Content Server executes an `Acquire` method when a user acquires a work item. Acquiring a work item sets the work item's state to `acquired`.

After a user acquires a work item, he or she becomes the work item's performer. 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. For example, delegating a work item to another user is implemented using a `Delegate` method. Adding a note to a package is implemented using an `Addnote` method. Removing a note is implemented using a `Removenote` method. And sending the work item to additional performers is implemented using a `Repeat` method.

Resetting priority values

Note: The information in this section does not apply to priorities set by work queue policies. It applies only to the activity priorities defined in the workflow definition (the `dm_process` object).

Each work item inherits the priority value defined in the process definition for the activity that generated the work item. Content Server uses the inherited priority value of automatic activities, if set, to prioritize execution of the automatic activities. (Content

Server ignores priority values assigned to manual activities.) A work item's priority value can be changed at runtime. In an application, you use a `Setpriority` method to accomplish this. To change a work item's priority, the work item cannot be in the finished state and the workflow must be in the running state.

By default, only the workflow supervisor or a user with Sysadmin or Superuser privileges can reset a work item's priority. However, if `enable_workitem_mgmt`, a `server.ini` key, is set to T, any user can change a work item's priority. (For information about workflow security, refer to [The `enable_workitem_mgmt` key](#), page 234.)

Changing a work item's priority generates a `dm_changepriorityworkitem` event. You can audit that event. Changing a priority value also changes the priority value recorded in any queue item object associated with the work item.

Completing work items

When a work item is finished, the performer indicates the completion through a client interface. Only a work item's performer, the workflow supervisor, or a user with Sysadmin or Superuser privileges can complete a work item. The work item must be in the acquired state. Internally, Content Server executes a `Complete` method when a work item is designated as completed.

Executing a `Complete` method updates the workflow's `r_last_performer` and `r_complete_witem` attributes. Updating the `r_complete_witem` attribute triggers evaluation of the activity instance's completion status. If the server decides that the activity is finished, it selects the output ports based on the transition condition, manufactures packages, delivers the packages to the next activity instances, and marks this activity instance as finished by setting the `r_act_state` attribute to `finished`.

For automatic activities, the method also records the return value, OS error, and result ID in the work item's `return_value`, `r_exec_os_error`, and `r_exec_result_id` attributes. If the `return_value` is not 0 and the `err_handling` is 0, the `Complete` method changes the activity instance's state to `failed` and pauses the associated work item. The server sends email to the workflow supervisor and creates a queue item for the failed activity instance.

Signing off manual work items

Frequently, a business process requires the performers to sign off the work they do. Content 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 options, refer to [Signature requirement support](#), page 95.

Package objects

Packages contain the objects on which the work is performed. Packages are implemented as `dmi_package` objects. A package object's attributes:

- 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's 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. [How activities accept packages, page 249](#), describes how packages are handled at runtime.

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's performer. A note may stay with a package as it moves through the workflow or it may be available only in the work items associated with one activity.

If an activity accepts multiple packages, Content 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.

Attachments

Attachments are objects that users attach to a running workflow or an uncompleted work item. Typically, the objects are objects that support the work required by the workflow's 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 may be removed when they are no longer needed. After an attachment is added, it is available to the performers of all subsequent activities.

Attachments may be added by the workflow's creator or supervisor, a work item's 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.

Programmatically, attachments are created using `IDfWorkflow.addAttachment` and removed from a workflow using `IDfWorkflow.removeAttachment`. Attachments are accessed programmatically using the `IDfWorkflowAttachment` interface.

The 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 may act as the workflow supervisor. In addition, superusers are treated like the creator of a workflow and can change object attributes, if necessary. However, messages that warn about execution problems are sent only to the workflow supervisor, not to superusers.

A workflow's supervisor is recorded in the `supervisor_name` attribute of the workflow object.

The workflow agent

The workflow agent is the Content Server facility that controls the execution of automatic activities. The workflow agent is installed and started with Content Server. It maintains a master session and, by default, three worker sessions.

When Content Server creates an automatic activity, the server notifies the workflow agent. The master session is quiescent until it receives a notification from Content 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 instructions on tracing or disabling the agent, as well as instructions on changing the number of worker sessions and the sleep interval, refer to [Configuring the workflow agent, page 124](#), in the *Content Server Administrator's Guide*.

The enable_workitem_mgmt key

The enable_workitem_mgmt key is a key in the server.ini file that controls who can perform certain workflow operations. The affected operations are:

- Acquiring a work item
- Delegating a work item
- Halting and resuming a running activity
- Setting a work item's priority value at runtime

If the key is not set or is set to F (FALSE), Content Server allows only the designated performer or a privileged user to perform those operations for a particular workitem. If the key is set to T (TRUE), any user can perform the operations. The default for the key is F (FALSE).

Instance states

This section describes the valid states for workflows, activity instances, and work items.

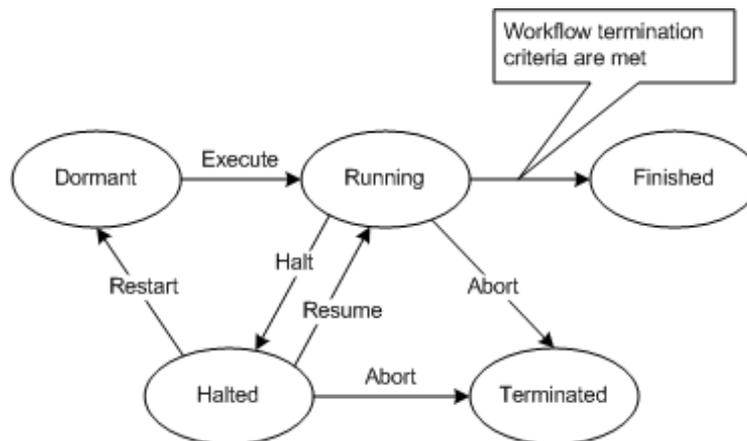
Workflow states

Every workflow instance exists in one of five possible states: dormant, running, finished, halted, or terminated. A workflow's current state is recorded in the `r_runtime_state` attribute 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.

Figure 9-2, page 235, illustrates the states.

Figure 9-2. Workflow state diagram



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's 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 `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 with the `Halt` method, which changes the workflow's state to halted. From a halted state, the workflow's supervisor can restart,

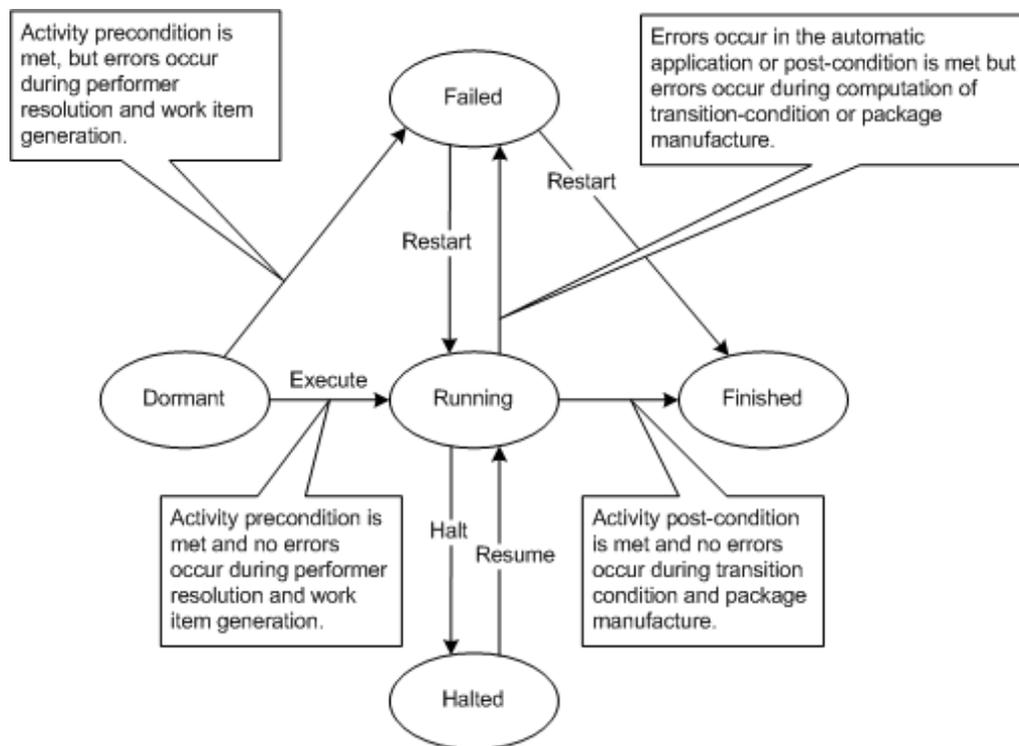
resume, or abort the workflow. (For information about all three options for a halted workflow, refer to [Changing workflow, activity instance, and work item states](#), page 255.)

Activity instance states

Every activity instance exists in one of five states: dormant, active, finished, failed, or halted. An activity instance's state is recorded in the `r_act_state` attribute of the `dm_workflow` object, as part of the activity instance.

Figure 9-3, page 236, illustrates the activity instance states and the operations or conditions that move the instance from one state to another.

Figure 9-3. Activity instance state diagram



During a typical workflow execution, an activity's state is changed by the server to reflect the activity's 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's starting condition is fulfilled and server begins to resolve the activity's performers and generate work items.

If the server encounters any errors, it changes the activity instance's 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 may 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.

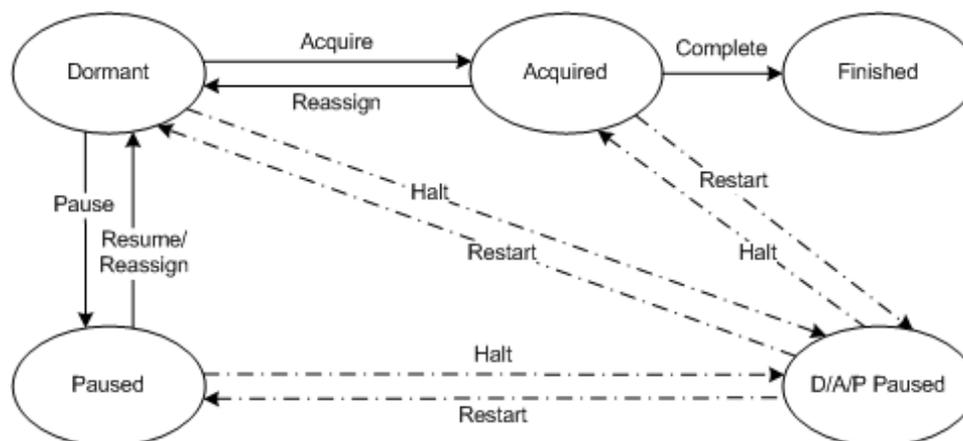
A running activity can be halted. Halting an activity sets its state to halted. By default, only the workflow supervisor or a user with Sysadmin or Superuser privileges can halt or resume an activity instance. However, if `enable_workitem_mgmt`, a `server.ini` key, is set to T (TRUE), any user can halt or resume a running activity.

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. (For information about how suspension intervals are implemented, refer to [How activity timers work](#), page 251.)

Work item states

A work item exists in one of the following states: dormant, paused, acquired, or finished. [Figure 9-4, page 237](#), shows the work item states and the operations that move the work item from one state to another.

Figure 9-4. Work item states



A work item's state is recorded in the `r_runtime_state` attribute of the `dmi_workitem` object.

When the server generates a work item for a manual activity, it sets the work item's state to dormant and places the peer queue item in the performer's inbox. The work item remains in the dormant state until the activity's performer acquires it. Typically, acquisition happens when the performer opens the associated inbox item. At that time, the work item's state is changed to acquired.

When the server generates a work item for an automatic activity, it sets the work item's state to dormant and places the activity on the queue for execution. The application must issue the `Acquire` method to change the work item's state to acquired.

After the activity's work is finished, the performer or the application must execute the `Complete` method to mark the work item as complete. (Refer to [Completing work items, page 231](#), for information about completing a task.) This changes the work item's state to finished.

A work item can be moved manually to the paused state by the activity's performer, the workflow's 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.

Starting a workflow

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's template. The `execute` method must be issued by the workflow's creator or supervisor or a user with `Sysadmin` or `Superuser` privileges. If the user is starting the workflow through a Documentum client interface, such as Documentum Desktop, the user must be defined as a `Contributor` also.

How execution proceeds

This section describes how a typical workflow executes. The description includes the following topics:

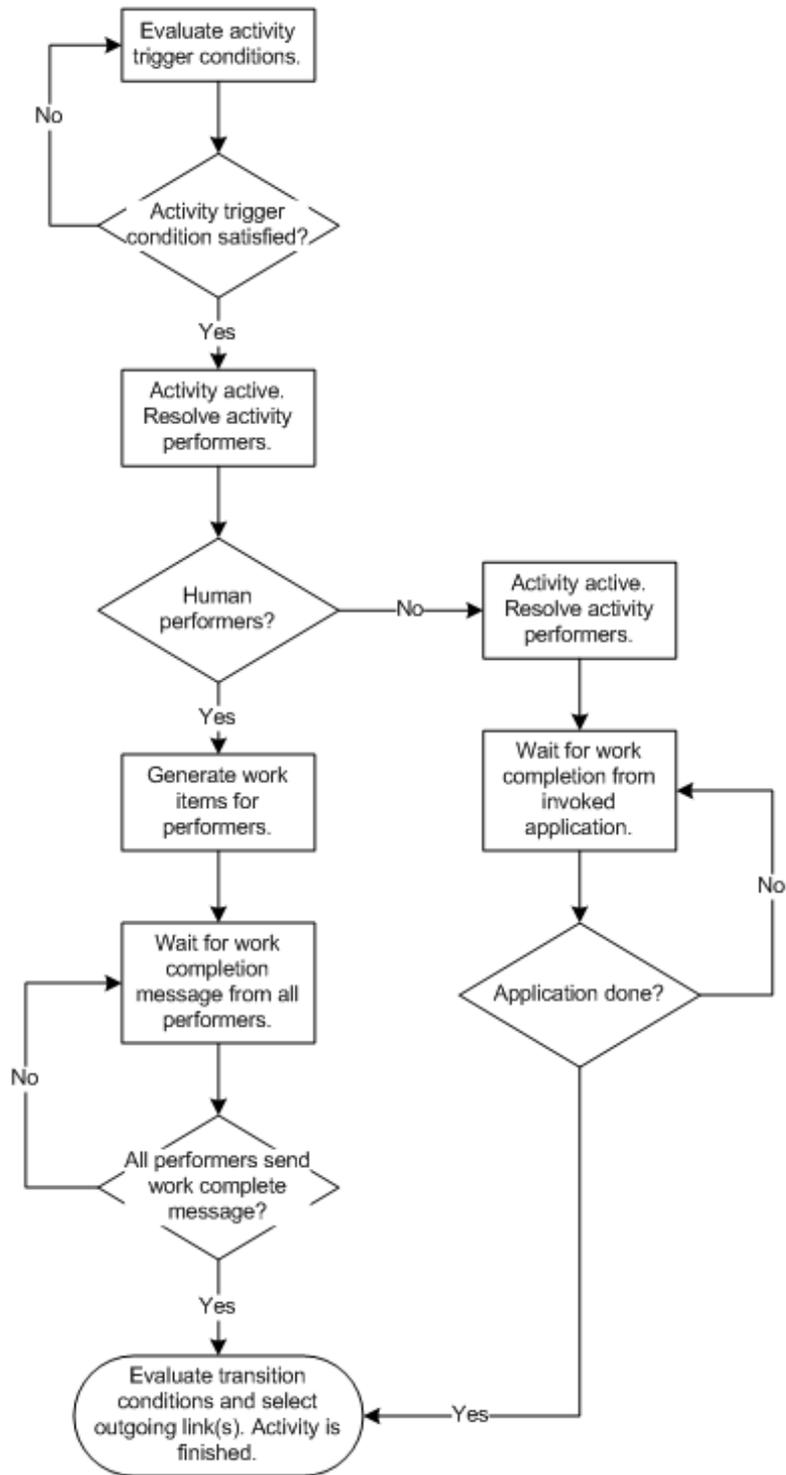
- [The workflow starts, page 241](#)
- [Activity execution starts, page 241](#)
- [Executing automatic activities, page 244](#)
- [Evaluating the activity's completion, page 246](#)

- [When the activity is complete, page 249](#)

In addition, this section contains a description of how packages are handled by activities ([How activities accept packages, page 249](#)) and how timers are implemented ([How activity timers work, page 251](#)).

[Figure 9–5, page 240](#), illustrates the general execution flow described in detail in the text of this section.

Figure 9-5. Execution flow in a workflow



The workflow starts

A workflow starts when a user issues the execute method against a `dm_workflow` object. The execute method does the following:

- Sets the `r_pre_timer` attribute 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` attribute 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.

Activity execution starts

For Step and End activities, execution begins when a package arrives at one of the activity's input ports. If the package is accepted, it triggers the server to evaluate the activity's starting condition. ([How activities accept packages, page 249](#), describes how packages are evaluated in detail.)

For Begin activities, execution begins when the 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's starting condition just as it does for Step and End activities.

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's performers.

After the server determines that an activity's starting condition is satisfied, it consolidates packages if necessary. Package consolidation is described in detail in [Package consolidation, page 242](#).

Next, the server determines who will perform the work and generates the required work items. (Refer to [Resolving performers and generating work items, page 243](#), for details.) If

the activity is an automatic activity, the server queues the activity for starting. ([Executing automatic activities, page 244](#), describes how automatic activities are executed.)

Evaluating the starting condition

An activity's 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` attributes in the activity definition. When a workflow is created, these values are copied to the `r_trigger_threshold` and `r_trigger_event` attributes in the workflow object.

When an activity's input port accepts a package, the server increments the activity instance's `r_trigger_input` attribute 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's inbox. If it is an automatic activity, the server passes the performer's name to the application invoked for the activity.

Package consolidation

If an activity's 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 package objects' i_acceptance_date attribute.

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.

Manual activities

The server uses the value in the performer_type attribute in conjunction with the performer_name attribute if needed to determine the activity's performer. (Table 9-1, page 206, lists the valid values for performer_type and the performer selection they represent.) After the performer or performers 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 attribute of the performer's user object.

If automatic delegation fails, the server reassigns the work item based on the setting of the control_flag attribute in the definition of the activity that generated the work item. (For details, refer to [Delegation, page 204](#).)

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.

Automatic activities

The server uses the value in the performer_type attribute in conjunction with the performer_name attribute if needed to determine the activity's performer. (Table 9-1, page 206, lists the valid values for performer_type and the performer selection they represent.) The server passes the name of the selected performer to the invoked program.

(Refer to [Executing automatic activities, page 244](#), below, for details of how automatic activities are executed.)

The server generates work items but not peer queue items for work items representing automatic activities.

Resolving aliases

When the `performer_name` attribute contains an alias, the server resolves the alias using a resolution algorithm determined by the value found in the activity's `resolve_type` attribute. [Resolving aliases in workflows, page 315](#), describes the resolution algorithms.

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.

Executing automatic activities

The master session of the workflow agent controls the execution of automatic activities. The workflow agent is an internal server facility. For a description of the workflow agent, refer to [The workflow agent, page 234](#).

Assigning an activity for execution

After the server determines the activity performer and creates the work item, the server notifies the workflow agent's 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` attribute of the work items to the name of the server config representing the Content Server. The maximum number of work items in a batch is the lessor 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 five times the number of worker threads (5 x no. of worker threads).

3. After all claimed work items are dispatched, the master agent returns to sleep until another notification arrives from Content Server or the sleep interval passes.

Note: If the Content 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. [Recovering automatic activity work items on Content Server failure, page 125](#), in the *Content Server Administrator's Guide* provides instructions for that procedure.

Executing an activity's 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. After verifying the ready state of the activity, it executes the method associated with the activity. The method is always executed as the server regardless of the `run_as_server` attribute setting in the method object. If the activity fails for any reason, the selected performer receives a notification.

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 can occur in repositories with multiple servers, each having its own workflow agent.)

The server passes the following information to the invoked program:

- Repository name
- User name (this is the selected performer)
- Login ticket (generated with Getlogin method)
- Work item object ID
- Mode value

The information is passed in the following format:

```
-dochbase_name repository_name -user user_name -ticket login_ticket
-packageId workitem_id mode mode_value
```

The mode value is set automatically by the server. [Table 9-2, page 246](#), lists the values for the mode parameter:

Table 9-2. Mode parameter values

Value	Meaning
0	Normal
1	Restart (previous execution failed)
2	Termination situation (re-execute because workflow terminated before automatic activity user program completed)

The 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.

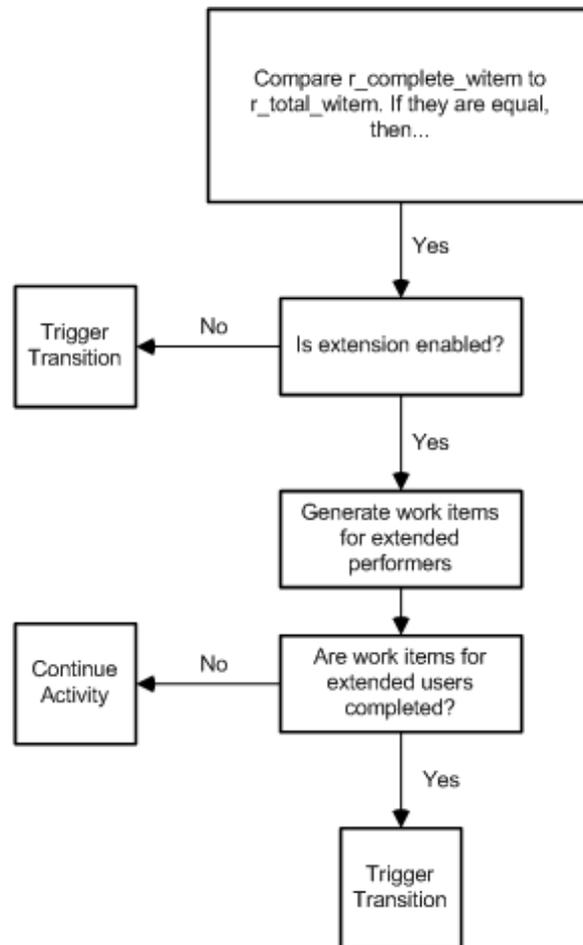
Evaluating the activity's completion

When a performer completes a work item, the server increments the `r_complete_witem` attribute 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` attribute to the value in the workflow's `r_total_workitem` attribute. The `r_total_witem` attribute records the total number of work items generated for the activity. The `r_complete_witem` attribute records how many of the activity's work items are completed.

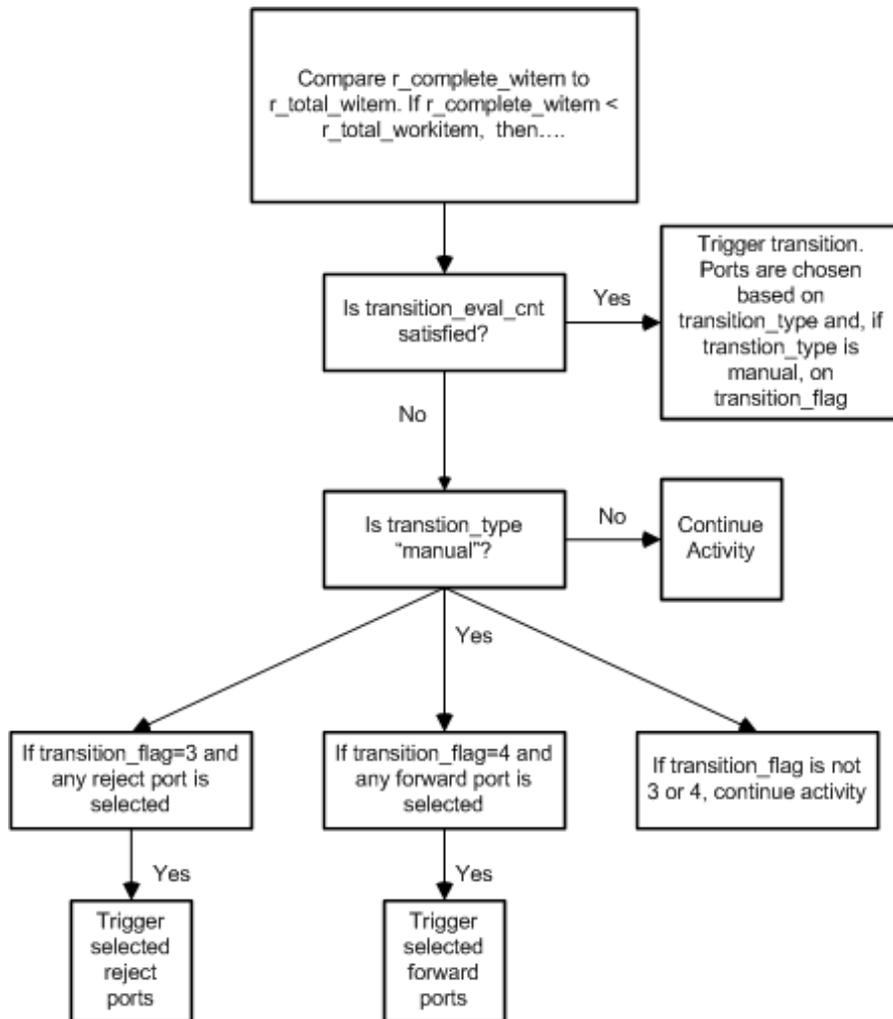
If the two values are the same and extension is not enabled for the activity, the server considers that the activity is completed. If extension is enabled, the server:

- Collects the second-round performers from the `r_ext_performer` attribute 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 `i_performer_flag` to indicate that the activity is in the extended mode and no more extension is allowed

[Figure 9-6, page 247](#), illustrates the decision process when the attributes are equal.

Figure 9-6. Behavior if r_complete_witem equals r_total_workitem

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` attribute to determine whether to trigger a transition. The `transition_eval_cnt` attribute specifies how many work items must be completed to finish the activity. The `transition_flag` attribute defines how ports are chosen for the transition. [Figure 9-7, page 248](#), illustrates the decision process when `r_complete_witem` and `r_total_workitem` are not equal.

Figure 9-7. Behavior if `r_complete_witem < r_total_workitem`

If an activity's transition is triggered before all the activity's work items are completed, Content Server marks the unfinished work items as pseudo-complete and removes them from the performers' inboxes. 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's transition is triggered before all work items are completed, any extended work items are not generated even if extension is enabled.

When the activity is complete

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 Content Server using a Setoutput method. 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. (For information about limiting the number of choices, refer to [Limiting output choices in manual transitions, page 220](#). For information about defining processing for the choices, refer to [Setting preferences for output port use in manual transitions, page 220](#).)

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` attribute is set, the server evaluates the route cases. If the activity's `r_predicate_id` attribute 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. (For information about how route cases are defined, refer to [Route cases for automatic transitions, page 221](#).) The server selects the ports associated with the first route case that returns a TRUE value.

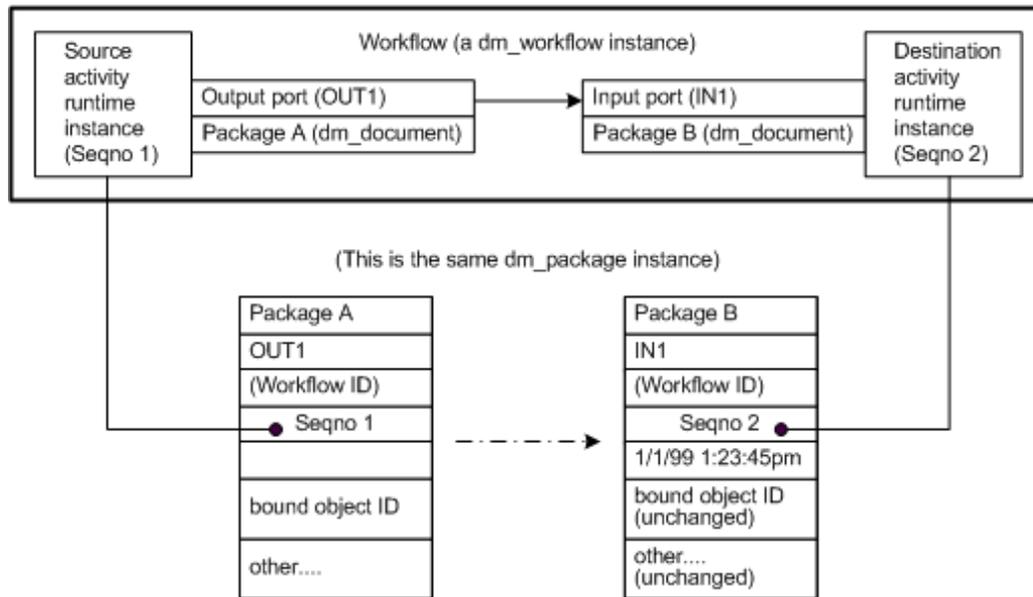
After the ports are determined, the server creates the needed package objects. If the package creation is successful, the server considers that the activity is finished. At this point, the cycle begins again with the start of the next activity's execution.

How activities accept packages

When packages arrive at an input port, the server checks the port definition to see if the packages satisfy the port's 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` attributes of those packages. (For details about packages, refer to [Package definitions, page 216](#).)

[Figure 9–8, page 250](#), illustrates this process.

Figure 9-8. Changes to a package 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:

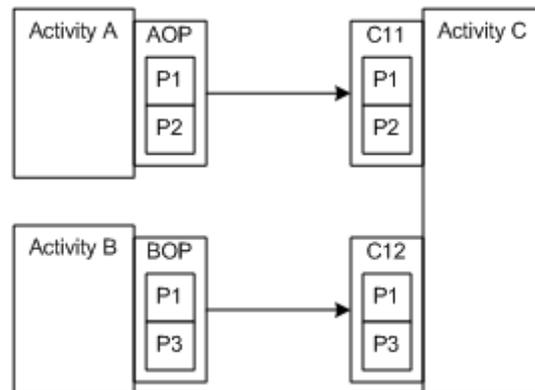
- Incrementing r_trigger_revert if the triggered port is a revert port
 - As soon as a revert port is triggered, the activity becomes active and no longer accepts any incoming packages (from input or other revert ports).
- Incrementing r_trigger_input if the triggered port is an input port
 - As soon 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 [Figure 9-9, page 251](#), 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 9-9. Package arrival



How activity timers work

There are three types of timers for an activity. An activity may 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

This section describes how these timers are implemented and behave in a running workflow. For a general description of each timer type, refer to [Warning and suspend timers, page 223](#).

Pre-timer instantiation

When a workflow instance is created from a workflow definition, Content 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` executes in the Java method server.

Post-timer instantiation

A post-timer is instantiated when the activity for which it is defined is started. When the activity is started, Content 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.

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, Content Server creates a `dmi_wf_timer` object that identifies the workflow, the activity, and the date and time at which to resume the activity.

Note: The argument that defines a suspend timer is supported only in 5.3 client libraries and BPM 5.3. The WFM client does not support suspend timers.

Activating pre- and post-timers

The task of checking the pre-timers and post-timers is performed by the `dm_WfmsTimer` job. The `dm_WfmsTimer` job executes the `dm_WfmsTimer` method, a Java method. The job finds all timer objects representing pre-timers and post-timers that have expired. (A timer is expired if the value in its `r_timer` attribute is less than the current date and

time.) For each expired timer, the job's method invokes the `dm_bpm_timer` method, passing it information about the timer. The information includes the timer's type and a module config object ID. The module config object points to a business object module in the Java method server. The `dm_bpm_timer` method executes the action defined in the business object module.

If the timer is defined to execute multiple times when needed, the `dmi_wf_timer` is updated to indicate the next trigger time and action.

The `dm_WfmsTimer` job and related methods are installed by a script when a repository is configured. The job is installed in the inactive state. If you use pre-timers and post-timers, make sure that your system administrator activates this job. When it is active, it runs every hour by default.

Activating suspend timers

The task of checking suspend timers is performed by the `dm_WfSuspendTimer` job. The job executes the `dm_WfSuspendTimer` method. The method checks the repository for all expired suspend timers. An expired suspend timer is a suspend timer whose `r_timer` value is less than the current date and time. For each expired suspend timer found, the method resumes the corresponding activity instance.

The `dm_WfSuspendTimer` method is a Java method executed by the Java method server. Both the job and method are installed by a script when a repository is configured. When the job is active, it runs every hour by default.

Compatibility with pre-5.3 timers

The implementation of pre-timers and post-timers in Content Server 5.3 is completely compatible with the warning timer implementation in prior workflow implementations. For details, refer to the Migration guide.

User time and cost reporting

Webtop allows an activity's performer to enter the time spent and cost incurred to complete an activity's task. The time and cost values the user enters are stored in the workitem object, in the `user_time` and `user_cost` attributes. If the `dm_completedworkitem` event is audited, the values are also recorded in the `string_3` attribute of the generated audit trail entry.

The time and cost values are not used by Content Server. They are recorded for the use of user applications.

Reporting on completed workflows

You can view reports about completed workflows using the Webtop Workflow Reporting tool. The data includes such information 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.

For each execution of the job, the method uses the value in the job's `method_data` attribute to determine which audit trail entries to collect. On the first execution, the attribute is blank and the method collects all entries. At the completion of each execution, `method_data[0]` is set to a value in the following format:

```
cutoff_date_utc=yyyy/mm/dd hh/mm/ss
```

where `yyyy/mm/dd hh/mm/ss` is the current UTC time.

On second and subsequent executions, the method collects only the audit trail entries generated after the date and time specified in `method_data[0]`.

The Webtop Workflow Reporting tool uses the information in the objects generated by the job to create its reports. The `dm_WFReporting` job is installed with Content Server. It is installed in the inactive state and must be activated if you want to obtain these reports. When active, the job runs once a day by default. Additionally, you must turn on auditing for all workflow events to generate the audit trail entries that are the source of the tool's data. For information about activating a job, refer to [Activating or inactivating a job, page 151](#), in the *Content Server Administrator's Guide*. For information about starting auditing, refer to [Auditing system events, page 409](#), also in the *Content Server Administrator's Guide*. For information about accessing and using the Webtop Workflow Reporting tool, refer to the Webtop documentation.

Changing workflow, activity instance, and work item states

This section describes the state changes that a workflow supervisor, a user with Sysadmin or Superuser privileges, or an application can manually apply to a workflow, an activity instance, or a work item.

Halting a workflow

Only the workflow supervisor or a user with Superuser or Sysadmin privileges can halt a workflow. You cannot halt a workflow if any work items generated by automatic activities are in the acquired state.

When a workflow is halted, the server changes the state of all dormant or acquired work items to D/A/P paused and changes the state of the workflow to halted. The running activities and current work items cannot change states and new activities cannot start.

If you are using the API to halt the workflow, use a Halt method.

You can resume, restart, or abort a halted workflow. Refer to [Resuming a halted workflow or activity](#), page 256, [Restarting a halted workflow or failed activity](#), page 256, and [Aborting a workflow](#), page 256 for information.

Activity instances in halted workflows

Halting a workflow freezes its activity instances, which freezes all generated work items. An activity instance can only change state if the containing workflow is running. Any attempted action that causes a state change is prohibited.

For example, if a workflow is halted after a user acquires a work item and the user completes the task and tries to mark the work item as finished, the server will not accept the change. Marking the item as finished would cause the activity instance's state to change to finished. Although the activity instance is active, the server denies the attempt because the containing workflow is halted.

Halting an activity instance

Halting an activity instance changes the state of the activity's dormant and acquired work items to D/A/P paused and changes the state of the activity instance to halted.

To halt an activity instance, use a Halt method. By default, only the workflow supervisor or a user with Superuser or Sysadmin privileges can halt an activity instance. If `enable_workitem_mgmt`, a `server.ini` key, is set to T (TRUE), any user can halt an activity instance. The activity instance must be in the active state.

To use a Halt method, you must know the instance's sequence number in the workflow. The sequence number is recorded in the `r_act_seqno` attribute of the workflow. This is a repeating attribute, with each index position representing one activity instance. To obtain the correct sequence number, query for the sequence number at the same index position as the activity's name (Each activity in a workflow must have a unique name within the workflow.) The activity instance's name is recorded in the `r_act_name` attribute. ([Forcing index correspondence in query results, page 346](#), contains instructions for querying to obtain values as corresponding index positions.)

Resuming a halted workflow or activity

Use a Resume method to resume execution of a halted workflow or activity instance.

Resuming a workflow returns any work items in the D/A/P paused state work items to their previous state, changes the halted activity instances to the running state, and changes the workflow's state to running.

Resuming a paused activity instance returns paused work items to their previous state (dormant or acquired) and changes the activity instance's state to running.

Restarting a halted workflow or failed activity

Restarting a workflow removes all generated work items and packages and restarts the workflow from the beginning, with all activity instances set to dormant.

Restarting a failed activity sets the activity's state to Active.

To restart a halted workflow or failed activity, use a Restart method.

Aborting a workflow

Aborting a workflow terminates the workflow and sets the `r_runtime_state` attribute to terminated, but does not remove the workflow's runtime objects from the repository.

To remove the aborted workflow and its associated runtime objects, such as packages and the work items, you can use a Destroy method or you can run `dmclean` with the `-clean_aborted_wf` argument set to T.

Use an Abort method to terminate a workflow. You must be the workflow supervisor or a user with Sysadmin or Superuser privileges. You cannot abort a workflow if any automatic work items are in the acquired state.

Pausing and resuming work items

You can pause a dormant work item. Work items are dormant until they are acquired or a user delegates the work item. You cannot pause an acquired work item.

To pause a dormant work item, you must be the workflow supervisor or a user with Sysadmin or Superuser privileges. Use a Pause method to pause a work item.

Resuming a paused work item returns the work item to the dormant state. To resume a paused work item, you must be the workflow supervisor or a user with Sysadmin or Superuser privileges. Use a Resume method.

Modifying a workflow definition

You can change a workflow definition by changing its process definition or the activity definitions.

Changing process definitions

When you change a process definition, you can either overwrite the existing definition with the changes or create a new version of the definition. Any changes you make are governed by object-level permissions.

Overwriting a process definition

To make changes to a process definition and save the changes without versioning, you must uninstall the process definition. To uninstall a process definition requires Relate permission on the definition or Sysadmin or Superuser privileges. To save your changes requires Write permission.

Uninstalling a process definition:

- Moves the definition to the validated state
- Halts all running workflows based on that definition

Uninstalling a process definition does not affect the state of the activity definitions included in the process definition.

If you change attributes defined for the `dm_process` object type, the server changes the definition state to draft when you save the changes. You must validate and reinstall the definition again.

If you change only inherited attributes (those inherited from `dm_sysobject`), the definition remains in the validated state when you save the changes. You must reinstall the definition, but validating it isn't necessary.

Versioning process definitions

Versioning a process definition has no impact on the running workflows based on the definition. You must have at least Version permission on the process object to create a new version of the definition. Use a Checkout or Branch method to obtain the process object for versioning. You can version a process definition without uninstalling the definition.

Note: If you have installed Business Process Manager and have created an email template that is associated with the process (workflow) definition, versioning the definition has no effect on the template. The template will be used for both the previous and the new version of the workflow definition.

When you check in (or save, for branching) your changes, the server sets the new version to the draft state. The new version must be validated and installed before you can start a workflow based on it.

Reinstalling after making changes

If you are overwriting the existing definition, after you save the changes, you must reinstall the definition. If you made changes to any of the attributes defined for the process object type, you must re-validate and then reinstall the process definition.

When you reinstall, you can choose how you want to handle any workflows that were halted when you uninstalled the process definition. You can choose to resume the halted workflows at the point from which they were halted. Or, you can choose to abort the workflows. Which option you choose depends on the changes you made to the workflow. Perhaps you added an activity that you want to perform on all objects in the workflow. In that case, you abort the workflows and then start each again.

Content Server does not automatically restart the aborted workflows. If you want to execute the aborted workflows again, you must issue an Execute method again to start them.

The default behavior when a process definition is reinstalled is to resume all halted workflows that reference that definition.

Changing activity definitions

Although an activity object is a SysObject subtype, you cannot version activities. You can only issue Save or Saveasnew methods on an existing activity definition. Using Save overwrites the existing definition. Using Saveasnew creates a copy of the definition.

Overwriting an activity definition

To make changes to an activity definition, uninstall the activity definition. To uninstall an activity definition, you must have Relate permission on all process definitions that include the activity definition or have Sysadmin or Superuser privileges.

Uninstalling an activity definition:

- Moves the definition to the validated state
- Uninstalls all process definitions that include the activity definition

Uninstalling a process definition moves the definition to the validated state and halts all running workflows based on the definition.

If you change attributes defined for the dm_activity object type, the server changes the definition state to draft when you save the changes. You must validate and reinstall the definition.

If you change only inherited attributes (those inherited from dm_sysobject), the definition remains in the validated state when you save the changes. You must reinstall the definition, but validating it isn't necessary.

Adding ports and package definitions

When you add a port to an activity definition, the port's name must be unique within the activity. After you add a port, you must add at least one package definition for the port.

Removing ports and package information

When you remove a port from an activity, you must also remove all package definitions associated with the port. Removing a port is implemented using a Removeport method.

Removing package definitions is done using `Removepackageinfo` methods. Each execution of a `Removepackageinfo` method removes one package definition.

Saving the changes

When you save changes to an uninstalled activity definition, the server sets all the process definitions that include the activity definition back to the draft state.

Destroying process and activity definitions

Destroying a process or activity definition removes it from the repository. To destroy a process or activity definition:

- The definition must be in the validated or draft state.
- The definition cannot be in use by any workflows.
- You must have Delete permission or Sysadmin or Superuser privileges.

Note: Any email templates you may have associated with a process or activity definition are not destroyed when the definition is destroyed. (Email templates are a feature of Business Process Designer. For more information about them, refer to the Business Process Designer documentation.)

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 different repository than the repository to which he or she is currently connected.

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's 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's home repository.
3. User A connects to the home repository and acquires the queue item. The user's 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's 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's 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's 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's 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.

Remote object routing

You can route a remote object (a SysObject or its subtype) using either the Addpackageinfo or Addpackage methods.

Use `Addpackageinfo` if you are identifying the remote object in a package definition when you design the activity. In this case, `Addpackageinfo` creates a reference link on behalf of the user. When the package is routed to the user, the user connects to the source repository and works on the object indirectly through the reference link (which requires the user to be able to connect to the repository where the object resides).

Use `Addpackage` if an activity performer is adding the package containing the object at runtime.

In either case, use the remote object ID or the reference link ID for the remote object (the reference link ID is the object ID of the mirror object that is part of the reference link).

Lifecycles

This chapter describes lifecycles, one of the process management services provided with Content Server. The chapter includes the following topics:

- [Introducing lifecycles, page 263](#)
- [Default lifecycles for object types, page 266](#)
- [Lifecycle definitions, page 267](#)
- [How lifecycles work, page 268](#)
- [Object permissions and lifecycles, page 275](#)
- [Integrating lifecycles and applications, page 276](#)
- [Designing a lifecycle, page 278](#)
- [Creating lifecycles, page 292](#)
- [Implementing a custom validation program, page 295](#)
- [Modifying lifecycles, page 297](#)
- [Getting information about lifecycles, page 298](#)
- [Deleting a lifecycle, page 299](#)

Introducing lifecycles

A lifecycle is a set of states that define the stages in an object's life. 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 may also have actions to be performed after entering a state.

For example, a lifecycle for an SOP (Standard Operating Procedure) might have the states draft, review, rewrite, approved, and obsolete. Before an SOP can move from the rewrite state to the approved state, business rules may 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 the employees informing them the SOP is available.

Normal and exception states

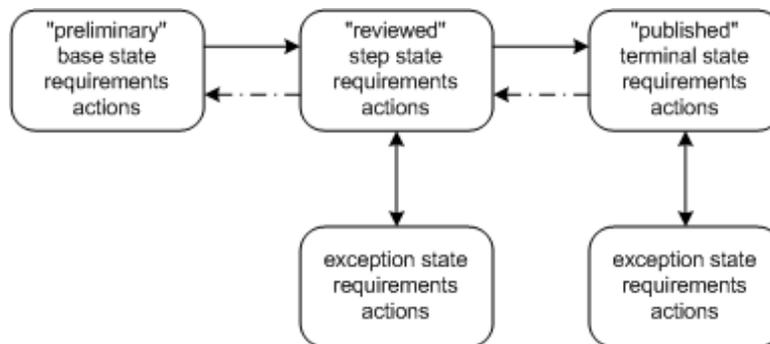
There are two kinds of states: normal and exception. Normal states are the states that define the typical stages of an object's life. Exception states represent situations outside of the normal stages of an object's life. All lifecycles must have normal states. Exception states are optional. Each normal 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's 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.

Figure 10-1, page 264, shows an example of a lifecycle with exception states. Like normal states, exception states have their own requirements and actions.

Figure 10-1. A 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 stages are for those objects. Then, decide whether any or all of those stages require an exception state.

Types of objects that may 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.

Exactly which types of objects a particular lifecycle handles is specified when you define the lifecycle. Lifecycles are a reflection of the stages 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 attributes: `included_type` and `include_subtypes`. These are repeating attributes. The `included_type` attribute records, by name, the object types that can be attached to a lifecycle. The `include_subtypes` attribute is a Boolean attribute 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's `included_type` attribute contains the document's type, or
- The lifecycle's `included_type` contains the document's supertype and the value at the corresponding index position in the `include_subtypes` attribute is set to `TRUE`

For example, suppose a lifecycle definition has the following values in those attributes:

```
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.

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. If you choose Java, any program that you write for any state in the lifecycle must be Java. If you choose Docbasic, any program that you write for any state in the lifecycle must be Docbasic.

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's definition or a superuser can bypass entry criteria.

[Entry criteria definitions, page 282](#), contains more information about defining entry criteria. For more information about defining actions on entry, refer to [Actions on entry definitions, page 285](#). And for more information about defining post-entry actions, refer to [Post-entry action definitions, page 287](#).

Default lifecycles for object types

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.

Lifecycle definitions

The definition of a lifecycle is stored in the repository as a `dm_policy` object. The attributes 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 attributes. The values at a particular index position across those attributes represent the definition of one state. The sequence of states within the lifecycle is determined by their position in the attributes. The first state in the lifecycle is the state defined in index position [0] in the attributes. The second state is the state defined in position [1], the third state is the state defined in index position [2], and so forth.

State definitions include such information as the name of the state, a state's type, whether the state is a normal or exception state, entry criteria, and actions to perform on objects in that state. (For a detailed list of the information that makes up a state definition, refer to [Lifecycle state definitions, page 278](#).)

Draft, validated, and installed definitions

Lifecycle definitions are stored in the repository in one of three states: 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 may be installed. Only after a lifecycle definition is installed may users attach the lifecycle to objects.

The state of a lifecycle definition is recorded in the `r_definition_state` attribute of the policy object.

Validation

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 Content 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` attribute are subtypes of the primary attachable type.
- All objects referenced by object ID in the policy definition exist.

- For Java-based lifecycles, that all SBO objects 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, Content Server executes the system-defined validation first and then the custom program. Both programs must complete successfully to successfully validate the definition. For instructions on writing a custom validation program, refer to [Implementing a custom validation program, page 295](#).

Validating a lifecycle definition requires at least Write permission on the policy object.

Installation

Lifecycles that have passed validation may 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.

How lifecycles work

This section provides a general description of lifecycle use and behavior and the supporting methods.

General overview

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, Content 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's `r_policy_id` attribute
- 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` attribute 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.

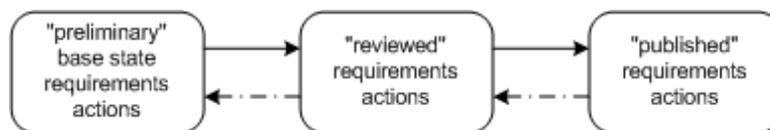
Each time the object is moved forward to a normal state or to an exception state, Content Server evaluates the entry criteria for the target state. If the object satisfies the criteria, the server performs the entry actions, and resets the `r_current_state` attribute to the number of the target state. If the target state is an exception state, Content 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 attributes) to an object as that object progresses through the lifecycle.

If an object is demoted back to the previous normal state, Content Server only performs the actions associated with the state and resets the attributes. It doesn't evaluate the entry criteria.

Objects cannot skip normal steps as they progress through a lifecycle.

Figure 10-2, page 269, is 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



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, Content 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` attribute is set for the state. (For more information about attachability, refer to [Attachability, page 279](#).)

When an object is attached to a state, Content 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.

Internally, attaching an object is accomplished using an `attach` method.

Movement 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.

Promotions

Promotion moves an object from one normal 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, Content Server will attempt to promote the object as the user defined in the `a_bpaction_run_as` attribute 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's policy object or be a superuser to bypass entry criteria.

Promotions are accomplished internally using a `promote` method. Bypassing the entry criteria is accomplished by setting the `override_flag` argument in the method to `TRUE`.

Batch promotions

Batch promotion is the promotion of multiple objects in batches. Content Server supports batch promotions using the `BATCH_PROMOTE` administration method. You can use it to promote multiple objects in one operation. Refer to [BATCH_PROMOTE, page 175](#), of the *Content Server DQL Reference Manual* for details and instructions.

Demotions

Demotion moves an object from a normal 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` attribute for the state must be `TRUE`.

Additionally, to demote an object back to the base state, the `return_to_base` attribute 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, Content Server will attempt to demote the object as the user defined in the `a_bpaction_run_as` attribute 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's current state is a normal state, the object can be demoted to either the previous normal state or the base state. If the object's current state is an exception state, the object can be demoted only to the base state. Demotions are accomplished internally using a `demote` method.

Suspensions

Suspension moves an object from the object's current normal state to the state's 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, Content Server will attempt to suspend the object as the user defined in the `a_bpaction_run_as` attribute 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's 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's policy object or be a superuser to bypass entry criteria.

Suspending an object is accomplished internally using a `suspend` method. Bypassing the entry criteria is accomplished by setting the `override_flag` argument in the method set to `TRUE`.

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, Content Server will attempt to resume the object as the user defined in the `a_bpaction_run_as` attribute 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` attribute set to `TRUE`.

When an object is resumed to either the normal state or the base state, the object must satisfy the target state's 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's policy object or be a superuser to bypass entry criteria.

Internally, resuming an object is accomplished using a resume method. Bypassing the entry criteria is accomplished by setting the `override_flag` argument in the method set to `TRUE`.

Scheduled transitions

A scheduled transition is a transition from one state to another at a pre-defined 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 an argument that allows you to schedule a transition for a particular date and time. If the argument is included when the method is issued, Content Server creates a job for the state change. The job's name is set to:

```
Bp_<SysObject_ID><scheduled_transition_time>
```

The `SysObject_ID` is the object ID of the object that is moved from one state to another. The `scheduled_transition_time` is the date and time the transition is expected to take place. It is formatted as:

```
<4-digit year><2-digit month><2-digit day><2-digit hour><2-digit  
minute><2-digit second>
```

For example: `Bp_090017fd6000291f20050111452145`

The job's scheduling attributes 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` attribute is set in the `docbase` config. If that is set, the job runs as the user defined in that attribute.

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. For instructions, refer to the Lifecycle Editor documentation.

Internal supporting methods

Installing Content 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. [Table 10-1, page 273](#), lists the methods.

Table 10-1. Lifecycle methods

Method name		
Java	Docbasic	Purpose
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. (For information about BATCH_PROMOTE, refer to BATCH_PROMOTE, page 175 , in the <i>Content Server DQL Reference Manual</i> .)
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.

How state changes work

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's scheduling attributes 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`. (For more information about the jobs, refer to [Scheduled transitions, page 272](#).)

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 Content Server.
2. If the policy does not allow the object to move from the current state to the next state, returns an error and exits.
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 above steps fails, abort the transaction and retur.
8. Set the `r_current_state` and `r_resume_state` attributes. For an attach method, also sets the `r_policy_id` and `r_alias_set_id` attributes.
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 processing run as the user who issued the state-change method. To change the default, you must set the `a_bpaction_run_as` attribute in the docbase config object. If the `a_bpaction_run_as` attribute is set in the docbase config

object, the actions associated with state changes are run as the user indicated in the attribute. Setting `a_bpaction_run_as` can ensure that users with the extended permission Change State but without adequate access permissions to an object are able to change an object's state. If the attribute 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, it is recommended that you also set the `client_session_timeout` key in the `server.ini` 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 Content Server, the client session will time out if the `client_session_timeout` value is five minutes. Setting `client_session_timeout` to a value greater than the value specified in `timeout_default` prevents that from happening.

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's 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's 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's owner or a superuser. Moving the document requires the Change Location permission and the appropriate base object-level permission. (The required object-level permissions are described in the Link and Unlink method descriptions, in the *Content Server API Reference Manual*.)

The actions associated with a state can be used to reset permissions as needed.

Object-level permissions, including the extended permissions, are described in [Chapter 4, Security Services](#).

Integrating lifecycles and applications

This section discusses the lifecycle features that make it easy to integrate lifecycles and applications.

Lifecycles, alias sets, and aliases

A lifecycle definition can reference one or more alias sets. When an object is attached to the lifecycle, Content 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 attributes. (Sysobjects can use aliases in the `owner_name`, `acl_name`, and `acl_domain` attributes.) 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. (The default resolution is described in [Determining the lifecycle scope for SysObjects](#), page 313.)

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. For more information, refer to [Using aliases in actions](#), page 291.

If you define one or more alias sets for a lifecycle definition, those choices are recorded in the policy object's `alias_set_ids` attribute.

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: Content 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 extension 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 attributes 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 attributes: `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 attributes 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` attribute, inherited from the `dm_state_extension` supertype.

State extensions must be created manually. The Lifecycle Editor does not support creating state extensions. For instructions about creating a state extension subtype and adding a state extension to a lifecycle state, refer to [Adding state extensions, page 294](#).

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.

EMC Documentum Document Control Management (DCM) and EMC Documentum Web Content Management (WCM) expect the states in a lifecycle to have certain state types. The behavior of either Documentum client 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 DCM or WCM, the lifecycle states must have state types that correspond to the state types expected by the client. (Refer to the DCM and WCM documentation for the state type names recognized by each.)

Custom applications can also use state types. Applications that handle and process documents can examine the `state_type` attribute 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 attribute 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 attributes: `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 DCM creates state type objects for the state types recognized by DCM. DCM 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.

Designing a lifecycle

When you design a lifecycle, make the following decisions:

- What objects will use the lifecycle

[Types of objects that may be attached to lifecycles, page 265](#), provides information about choosing object types for a lifecycle.
- What normal and exception states the lifecycle will contain, and for each state, what is the definition of that state

[Normal and exception states, page 264](#), describes normal and exception states. [Lifecycle state definitions, page 278](#), contains guidelines for defining states.
- Whether to include an alias set in the definition

[Lifecycles, alias sets, and aliases, page 276](#), provides information for this decision.
- Whether you want to assign state types

If objects attached to the lifecycle will be handled by the Documentum clients DCM or WCM, 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's state type, you must assign state types. For more information about state types, refer to [State types, page 277](#).
- What states, if any, will have state extensions

[State extensions, page 276](#), describes the use of state extensions.
- Whether you want to use a custom validation program

[Validation, page 267](#), provides information about using a custom validation program.

Lifecycle state definitions

Each state in the lifecycle has a state definition. All of the information about states is stored in attributes in the `dm_policy` object that stores the lifecycle definition. For example, whether a state is a normal or exception state is recorded in the `state_class` attribute. The attributes that record a state definition are repeating attributes, and the values at a particular index position across the attributes represent the definition of one state in the lifecycle. If you are using the Lifecycle Editor to create a lifecycle, these attributes are set automatically when you create the definition. If you are creating a lifecycle outside the Editor, you must set the attributes yourself.

You must define a state's basic characteristics, and optionally, the state's type, entry criteria, actions on entry, and post-entry actions.

The basic characteristics are:

- The state's 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` attribute of the `dm_policy` object holds the names of the states.
- For normal states, whether users can attach an object to the state
[Attachability, page 279](#), describes this characteristic of states.
- Whether objects can be returned to the base state from the state after Save, Checkin, Saveasnew, or Branch operations
[Return to base state setting, page 280](#), provides information about this characteristic.
- Whether objects can be demoted from the state
[Demoting from a state, page 281](#), provides information about this characteristic.
- Whether users can schedule transitions.
[Allowing scheduled transitions, page 281](#), provides information for this decision.
- The state's type, if any
[State types, page 277](#), has guidelines on whether state types are required.
[Entry criteria definitions, page 282](#) describes the options for defining entry criteria. The options for defining actions on entry are described in [Actions on entry definitions, page 285](#). And [Post-entry action definitions, page 287](#) describes the options for defining post-entry actions.

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. How many states in a lifecycle allow attachments will depend on the lifecycle. For example, if a lifecycle handles only documents created within the business, you may want to allow attachments only for the first state, the stage at which documents are created. If the lifecycle also handles document drafts that are imported from external sources, you may want to allow both the creation and review stages to allow attachments.

If a state allows attachments, users can attach objects to the lifecycle at that state in the lifecycle, skipping any prior states.

Only normal states can allow attachments. An exception state cannot allow attachments.

Whether a state allows attachments is defined in the `allow_attach` attribute. This is a Boolean attribute.

Return to base state setting

Some operations that users may perform on an object may trigger a business rule that requires the object to be returned to the base state in its lifecycle. For example, checking in an object creates a new version of the object, and business rules might require the new version to start its life at the beginning of the lifecycle. When you define a state, you indicate whether to allow objects to be returned to the base state automatically from that state. If you allow that, the Lifecycle Editor lets you choose from the following operations to trigger the return to base operation:

- Checkin, Save, and Saveasnew operations
- Checkin operation only
- Save operation only
- Saveasnew operation only
- Branch operation only

You can choose any or all of the options as the trigger returning an object to the base state. For checkin and branch operations, the new version is returned to the base state (note that for the branch operation, the return-to-base occurs when the new version is saved). For saveasnew operations, the new copy of the object is returned to the base state. For save operations, the saved object is returned to the base state.

The return-to-base behavior is controlled internally by two attributes in the policy object: `return_to_base` and `return_condition`. These are repeating attributes. Each index position corresponds to a lifecycle state. `return_to_base` is a Boolean attribute that controls whether an object in a particular state can be returned to the base state. `return_condition` is an integer attribute that identifies what operations cause the object to be returned to the base state. `return_condition` can be any of the following values or their sums:

- 0, for the Checkin, Save, and Saveasnew operations
- 1, for the Checkin operation only
- 2, for the Save operation only
- 4, for the Saveasnew operation
- 8, for the Branch operation

For example, if `return_to_base[3]` is set to `TRUE` and `return_condition[3]` is set to 1, whenever an object in the corresponding state is checked in, the new version is returned to the base state. If you chose multiple operations to trigger a return to base for a state, then `return_condition` would be set to the sum of those operations for that state. For example, suppose you chose to trigger a return to base for save and saveasnew operations for the fourth state. In this case, `return_condition[3]` would be set to 6 (the sum of 2 + 4).

The default setting for `return_to_base` for all states is `FALSE`, which means that objects remain in the current state after a checkin, save, saveasnew, or branch. The default setting for `return_condition` for all states is 0, meaning that a return to base occurs after Checkin, Save, and Saveasnew operations if `return_to_base` is `TRUE` for the state.

When an object is returned to the base state, the object is tested against the base state's entry criteria. If those succeed, the actions on entry are performed. If either fails, the checkin or save method fails. If it both succeed, the checkin or save succeeds and the state's actions are executed. If the actions don't succeed, the object remains in the base state. (The checkin or save cannot be backed out.)

Demoting from a state

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's 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` attribute to `TRUE` for that state.

When an object is demoted, Content Server does not check the entry criteria of the target state. However, Content Server does perform the system and user-defined actions on entry and post-entry actions.

Allowing 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 scheduled transition out of a state is allowed for a particular state is recorded in the `allow_schedule` attribute. This attribute 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 attribute 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.

For information about how scheduled transitions are implemented internally, refer to [Scheduled transitions, page 272](#).

Entry criteria definitions

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
- One or more Boolean expressions
- Both Boolean expressions and a Java program

Java-based programs are stored in the repository as a simple module and a jar file. For information about simple modules, refer to the *DFC Development Guide*.

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

Using a Java program or Docbasic program for the entry criteria lets you define complex conditions for entry. You can also use the program to enforce a sign-off requirement on the current state before the object is promoted to the next state. If you define both Boolean expressions and a program, the expressions are evaluated first and then the program. The expressions must evaluate to TRUE and the program must complete successfully before the object is moved to the new state.

[Java programs as entry criteria, page 282](#) describes how to create a Java program for entry criteria. [Docbasic programs as entry criteria, page 283](#), describes how to create a Docbasic program for entry criteria. [Boolean expressions as entry criteria, page 284](#), contains information about defining a simple Boolean expression. For information about adding a signature requirement to a program, refer to [Including electronic signature requirements, page 291](#).

Java programs as entry criteria

A Java program used as entry criteria must implement the following interface:

```
public interface IDfLifecycleUserEntryCriteria
{
    public boolean userEntryCriteria(IDfSysObject obj,
                                     String userName,
                                     String targetState)
        throws DfException;
}
```

A session argument is not needed, as the session is retrieved from the IDfSysObject argument.

Any attributes referenced in the program must be attributes defined for the primary object type of the lifecycle. (The primary object type is the object type identified in `include_type[0]`.) The function cannot reference attributes inherited by the primary object type.

If `userEntryCriteria` returns false or an exception is thrown, Content Server assumes that the object has not satisfied the criteria.

After you write the program, package it in a jar file. When you add the program to a state's entry criteria, the Lifecycle Editor creates the underlying module and jar objects that associate the program file with the lifecycle.

Note: You package entry criteria, actions on entry, and post-processing action programs in a single jar file. Additionally, the jar object created to store the jar file may be versioned. However, Content Server will always use the jar file associated with the CURRENT version of the jar object.

For an example of an entry criteria program, refer to [Sample implementations of Java interfaces, page 289](#).

Docbasic programs as entry criteria

To define entry criteria more complex than an expression, use a Docbasic function. The function must be named `EntryCriteria` and must have the following format:

```
Public Function EntryCriteria(
    ByVal SessionID As String,
    ByVal ObjectId As String,
    ByVal UserName As String,
    ByVal TargetState As String,
    ByRef ErrorStack As String) As Boolean
```

Use the `ErrorStack` to pass error messages back to the server. Error code 1500 is prefixed to the `ErrorStack`. Set the returned value of `EntryCriteria` to FALSE upon error and TRUE otherwise.

Any attributes referenced in the function must be attributes defined for the primary object type of the lifecycle. (The primary object type is the object type identified in `include_type[0]`.) The function cannot reference attributes inherited by the primary object type.

To run successfully, the scripts must be created on a host machine that is using the same code page as the host on which the actions will execute or the scripts must use only ASCII characters.

When you add the program to a lifecycle state's entry criteria, the program is stored in the repository as a `dm_procedure` object. In turn, the object ID of the procedure object is stored in the policy's repeating attribute, `user_criteria_id`. Each index position can contain one procedure object ID. The program represented by the object ID at a particular

index position in the `user_criteria_id` attribute is applied to objects entering the state identified in the corresponding index position in the `state_name` attribute.

Because procedure objects can be versioned, policy objects have the repeating attribute `user_criteria_ver` to allow you to late-bind a particular version of a procedure to a state as entry criteria. The index positions in `user_criteria_ver` correspond to those in `user_criteria_id`. For example, if you identify a version label in `user_criteria_ver[2]`, when the server runs the procedure identified in `user_criteria_id[2]`, it runs the version of the procedure that carries the label specified in `user_criteria_ver[2]`.

If there is no version label defined in `user_criteria_ver` for a particular procedure, the server runs the version identified by the object ID in `user_criteria_id`.

If you are not creating the lifecycle using the Lifecycle Editor, use a Set method to set the `user_criteria_id` and `user_criteria_ver` attributes.

Boolean expressions as entry criteria

A Boolean expression is any comparison expression that resolves to true or false. For example:

```
title=MyBook
```

You can include one or more Boolean expressions as entry criteria for a state. Attributes referenced in the expressions must be attributes defined for the lifecycle's primary object type (the object type identified in `included_type[0]`). The expressions cannot reference attributes inherited by that object type. For example, the expression `title=MyBook` references the `title` attribute, which is defined for the `dm_sysobject` type. That means that the lifecycle that includes the state for which this criteria is defined must have `included_type[0]` set to `dm_sysobject`.

Entry criteria are typically defined for a state using Lifecycle Editor. However, you can add a Boolean expression to a lifecycle state by setting the `_entry_criteria` computed attribute. To define criteria for a particular state, set the computed attribute at the index position corresponding to the state's index position in the `state_name` attribute. For example, setting `_entry_criteria_id[1]` defines entry criteria for objects entering the state identified in `state_name[1]`. To illustrate, the following statement defines the expression `title=Monthly Report` as entry criteria for the state identified in `state_name[1]` of the lifecycle identified by the object ID 4600000123541321:

```
dmAPISet ("set,s0,4600000123541321,_entry_criteria[1]",  
"title='Monthly Report'")
```

To define multiple expressions, use a variable. For example:

```
expression$="title=""MonthlyReport""and r_page_count<=1  
and authors(0)=JohnP"  
status$=  
dmAPISet ("set,"&sess & "," & pol_id & ",entry_criteria(0)",expression)
```

The expressions are stored in a func expr object and the object ID of the func expr object is recorded in the entry_criteria_id attribute. The entry_criteria_id attribute is a repeating attribute, and each index position can contain the object ID of one func expr object. The expressions stored in the func expr object at a particular index position are applied to the state defined by the corresponding index position.

To remove entry criteria, set the _entry_criteria attribute at the corresponding index position to an empty string or to a single space.

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's ACL, changing a object's repository location, or changing an object's 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. (For a listing of the pre-defined actions, refer to the Documentum Application Builder documentation.)

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 simple modules and a jar file. For information about defining Java programs for actions on entry, refer to [Java programs as actions on entry, page 286](#). The *DFC Development Guide* describes simple modules.

If the lifecycle is Docbasic-based, the actions on entry program must be a Docbasic program. For information about using Docbasic programs, refer to [Docbasic programs as actions on entry, page 286](#)

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.

Using system-defined actions on entry

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. If you are using a Java-based lifecycle, the actions are stored as service-based objects (SBO) and recorded in the system_action_state attribute. If you are using a Docbasic-based lifecycle, the chosen actions are recorded in the action_object_id attribute.

Note: The programs for system-defined actions for Docbasic-based lifecycles are found in the `bp_actionproc.ebs` script. The procedure object associated with this script is called `dm_bpactionproc`. Typically, actions from this script are selected using the Lifecycle Editor. However, you can use the actions defined in this script independently.

Java programs as actions on entry

A Java program used as an action on entry program must implement the following interface:

```
public interface IDfLifecycleUserAction
{
    public void userAction(IDfSysObject obj,
                          String userName,
                          String targetState)
        throws DfException;
}
```

A session argument is not needed, as the session is retrieved from the `IDfSysObject` argument.

All attributes referenced by the operations in the program must be defined for the lifecycle's primary object type, which is identified in `included_type[0]` in the lifecycle definition.

After you write the program, package it in a jar file. When you add the program to a state's actions on entry, the Lifecycle Editor creates the underlying module and jar objects that associate the program file with the lifecycle.

Note: You can package the actions on entry, post-processing actions, and entry criteria programs in a single jar file. Additionally, the jar object created to store the jar file may be versioned. However, Content Server will always use the jar file associated with the CURRENT version of the jar object.

Unless the program throws an exception, Content Server assumes that the actions are successful.

For an example of an actions on entry program, refer to [Sample implementations of Java interfaces](#), page 289.

Docbasic programs as actions on entry

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` attribute. These attributes are set internally when you identify the programs while creating or modifying a lifecycle using Lifecycle Editor.

User-defined Docbasic programs for actions on entry must represent a function named `Action` that has the following format:

```
Public Function Action(  
    ByVal SessionID As String,  
    ByVal ObjectID As String,  
    ByVal UserName As String,  
    ByVal TargetState As String,  
    ByRef ErrorStack As String) As Boolean
```

Use the `ErrorStack` to pass error messages back to the server. Error code 1600 is prefixed to the `ErrorStack`. Set the returned value of `Action` to `FALSE` upon error and `TRUE` otherwise.

Any attributes referenced in the function must be attributes defined for the primary object type of the lifecycle (the lifecycle's primary object type is defined in `included_types[0]`). The function cannot reference attributes inherited by the primary object type.

To run successfully, the scripts must be created on a host machine that is using the same code page as the host on which the actions will execute or the scripts must use only ASCII characters.

When you add the procedure to the actions on entry for a lifecycle state, Content Server creates a procedure object for the program and sets the `user_action_id` to the object ID of that procedure object. The `user_action_id` attribute is a repeating attribute. The procedure represented by the procedure object ID at a particular index position is applied to objects attempting to enter the state at the corresponding index position in `state_name`. The procedure must return successfully before the object can enter the state.

Because procedure objects can be versioned, policy objects have the repeating attribute `user_action_ver` to allow you to late-bind a particular version of a procedure to a state as an action. The index positions in `user_action_ver` correspond to those in `user_action_id`. For example, if you identify a version label in `user_action_ver[2]`, when the server runs the procedure identified in `user_action_id[2]`, it runs the version of the procedure that carries the label specified in `user_action_ver[2]`.

If there is no version label defined in `user_action_ver` for a particular procedure, the server runs the version identified by the object ID in `user_action_id`.

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 Java-based, the post-entry action programs must be Java programs. The programs are stored in the repository as simple modules and a jar file. [Java programs as post-entry actions, page 288](#), describes how to create a Java post-entry action program. The *DFC Development Guide* describes simple modules.

If the lifecycle is Docbasic-based, the post-entry action programs must be Docbasic programs. [Docbasic programs as post-entry actions, page 288](#), describes how to create a Docbasic program.

Java programs as post-entry actions

A Java program used as an post-entry action program must implement the following interface:

```
public interface IDfLifecycleUserPostProcessing
{
    public void userPostProcessing(IDfSysObject obj,
                                   String userName,
                                   String targetState)
        throws DfException;
}
```

A session argument is not needed, as the session is retrieved from the IDfSysObject argument.

After you write the program, package it in a jar file. When you add the program to a state's post-entry actions, the Lifecycle Editor creates the underlying module and jar objects that associate the program file with the lifecycle.

Note: You can package post-entry actions, actions on entry, and entry criteria programs in a single jar file. Additionally, the jar object created to store the jar file may be versioned. However, Content Server will always use the jar file associated with the CURRENT version of the jar object.

Unless the program throws an exception, Content Server assumes that the actions are successful.

For an example of a post-entry actions program, refer to [Sample implementations of Java interfaces, page 289](#).

Docbasic programs as post-entry actions

Docbasic post-entry actions are functions named PostProc. The PostProc function must have the following format:

```
Public Function PostProc(
    ByVal SessionID As String,
    ByVal ObjectID As String,
    ByVal UserName As String,
```

```
ByVal TargetState As String,
ByRef ErrorStack As String) As Boolean
```

Use the `ErrorStack` to pass error messages back to the server. Error code 1700 is prefixed to the `ErrorStack`. Set the returned value of `PostProc` to `FALSE` upon error and `TRUE` otherwise. Note that any errors encountered when the functions run are treated by the server as warnings.

Any attributes referenced in the function must be attributes defined for the primary object type of the lifecycle. The function cannot reference attributes inherited by the primary object type.

To run successfully, the scripts must be created on a host machine that is using the same code page as the host on which the actions will execute or the scripts must use only ASCII characters. The scripts are stored in the repository in `dm_procedure` objects, and the object IDs of the procedure objects are recorded in the lifecycle definition in the `user_postproc_id` attribute.

The `user_postproc_id` attribute is a repeating attribute. The value at each index position identifies one procedure object. The procedure object contains the post-entry actions to be performed for the state identified at the corresponding index position in `state_name`.

If you are not using Lifecycle Editor to create the lifecycle, use a `Set` method to set the `user_postproc_id` attribute.

Because procedure objects can be versioned, policy objects have the repeating attribute `user_postproc_ver` to allow you to late-bind a particular version of a procedure to a state as a post-entry action. The index positions in `user_postproc_ver` correspond to those in `user_postproc_id`. For example, suppose you specify the version label `CURRENT` in `user_postproc_ver[2]`. When the server runs the post-processing procedure for the state represented by index position [2], instead of running the version identified in `user_postproc_id[2]`, it searches the procedure's version tree, finds the `CURRENT` version, and runs the `CURRENT` version. When you specify a version in `user_postproc_ver`, the version carrying that version label is always used.

If there is no version label defined in `user_postproc_ver` for a particular procedure, the server runs the version identified by the object ID in `user_postproc_id`.

Sample implementations of Java interfaces

Here is a sample Java package that contains interface implementations for entry criteria, actions on entry, and post-entry programs.

```
package LifecyclePackage;

import com.documentum.fc.lifecycle.*;
import com.documentum.fc.client.*;
import com.documentum.fc.common.DfException;
```

```
import com.documentum.fc.common.DfTime;
import com.documentum.fc.common.IDfTime;

public class PublishedState
    implements IDfModule
        IDfLifecycleUserEntryCriteria,
        IDfLifecycleUserAction,
        IDfLifecycleUserPostProcessing
{
    private boolean isReviewSignedOff (IDfSysObject obj)
        throws DfException
    {
        IDfQuery query = new DfQuery();
        query.setDQL ("select user_name from dm_audittrail " +
            "where event_name = 'dm_signoff' and " +
            "audited_obj_id = '" + obj.getObjectId().toString() + "' and " +
            "string_2 = 'REVIEWED'");

        IDfCollection collection = query.execute (obj.getSession(), DfQuery.DF_READ_QUERY);
        boolean result = collection.next();
        collection.close();
        return result;
    }

    private void publishSignOff (IDfSysObject obj, String userName)
        throws DfException
    {
        obj.signoff (userName, "", "PUBLISHED");
    }

    public boolean userEntryCriteria(IDfSysObject obj,
        String userName,
        String targetState)
        throws DfException
    {
        return isReviewSignedOff (obj);
    }

    public void userAction(IDfSysObject obj,
        String userName,
        String targetState)
        throws DfException
    {
        obj.mark ("PUBLISHED");
    }

    public void userPostProcessing (IDfSysObject obj,
        String userName,
        String targetState)
        throws DfException
    {
        publishSignOff (obj, userName);
    }
}
```

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.

Complete information about using digital and electronic signatures or simple sign-offs is found in the *Content Server Administrator's Guide*.

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 attributes of a SysObject. You can use an alias in place of path name in the Link and Unlink methods. (For details of how aliases are implemented and used, refer to [Appendix A, Aliases](#).)

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.

Creating lifecycles

Lifecycles are typically created using the Lifecycle Editor. 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.

Basic procedure

You can create a new lifecycle from the beginning or you can copy an existing lifecycle and modify it. You must have Create Type, Sysadmin, or Superuser privileges to create or copy a lifecycle (dm_policy object).

The basic steps are:

1. Design the lifecycle.
Use the information in [Designing a lifecycle, page 278](#), to design the lifecycle.
2. Create a draft lifecycle definition.
Refer to [Draft, validated, and installed definitions, page 267](#), for information about draft lifecycles.
3. If using state extensions, create the state extension subtype and object instances.
Refer to [Adding state extensions, page 294](#), for instructions.
4. If using a custom validation program, create and install the program.
Refer to [Implementing a custom validation program, page 295](#), for instructions.
5. Validate the draft lifecycle definition.
Refer to [Validation, page 267](#) information about lifecycle validation.
6. Install the validated lifecycle definition.

Testing and debugging a lifecycle

When you develop a lifecycle, you may want to test the lifecycle, to ensure that the entry criteria and actions are functioning correctly. And you may need to debug one or more operations. Use the information in this section to help with these processes.

Testing a lifecycle

During lifecycle development, you can use promote and resume methods to test state entry criteria and actions. Both these methods have a `test_flag` argument. If the argument is set to `TRUE`, the method performs the operations associated with the method without actually moving the object.

Executing in test mode launches the transition method (`dm_bp_transition_java` or `dm_bp_transition`), which tests to see whether requirements are met and the actions can succeed, but the server does not promote or resume the object. The server only returns any generated error messages. If the server does not return any error messages, the requirements are met and the actions can succeed.

Debugging a lifecycle

After you have created a lifecycle, you might want to move a document through the lifecycle to test the various entry criteria and actions. To log the operations that occur during the tests, you can set a debug flag.

Java-based lifecycles

For Java-based lifecycles, add the following argument to the `method_verb` attribute of the method object associated with the `dm_bp_transition_java` method:

```
-debug T
```

This argument can also be added to the `method_verb` attribute value for the `dm_bp_validate_java` and `dm_bp_batch_java` methods.

Use Documentum Administrator to set the method verb.

Note: You can also debug Java lifecycle programs by turning on the tracing facility in DFC. For instructions about starting tracing in DFC, refer to the DFC documentation.

Docbasic-based lifecycles

To log the operations performed by the `dm_bp_transition.ebs` script, you set a Boolean variable, named `debug`, to true in the `dm_bp_transition` method.

The `dm_bp_transition.ebs` method is found in `%DM_HOME%\bin ($DM_HOME/bin)`.

Log file location

If debug is true, the method generates a log file in
%DOCUMENTUM%\dba\log\repository_name\bp
(\$DOCUMENTUM/dba/log/repository_name/bp).

Adding state extensions

You must add state extensions manually. You cannot add state extensions to a lifecycle using the Lifecycle Editor. Any user can create state extension objects for a lifecycle. However, to add the extension type to the lifecycle definition, a user must have at least Version permission for the lifecycle or be the owner of the lifecycle.

To add state extensions to a lifecycle:

1. Create a subtype of the dm_state_extension type for the lifecycle.
Use the Documentum Application Builder to define a subtype of dm_state_extension. Use the information in [State extensions, page 276](#), to identify the attributes needed in the subtype.
2. Create the extension objects for the states requiring extensions.
Refer to [Creating extension objects, page 294](#), for instructions.
3. Set the extension_type attribute in the lifecycle definition.
Use the API or DQL to set the extension_type attribute. You cannot use the Lifecycle Editor. Set the attribute to the name of the object type created in Step 1.

Creating extension objects

After you create a subtype of dm_state_extension for a lifecycle, you must create objects of that subtype for each lifecycle state that requires an extension. Use the API to create the objects.

Create the instances of the dm_state_extension subtype using the API. Create one extension object for each state requiring an extension.

You must set the following inherited attributes for each extension object:

- state_no
Set state_no to the state's number in the lifecycle.
- parent_id

- Set `parent_id` to the object ID of the `dm_policy` object representing the lifecycle's definition
- `child_id`
Set `child_id` to the object ID of the extension object. (The object ID is returned by the Create API.)
 - `relation_name`
Set `relation_name` to `BP_STATE_EXTENSION`.
 - `permanent_link`
`Permanent_link` determines whether the relationship is maintained when the parent object is versioned. Set it to `T (TRUE)` to retain the relationship between the extension and the state when the lifecycle definition is versioned. Set it to `F (FALSE)` if you do not want to maintain the relationship when the lifecycle is versioned.
- Additionally, set the attributes defined for the subtype to the information needed by the application for that state in the lifecycle.

Implementing a custom validation program

This section outlines the basic procedure for creating and installing a user-defined lifecycle validation method. Documentum provides standard technical support for the default validation method installed with the Content Server software. For assistance in creating, implementing, or debugging a user-defined validation method, contact Documentum Professional Services or Documentum Developer Support.

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.

To add a custom validation program to a lifecycle:

1. Create the custom validation program:
 - If your lifecycle is a Java-based lifecycle, you must write a Java program for validation. [Java validation program requirements, page 296](#), describes the requirements for a Java program.
 - If your lifecycle is a Docbasic-based lifecycle, you must write a Docbasic validation program. [Docbasic validation program requirements, page 296](#), describes the requirements for a Docbasic program.
2. Identify the custom validation program in the lifecycle definition.
Refer to [Adding the custom program to the lifecycle definition, page 296](#), for instructions.

Java validation program requirements

A Java program used to validate a lifecycle definition must implement the following interface:

```
public interface IDfLifecycleValidate
{
    public boolean userValidate(IDfSysObject policyObj,
                               String userName)
        throws DfException;
}
```

A return value of false or an exception indicates that the lifecycle definition did not pass the validation.

Docbasic validation program requirements

A custom Docbasic validation program must be a script that has the following function:

```
Public Function ValidationProc
    (ByVal SessionId As String,
     ByVal PolicyId As String,
     ByVal UserName As String,
     ByRef ErrorStack As String) As Boolean
...
    ValidationProc=True
End Function
```

SessionId is the current session string.

PolicyId is the object ID of the lifecycle.

UserName is the value in the user's user_os_name attribute.

ErrorStack is a user-defined error message.

The program is stored in the repository as a dm_procedure object. Procedure objects are subtypes of dm_sysobject. Consequently, you can version these objects.

Adding the custom program to the lifecycle definition

Use Lifecycle Editor to identify a lifecycle's custom validation program. For Java programs, you identify the path to the jar file. For Docbasic programs, you identify the procedure's name and version label. If you need help, consult the Editor's online help or the EMC Documentum Application Builder documentation.

Modifying lifecycles

You can change a lifecycle by adding states, deleting states, and rearranging the order of the states. You can also change the entry criteria, the actions on entry, and the post-entry actions.

Before you make changes to the state definitions or change the object types to which the lifecycle can be attached, you must uninstall the lifecycle. Uninstalling the lifecycle moves the policy object back to the validated state and suspends the lifecycle for all objects attached to it. You must have Write permission on the policy object to uninstall the lifecycle.

When you save your lifecycle definition changes, Content Server automatically moves the policy object back to the draft state. You must validate the lifecycle again and reinstall it to resume the lifecycle for the attached objects.

Changes that do not affect the lifecycle definition, such as changing the policy object's owner, do not affect the state of the policy object.

Possible changes

You can make many sorts of changes to a lifecycle definition, including:

- Adding one or more states at the end of the lifecycle
- Inserting states between existing states
- Deleting states or replacing them with a new state

Note: By default, you cannot remove a state from a lifecycle definition if there are objects attached to the lifecycle. You can override the constraint in the Lifecycle Editor or by setting the `force_flag` argument in the `removeState` method to `TRUE`. No notification is sent to the owners of the objects attached to the lifecycle when you use the `force_flag` argument.

- Rearranging existing states
- Adding, modifying, or deleting entry criteria, actions on entry, and post-entry actions

Use the Lifecycle Editor to make any changes to a lifecycle's definition.

Uninstall and installation notifications

When a lifecycle is uninstalled, all objects attached to that lifecycle are held in their current states until the lifecycle is re-installed and active again. If you are uninstalling a lifecycle to make changes, Lifecycle Editor allows you to send notifications to the

owners of objects attached to the lifecycle. Similarly, you can send a notification when you re-install the lifecycle definition.

Getting information about lifecycles

The repository includes two types of information about lifecycles:

- Information within policy objects about the lifecycle object itself
- Information within objects about an attached lifecycle

Lifecycle information

The following computed attributes for a policy object contain information about the policy itself:

- `_next_state` contains the symbolic name of the next normal state. Its value is NULL for the terminal state.
- `_previous_state` contains the symbolic name of the previous normal state. Its value is NULL for the base state.
- `_state_type` identifies the type of state. Valid values are:
 - -1, for an exception state
 - 0, for the base state
 - 1, for the terminal state
 - 2, for an intermediate state
- `_included_types` lists all acceptable object types for the policy
- `_alias_sets` lists the alias set at the specified index position

Use a Get method to retrieve the value of a computed attribute.

Object information

Computed attributes contain information about an object's current state in a lifecycle, and about the object's policy-related permissions.

Three computed attributes of every SysObject contain information about an object's current state in a lifecycle.

- `_policy_name` contains the name of the attached lifecycle
- `_current_state` contains the name of the current state

- `_resume_state` contains the name of the previous normal state

Three parallel attributes describe an object's current state in a lifecycle:

- The `r_policy_id` attribute contains the object ID of the policy object representing the lifecycle attached to the object
- The `r_current_state` attribute contains the name of the object's current state in the lifecycle.
- The `r_resume_state` attribute contains the name of the previous normal state if the current state is an exception state.

In addition, there are computed attributes that record the user's basic and extended permissions. These attributes may be useful when checking permissions for a lifecycle operation. Those attributes are included in [Table 1-1, page 17](#), of the *EMC Documentum Object Reference Manual*.

Deleting a lifecycle

To remove a lifecycle, use a Destroy method. You must have Delete permission on the lifecycle's policy object. You cannot destroy an installed lifecycle. Nor can you destroy a lifecycle that is defined as the default lifecycle for an object type.

By default, the Destroy method fails if any objects are attached to the lifecycle. A superuser can use the `force_flag` argument to destroy a lifecycle with attached objects. The server does not send notifications to the owners of the attached objects if the `force_flag` is used. When users attempt to change the state of an object attached to a lifecycle that has been destroyed, they receive an error message.

Destroying a policy object also destroys the expression objects identified in the `entry_criteria_id`, `action_object_id`, and `type_override_id` attributes.

The objects identified in the `user_criteria_id` and `user_action_id` attributes are not destroyed.

Tasks, Events, and Inboxes

This chapter describes tasks, events, and inboxes, supporting features of Process Management Services. The topics in this chapter are:

- [Introducing tasks and events, page 301](#), is a brief description of tasks and events.
- [Introducing Inboxes, page 302](#), describes repository inboxes.
- [dmi_queue_item objects, page 303](#), describes the objects that store inbox content in a repository.
- [Determining Inbox content, page 303](#), describes how to determine what is in an inbox.
- [Manual queuing and dequeuing , page 304](#), describes how to put an object or event notification in an inbox manually.
- [Signing off tasks, page 305](#), describes how to require users to electronically sign off a task.
- [Registering and unregistering for event notifications, page 306](#), describes how to request event notifications and how to remove requests for event notifications.
- [Querying for registration information, page 307](#), describes how to obtain information about the events for which a user is registered.

Introducing tasks and events

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, Content 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 also. Content Server supports a large number of system-defined events, representing operations such as checkins, promotions, and demotions. (Refer to [Audit, page 115](#), of the *Content Server API Reference Manual* for tables listing all system-defined events.)

Users can register to receive notifications of system-defined events. When a system-defined event occurs, Content Server sends an event notification automatically to any user who is registered to receive the event.

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 wish 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 Content Server to send a message with the event notification.

Users cannot register for application-defined events. Generating application-defined events and triggering notifications of the events are managed completely by the application.

Typically, users access tasks and event notifications through their repository inboxes. Inboxes are the electronic equivalent of the physical inboxes that sit on many people's desks. [Introducing Inboxes, page 302](#), describes inboxes.

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. [dmi_queue_item objects, page 303](#), describes queue items. For information about work items, refer to [Work item objects, page 229](#).

Introducing Inboxes

On your desk, a physical inbox holds various items that require your attention. Similarly, in the Documentum system, you have an electronic inbox that holds items that require your attention.

What an Inbox contains

Inboxes contain workflow tasks, event notifications, and 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 co-worker might ask you to review a presentation.

Accessing an Inbox

Users access their inboxes through the Documentum 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 are not defining home repositories for users, then Content 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 referencing `dmi_queue_item` objects.

dmi_queue_item objects

All items that appear in an inbox are managed by the server as objects of type `dmi_queue_item`. The attributes of a queue item object contain information about the queued item. For example, the `sent_by` attribute contains the name of the user who sent the item and the `date_sent` attribute tells when it was sent. (For a complete list of the `dmi_queue_item` attributes, refer to [Queue Item](#), page 352, in the *EMC Documentum Object Reference Manual*.)

`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 attributes that are set when an item is removed from an inbox are particularly helpful when examining the history of a project with which tasks are associated. These attributes are:

- `dequeued_by`
`dequeued_by` contains the name of the user that removed the item from the inbox.
- `dequeued_date`
`dequeued_date` contains the date and time that the item was removed.

Determining Inbox content

There are a variety of ways to obtain the content of a particular inbox. You can use a `GET_INBOX` administration method or a `Getevents` method or you can query the `dm_queue` view. The `GET_INBOX` method returns all items in a user's inbox. A `Getevents` methods returns all items placed on the queue since the last execution of a `Getevents`. The `dm_queue` view is a view of all queue item objects.

To determine whether to refresh an inbox, you can use an `Anyevents` method to check for new items. A new item is defined as any item queued to the inbox since 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.

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. Refer to [GET_INBOX, page 219](#), in the *Content Server DQL Reference Manual* for instructions on using this method.

Getevents method

A 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. (Refer to [Queue Item, page 352](#), of the *EMC Documentum Object Reference Manual* for information about the attributes of a queue item object.)

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'
```

Manual queuing and dequeuing

Most inbox items are generated automatically by workflows or an event registration. However, you can manually queue an object or a workflow-related event notification in an application using the Queue method. You can also manually take an item out of an inbox. This is called dequeuing the item.

Queuing items

Use a Queue method to place an item in an inbox. Executing Queue creates a queue item object. You can queue an object 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. Content 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 attribute 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, perhaps the application reads the priorities and presents the items to the user in priority order. The priority is ignored by Content Server.

Optional arguments allow you to include a message to the user receiving the item.

Queue methods return a stamp value that represents the queued item's position in the inbox. To remove the item from an inbox (dequeue the item), you must know this value.

Dequeuing an Inbox item

Use a Dequeue method to remove an item placed in an inbox using a Queue method. You must provide the stamp value returned by the Queue method as an argument to Dequeue. You can obtain this value using the Getevents method.

Executing a Dequeue method sets two queue item attributes:

- `dequeued_by`
This attribute contains the name of the user who dequeued the item.
- `dequeued_date`
This attribute contains the date and time that the item was dequeued.

Signing off tasks

You can write an application that requires users to electronically sign off a task before it is removed from the user's queue. This feature is implemented using a Signoff method. A Signoff method requires a user name and password as arguments.

For example, an application can provide a dialog that asks the user for a password before allowing a user to remove a task from an inbox. Then, the application can use the answer and the user name of the current logged in user to execute the Signoff method.

Executing Signoff creates an audit trail entry recording the sign off.

Registering and unregistering for event notifications

An event notification is a notice from Content Server that a particular system event has occurred. To receive an event notification, 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 may want to know whenever a particular document is checked out. Or perhaps you want to know when any document is checked out. Maybe you want to know when any action (checkin, checkout, promotion, and so forth) happens to a particular document. Each of these actions is an event, and you can register to receive notification when they occur. After you have registered for an event, the server continues to notify you when the event occurs until you remove the registration.

Registering for events

You can register to receive events using Documentum Administrator. You can also use a Register method. Events that you can register for are listed in [Appendix B, System Events](#), of the *Content Server API Reference Manual*.

Although you must assign a priority value to an event when you use a Register 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 Content Server.

Optional arguments allow you to include a message to the user receiving the item.

You cannot register another user for an event. Executing a Register method registers the current user for the specified event.

Removing a registration

To remove an event registration, use Documentum Administrator or an Unregister method.

Only a user with Sysadmin or Superuser privileges can remove another user's 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.

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'
```


Aliases

This appendix describes how aliases are implemented and used. Aliases support Content Server's process management services. The appendix includes the following topics:

- [Introducing aliases, page 309](#)
- [Internal implementation, page 310](#)
- [Defining aliases, page 310](#)
- [Alias scopes, page 311](#)
- [Resolving aliases in SysObjects, page 313](#)
- [Resolving aliases in template ACLs, page 314](#)
- [Resolving aliases in Link and Unlink methods, page 314](#)
- [Resolving aliases in workflows, page 315](#)

Introducing aliases

Aliases are place holders 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` attributes
- In ACL template objects, in the `r_accessor_name` attribute
- In workflow activity definitions (`dm_activity` objects), in the `performer_name` attribute
- In a Link or Unlink method, in the folder path argument

Using aliases lets you 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's performer, or the user permissions in a document's ACL is no longer hard coded into the application. Instead, aliases are placeholders for these values. The aliases are resolved to real user names or 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's 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's 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.

Internal implementation

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 attributes alias_name and alias_value. The values at each index position represent one alias and the corresponding real user or group name or folder path. (For information about creating, modifying, or deleting alias sets, refer to [Alias sets, page 72](#), of the *Content Server Administrator's Guide*.)

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.

Defining aliases

When you define an alias in place of a user or group name or a 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 attribute of the alias set object.

To put an alias in a SysObject or activity definition, use the Set method. To put an alias in a template ACL, use the Grant method. To include an alias in the 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. The following Set method sets a document's owner_name attribute to the engr_vp alias:

```
dmAPISet ("set,s0,09000001801e754c,owner_name",
```

```
"%engr_aliases.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:

```
dmAPISet("set,s0,09000001801e754c,owner_name","%engr_vp")
```

In such cases, the server uses a pre-defined algorithm to search one or more alias scopes to resolve the alias name.

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 pre-determined, ordered series of scopes for an alias name matching the alias name in the specification. Which scopes are searched depends on where the alias is found.

Workflow alias scopes

To resolve an alias in an activity definition that doesn't 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's performer
- Server configuration

Within the workflow scope, the server searches in the alias set defined in the workflow object's `r_alias_set_id` attribute. This attribute is set when the workflow is instantiated. The server copies the alias set specified in the `perf_alias_set_id` attribute of the workflow's definition (process object) and sets the `r_alias_set_id` attribute 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 config's `alias_set` attribute.

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's alias set is defined in the `alias_set_id` attribute of the user's 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's alias set is identified in the `alias_set_id` attribute.

Within the server configuration scope, the search is conducted in the alias set defined in the `alias_set_id` attribute of the server config object.

Non-workflow alias scopes

Aliases used in non-workflow contexts have the following possible scopes:

- Lifecycle
- Session
- User
- Group
- Server config

When the server searches within the lifecycle scope, it searches in the alias set defined in the SysObject's `r_alias_set_id` attribute. This attribute is set when the object is attached to a lifecycle. (Refer to [Determining the lifecycle scope for SysObjects](#), page 313, for details.)

Within the session scope, the server searches the alias set object defined in the session config's `alias_set` attribute.

Within the user's scope, the search is in the alias set object defined in the `alias_set_id` attribute of the user's user object. The user is the user who initiated the action that caused the alias resolution to occur. For example, suppose 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's default group.

Within the system scope, the search is in the alias set object defined in the `alias_set_id` attribute of the server config object.

Determining the lifecycle scope for SysObjects

A SysObject's 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` attribute, you can either choose one from the list as the object's 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's `alias_set_ids` attribute.
- 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's alias set isn't found, the server uses the alias set defined for the user's default group if that alias set is in the `alias_set_ids` list.
- If the default group scope's alias set isn't 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` attribute.

If the policy object has no defined alias set objects in the `alias_set_ids` attribute, the SysObject's `r_alias_set_id` attribute is not set.

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's `r_alias_set_id` attribute. 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's scope, the server looks at the alias set defined for the user scope.
- The user scope is the alias set defined in the `alias_set_id` attribute of the `dm_user` object for the current user.
- If the alias is not found in the user's scope, the server looks at the alias set defined for the user's default group scope.
- If the alias is not found in the user's default group scope, the server looks at the alias set defined for the system scope.

If the server doesn't find a match in any of the scopes, it returns an error.

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` attribute of the object. This alias set is the object's lifecycle scope. If no match is found, the server returns an error.
- If the object to which the template is applied doesn't 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` attribute 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` attribute 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's 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, then Content Server returns an error stating that an error set was not found for the current user.

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 algorithm the server uses to resolve the alias name is the same as that used for resolving aliases in SysObjects (described in [Resolving aliases in SysObjects](#), page 313).

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's alias set. (Refer to [Resolving aliases during workflow startup, page 315](#), below, for details.)

Resolving an alias when an activity starts is done automatically by the server. [Resolving aliases during activity startup, page 316](#), describes the algorithm used by the server to resolve aliases at runtime.

Resolving aliases during workflow startup

A workflow definition can include an alias set to be used to resolve aliases found in the workflow's 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 very flexible workflow template. It allows the workflow's 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's `r_alias_set_id` attribute to the copy's object ID.

If the workflow is started through the Workflow Manager (WFM), the WFM 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's 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's 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.

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

The default resolution algorithm

The server uses the default resolution algorithm when the activity's `resolve_type` attribute 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's performer
- Server configuration

The server examines the alias set defined in each scope until a match for the alias name is found.

The package resolution algorithm

The server uses the package resolution algorithm if the activity's `resolve_type` attribute is set to 1. The algorithm searches only the package or packages associated with the activity's incoming ports. Which packages are searched depends on the setting of the activity's `resolve_pkg_name` attribute.

If the `resolve_pkg_name` attribute is set to the name of a package, the server searches the alias sets of the package's components. The search is conducted in the order in which the components are stored in the package.

If the `resolve_pkg_name` attribute 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.

The user resolution algorithm

The server uses the user resolution algorithm if the activity's `resolve_type` attribute 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's default group.

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's `performer_type` attribute. 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.

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's supervisor
- Assigns the work item to the supervisor

Writing Distributed Applications

This appendix contains information and guidelines for writing applications to run in a multi-repository distributed configuration.

The client library (DMCL) provided with Content Server supports applications that allow users to access objects in multiple repositories in one session. To ensure consistent security across the repositories, the users, groups, and external ACLs in the repositories must match. Use a federation to automate the task of keeping the users, groups, and external ACLs matching in all repositories. For information about federations, refer to [Manipulating a federation, page 120](#) of the *Distributed Configuration Guide*.

To ensure that all users see the same attribute information for objects, the object type definitions should be the same in each repository also.

This appendix contains the following topics:

- [Client Library \(DMCL\) support for distributed applications, page 319](#), which describes the features supported by the DMCL that support applications running against multiple repositories.
- [Security, page 323](#), which describes how security is implemented for reference links and replicas.
- [Distributed messaging, page 324](#), which describes Content Server's distributed messaging services.
- [Information for client applications, page 325](#), which contains guidelines and information to help you write applications against multiple repositories.
- [Reference link refreshes, page 329](#), which describes how reference links are refreshed.
- [The callback attributes, page 330](#), which describes a set of attributes you can use in applications to implement connection callback functions.

Client Library (DMCL) support for distributed applications

This section describes how the DMCL supports applications that access multiple repositories.

Subconnections

After an application starts a repository session, the application can access data and perform operations on objects in different repositories without executing Connect methods to open sessions with the repositories. When an application references an object in a repository to which there is no established connection, the DMCL automatically begins a session with the repository. The DMCL uses the authentication information of the current user to establish the connection.

The maximum number of connections in a single repository session is defined by the value in the `max_connection_per_session` attribute of the `api` config object. If the limit is reached and the DMCL requires another connection, it closes the least recently used connection to allow it to open the new connection.

The DMCL maintains a separate object and type cache for each connection in a session.

Method scoping

A method is executed within a particular repository scope. That is, the operations performed by the method affect the objects in one repository. When an application accesses multiple repositories in one session, it is important that each method call execute against the correct repository.

The DMCL identifies the repository scope for each method call and opens a connection to that repository or switches to that repository connection if one is already open. This context switching is done automatically.

To identify the repository scope, the DMCL uses:

- A *scoped argument* in the method call if a scoped argument is present
- A *subconnection identifier* in the method call
- The repository identified in the `docbase_scope` attribute of the session config object

A scoped argument is an argument that identifies a repository. For example, the Checkout method has an argument that identifies the document to be checked out by its object ID. The object ID is a scoped argument. It identifies the repository that contains the document.

A subconnection identifier is a session identifier that identifies the primary session and the subconnection. The format for a subconnection identifier is `Sncn`; for example, `S0c1` or `S0c2`. A subconnection identifier is obtained using the `Getconnection` method. (For more information, refer to [Primary sessions and subconnections](#), page 31.)

If a method call does not have a scoped argument and does not include a subconnection identifier, the repository scope is the repository identified in the `docbase_scope` attribute of the session config object.

For more information about repository scope, refer to [Defining the repository scope](#), page 43.

Reference links

A reference link is a pointer in one repository to an object in a different repository. The DMCL automatically creates reference links when an application performs the following operations:

- Links an object in a remote repository to a local folder or cabinet
- Checks out a remote object
- Adds a remote object to a local virtual document

A reference link consists of two objects: a mirror object and a dm_reference object. The mirror object is an object in the repository that mirrors the attribute values of the source object in the remote repository. The dm_reference object is the internal link between the mirror object in the local repository and the source object in the remote repository.

Valid object types for reference links

The DMCL supports reference links only for objects of the following types:

- cabinet
- docbase config
- document
- folder
- note
- procedure
- query
- router
- script
- server config
- smart list
- SysObject

The DMCL does not support creating reference links for other object types.

Reference link binding

By default, the operation that creates the reference link also defines which version of the source object is bound to the reference link. For example, when users check out a document, they identify which version they want. If they are checking out a remote document, the specified version is bound to the reference link.

If the version is not identified as part of the operation, the server automatically binds the CURRENT version of the source object to the reference link.

You can change the default binding. Refer to [Defining a binding label for reference links](#), page 326 for instructions.

Reference link storage

When a reference link is created by a linking operation, the mirror object is stored in the requested location. For example, if a user links a document from repository B to FolderA in repository A, the mirror object is stored in FolderA.

When a reference link is created by other operations, such as checkout or assemble, the mirror objects are stored in folders in the local repository under /System/Distributed References. For example, mirror objects created when users check out remote objects are stored in /System/Distributed References/Checkout.

Type-specific behavior of reference links

The DMCL does not allow users to link local objects to a cabinet or folder that is a reference link. For example, suppose a folder from the Marketing repository is linked to a cabinet in the Engineering repository. Users in the Engineering repository cannot add objects to the folder.

If users execute a script or procedure that is a reference link, the DMCL fetches the script or procedure from the remote repository and executes it against the local repository session. The script or procedure is not executed against the remote repository.

Indirect references

An indirect reference is an object ID in the format *@mirror_object_id*. When an indirect reference appears in a method, the DMCL dereferences the mirror object ID and performs the operation on the source object.

Replicas

Replicas are objects that are created by object replication jobs. Object replication jobs copy objects from one repository to another. The copy created in the target repository is called

a replica. The original is called the *source object*. A replica includes both the attribute values and the content of the source object.

Applications can access the source objects by referencing the replica objects in method calls. For example, suppose that DocA in repository A is replicated into repository B. An application that starts a session with repository B can check out DocA by executing the method call against the replica in repository B. The DMCL recognizes that DocA in repository B is a replica and opens a connection (if necessary) to repository A and checks out the source DocA.

It is not necessary to use an indirect reference when you include a replica object ID in a method call.

Security

This section describes how security is handled for reference links and replicas.

Reference links

Security on the source object is controlled by Content Server for the source repository whether the update is made directly to the source or through a reference link.

Security on the mirror objects is controlled by Content Server for the repository that contains the mirror object. The ACL applied to the mirror object is derived from source object's ACL using the following rules:

- If the source object's ACL is a private ACL, the server creates a copy of the ACL and assigns the copy to the mirror object.
- If the source object's ACL is a system ACL and there is a system ACL in the local repository with the same name, the server assigns the local system ACL to the mirror object.

In this situation, the access control entries in the system ACLs are not required to match. Only the ACL names are matched. It is possible in these cases for users to have different permissions on the source object and the reference link.

- If the source object's ACL is a system ACL and there is no system ACL with a matching name in the local repository, the server creates a local copy of the ACL and assigns the copy to the mirror object. The copy is a private ACL.

Replicas

Replica security is handled by the local Content Server. Each replica is assigned an ACL when the replica is created by the object replication job. The job's replication mode determines how the ACL is selected. For details about replica security, refer to [Replication modes, page 46](#) in the *Distributed Configuration Guide*.

Distributed messaging

Distributed messaging is a feature of Content Server that supports multi-repository configurations. The server copies events from one repository to another to facilitate distributed workflows and distributed event notification.

Distributed workflow occurs when an application or user assigns an activity in a workflow definition in a repository to a user in another repository. When the activity is started, the server creates a distributed task (queue item) in the local repository that is copied to the user's home repository. It appears in the user's home repository inbox.

Distributed event notification occurs when users register for event notifications in a repository that is not their home repository. When the event occurs, the server in the repository that contains the registration creates a distributed event notification (queue item) that is copied to the user's home repository. The event notification appears in the user's home repository inbox.

Distributed messaging is implemented using five attributes in the `dmi_queue_item` object type. [Table B-1, page 324](#) briefly describes these attributes.

Table B-1. Attributes implementing distributed messaging in `dmi_queue_item`

Attribute	Description
<code>source_docbase</code>	The repository in which the event or task originates.
<code>target_docbase</code>	The home repository of the user receiving the event or task.
<code>remote_pending</code>	TRUE if the <code>dmi_queue_item</code> must be copied to the target repository. FALSE if it has already been copied or is not a distributed queue item.

Attribute	Description
source_event_id	Object ID of the source dmi_queue_item object that generated this queue item in the target repository. This attribute is only set in the target repository.
source_event_stamp	The i_vstamp value of the source dmi_queue_item object that generated this queue item in the target repository. This attribute is only set in the target repository.

Information for client applications

This section contains specific information and guidelines you can use to help ensure that applications running against multiple repositories function correctly.

Query and method guidelines

- Queries that return the contents of a cabinet or folder must be directed against the repository in which the cabinet or folder resides. The query cannot identify the cabinet or folder using an indirect reference.

Refer to the example given in [Dereference, page 181](#) in the *Content Server API Reference Manual* for an example of how to do this using the API.

- Applications must use the mirror object ID to identify a reference link in a method call. The remote object cannot be updated or accessed by using the dm_reference object ID in a method call.
- DQL statements always operate on the mirror object. They do not affect the source object. You can use DQL to:
 - Query the mirror object attributes
 - Update local attributes of the mirror object
 - Delete the mirror object

The DESCEND option in a SELECT statement's IN DOCUMENT clause and in the FOLDER and CABINET predicates is not operative if the object being searched is a reference link.

Executing Smart Lists or stored queries

If the application executes a smart list or stored query, the query must be directed against the appropriate repository.

Executing scripts and procedures

The script or procedure executes in the context of the current repository scope.

Annotations

If the application creates annotations for a document, we recommend that the annotations be created in the same repository in which the document resides.

Object and type caches

If users are viewing attribute data stored with the mirror object, we recommend that object type definitions be the same across all repositories. If the application accesses the source object for all operations, including viewing attribute data, then the object type definitions can be different. The DMCL maintains separate object and type caches for each connection in a session.

Defining a binding label for reference links

The binding between a mirror object and its source object is typically set by default when the user executes the operation that creates the reference link. For example, suppose a user, who is logged in to repository A, checks out the approved version of DocumentX, which resides in repository B. The server creates a reference link in repository A to the approved version of DocumentX.

If the user's operation doesn't identify a specific version of the object, the server binds the CURRENT version to the reference link by default.

You can override the binding specified at runtime by setting the `reference_binding_label` attribute in the session config object. For example, suppose `reference_binding_label` is set to CURRENT before the user checks out the approved version of DocumentX. Even though the user requests the approved version, the server binds the CURRENT

version to the reference link. When the user opens the document, the server opens the CURRENT version. The setting in `reference_binding_label` overrides the binding label identified by the user.

If the version defined in `reference_binding_label` doesn't exist, the server returns an error.

To return to the default behavior, set the `reference_binding_label` attribute to a blank string.

Operations on reference links

There are only a few operations that you can execute against the reference link. You can:

- View the attributes of the mirror object
- Link and unlink the reference link
- Update local attributes
- Delete the reference link

To perform an operation on the reference link, use only the mirror object ID in the method call. For example:

```
dmAPIGet("get,session,mirror_object_id,title")
```

You can delete a reference link if you meet one of the following conditions:

- You have Delete permission to the mirror object
- You performed the operation that created the reference link
- You have Superuser privileges in the repository that contains the reference link

Operations on reference link source objects

Use an indirect reference in method calls to direct the operation to the source object of a reference link. For example, the following method sets the subject attribute of the source document:

```
dmAPISet("set,session,@mirror_object_id,subject",  
"Marketing Proposal")
```

Operations on replicas

Any updates you make to a replica's global attributes after you fetch or check it out are applied to the source object and a synchronous refresh action updates the replica. Checking out a replica checks out the source object.

Methods that update local attributes affect only the replica.

Methods that do not update an object operate on the replica. For example, the following method returns the value of the title attribute in the replica:

```
dmAPIGet("get,c,replica_doc_id,title")
```

Retrieving content

For reference links, the `Getfile` and `Getcontent` methods always operate on the source object whether you identify the source object or an indirect reference in the method call. For example, suppose `DocumentA` from repository A is linked to a folder in repository B. Users in repository B can use either of the following method calls to retrieve the content file for `DocumentA`:

```
dmAPIGet("getfile,s0,mirror_object_DocumentA")
```

```
dmAPIGet("getfile,s0,@mirror_object_DocumentA")
```

When applied to a replica, the methods retrieve the content file associated with the replica.

Distributed operations

The following methods can operate across repositories:

- **Link and Unlink**
You can link an object in a repository to a folder or cabinet in another repository. Similarly, you can unlink a remote object from a local folder or cabinet.
- **Appendpart, Insertpart, and Removepart**
You can append or insert a remote object into a virtual document in local repository. You can also remove a remote component from a local virtual document.
- **Addnote**
You can add a remote note (annotation) to a document in the local repository. (This is not the recommended best practice. We recommend that you store annotations in the same repository as the document to which they are attached.)
- **Saveasnew**
You can copy a document from one repository to another.
- **Addpackage**
You can add a remote document as a package in a local workflow.

Explicit transactions

The DMCL does not support multi-repository explicit transactions. You cannot make any method call that updates a remote object inside an explicit transaction.

Reference link refreshes

Mirror objects are refreshed when:

- The `dm_DistOperations` job runs
- The source object is accessed through the reference link by a method
- A Refresh method is issued

The `dm_DistOperations` job

The `dm_DistOperations` job is installed with Content Server as one of the system administration tools. Like the other administration jobs, it runs on a defined schedule.

When the job runs, it finds all the reference links that are ready for refreshes and invokes a Refresh method for each.

For details about the `dm_DistOperations` job, refer to [The `dm_DistOperations` job](#), page 132 in the *Distributed Configuration Guide*.

Method access

A mirror object is checked to determine if a refresh is needed when a method call includes an indirect reference to the object's source. The DMCL compares the `i_vstamp` value of the mirror object and the source object. If they are not the same, a refresh of the mirror object is necessary.

If the method call does not update a source object, the DMCL queues a refresh request for the mirror object. The next execution of the `dm_DistOperations` job in the mirror object's repository refreshes the mirror object.

If the method call updates the source object, the mirror object is refreshed immediately, when the update operation is completed.

The Refresh method

The Refresh method allows users and applications to request a refresh for a mirror object. Using the Refresh method requires Browse permission on the source object. The syntax is:

```
dmAPIExec("refresh,session,mirror_object_id")
```

The callback attributes

There are six repeating attributes in the session config object that allow you to provide callback functionality in applications. These attributes are:

- `new_connection_callback` and `new_connection_data`
- `connect_success_callback` and `connect_success_data`
- `connect_failure_callback` and `connect_failure_data`

The attributes have an integer datatype. The values in the attributes are memory addresses for the functions you define and the data to be passed to the functions.

The `dmapp.h` header file provided with Content Server contains the definitions of the callback functions. The definitions are found in a section called Connection Callbacks. The functions you write must conform to those definitions, and your application must include the `dmapp.h` header file.

If you define more than one function for a particular callback use, the server executes the functions in the order in which they are identified in the attribute. For example, if you write two Connect Success callback functions, their memory addresses appear in `connect_success_callback[0]` and `connect_success_callback[1]`. The server will execute the function identified by the [0] index position first, then the function identified by the [1] index position.

The server passes a function identified at a particular index position the data provided at the memory address in the corresponding index position in the `_data` attribute. For example, the server passes the data at the memory address specified in `connect_success_data[0]` to the function defined at the memory address in `connect_success_callback[0]`.

To enable the callback functionality, the `connect_callback_enabled` attribute in the session config must be set to `TRUE`.

The new connection callback attributes

A new connection callback function allows an application to track when the DMCL establishes new subconnections to Content Servers. The new connection callback functions are executed before the attempt to log in to the server.

New connection callback functions are not executed when the primary session is started.

The connect success callback attributes

A connect success callback function allows an application to track successful new subconnections. Use a connect success callback to clear any status messages generated by new connection callback functions.

The connect success callback functions are executed after the attempt to log in to the server.

Connect success callback functions are not executed when the primary session is started.

The connect failure callback attributes

A connect failure callback function allows an application to capture login failures for subconnection attempts and prompt the user for authentication information.

Connect failure callback functions are called when a login failure occurs when the DMCL attempts to establish a new subconnection. After the callback function completes, the DMCL attempts to establish the connection again.

Connect failure callback functions are not executed if the log in to the primary session fails.

% (percent sign) in alias specification, 310

A

a_bpaction_run_as attribute, 274

a_contain_desc attribute, 165

a_contain_type attribute, 165

a_content_type attribute, 142

a_controlling_app attribute, 86

a_full_text attribute, 134, 142

a_is_signed attribute, 102

a_storage_type attribute, 137

AAC tokens, *see* application access control tokens

Abort method, 256

absolute linking, 166

ACLs

 custom, creating, 148

 default, assigning, 147

 described, 91

 entries, described, 92

 grantPermit method, 149

 kinds of, 92

 non-default, assigning, 148

 object-level permissions, 88

 revokePermit method, 149

 templates

 alias use, 309

 aliases, resolving, 314

 described, 93

 Trusted Content Services and, 148, 150

ACLs

 replacing, 151

acquired state, for work items, 238

actions on entry (lifecycle)

 aliases in, 291

 defining, 285

 described, 266

 Docbasic programs, 286

 execution order, 285

 Java programs, 286

active state (activity instance), 236

activities

 automatic, 203

 Begin, 201

 described, 201

 End, 201

 links between, 202

 manual, 203

 manual transitions

 output choice behavior, 220

 output choices, limiting, 220

 number of work items to complete, defining, 219

 performer categories, 206

 repeatable, 205

 starting conditions, 215

 Step, 201

 suspend timers, 225

 task subjects, 213

 timer implementations, 251

 transition behavior, 219

 transition types, 219

 trigger condition, 215

 trigger events, 215

 validation checks, 227

 warning timers, 224

activity definitions

 automatic transitions, 220

 control_flag attribute, 204

 delegation, 204

 described, 202

 destroying, 260

 extension characteristic, 205

 installing, 228

 manual transitions, 220

 modifying, 259

 multiple use, 201, 205

 names of, 202

- package definitions
 - adding, 259
 - compatibility, 218
 - removing, 259
 - scope, 217
 - validation, 218
- performers
 - alias as performer, 212
 - defining, 205, 211
- ports
 - adding, 259
 - described, 215
 - input, 216
 - output, 216
 - removing, 259
 - requirements for, 216
 - revert, 216
- prescribed transitions, 219
- priority values, 203
- route cases, described, 221
- saving, 260
- starting condition, defining, 215
- states, 226
- task subjects, defining, 213
- transition_eval_cnt attribute, 219
- transition_flag attribute, 221
- transition_max_output_cnt attribute, 220
- trigger conditions, 215
- trigger events, 215
- uninstalling, effects of, 259
- validation, 226 to 227
- XML files as packages, 216
- XPath expressions and, 221
- activity instances
 - alias resolution
 - alias_category attribute, 317
 - default, 316
 - described, 244
 - errors, 317
 - package resolution algorithm, 316
 - user resolution algorithm, 317
 - completion, evaluating, 246
 - defined, 229
 - dm_bpm_transition method, 249
 - execution, 241, 244
 - halted workflows and, 255
 - halting, 255
- packages
 - acceptance protocol, 249
 - consolidation of, 242
 - restarting failed, 256
 - resuming
 - automatically, 237
 - paused activities, 256
 - sequence number, obtaining, 256
 - starting condition
 - described, 215
 - evaluating, 242
 - states of, 236
 - timer instantiation, 252
 - transition conditions, evaluating, 249
 - user time and cost reporting, 253
 - work items
 - completing, 231
 - pausing and resuming, 257
- addAttachment method, 233
- Addsignature method, auditing, 93
- Addsignature method
 - audit trail entries, 99
 - auditing execution, 93
 - description of actions, 97
 - permissions needed, 101
- Addnote method, 156
- addPackage method, 241
- Addpackageinfo method, 259
- Addport method, 259
- alias set object type, 310
- alias sets
 - alias set object type, 310
 - group alias sets, 312
 - lifecycle usage, 276
 - server configuration scope, 312
 - session alias set, 311
 - user alias sets, 312
- alias_category attribute, 317
- alias_name attribute, 310
- alias_set attribute, 311 to 312
- alias_set_id attribute, 312
- alias_set_ids attribute, 276
- _alias_sets (computed attribute), 298
- alias_value attribute, 310
- aliases
 - alias set object type, 310
 - apisession, 30
 - lifecycle scope, defining, 313
 - lifecycle state actions, use in, 276, 291
 - methods, resolving in, 314

- object types and, 309
 - purpose, 309
 - resolution
 - errors, 317
 - in activities, 244, 316
 - in lifecycles, 312
 - in workflows, 213, 311
 - non-workflow scopes, 312
 - scope, defined, 311
 - specification format, 212, 310
 - workflows, use in, 212
 - allow_attach attribute, 269, 279
 - allow_demote attribute, 281
 - allow_schedule attribute, 281
 - ALTER TYPE (statement), 74
 - annotations
 - Addnote method, 156
 - creating, 155
 - deleting from repository, 156
 - described, 154
 - detaching, 156
 - distributed environments, use in, 326
 - effect of operations on, 156
 - keeping across versions, 156
 - workflow package notes, 232
 - Anyevents method, 303
 - API (Application Program Interface)
 - data dictionary, querying, 72
 - session alias, 30
 - api config object, 30, 108
 - append methods, 142
 - Appendpart method, 163
 - application access control tokens
 - described, 39
 - dmtkgen utility, 42
 - expiration, 42
 - format, 41
 - generating, 42
 - login ticket key use, 41
 - methods and, 43
 - scope, 41
 - superuser privileges and, 40
 - use of, 40
 - application component classifiers, 70
 - application events, auditing, 94
 - application_code attribute, 86
 - applications, *see* client applications
 - ASCII use requirements, 111 to 112
 - assembling virtual documents, 173
 - assembly behavior, defining, 167, 172
 - assembly objects
 - modifying, 178
 - object type for, 163
 - attach method, 269
 - attachability, lifecycle states, 279
 - attachments, for workflows, 233
 - attributes
 - a_full_text, 134
 - constraints (data dictionary), 66, 69
 - data dictionary information,
 - retrieving, 71
 - default values, defining, 70
 - described, 58 to 59
 - global and local, 59
 - immutability and, 125
 - mapping information (data dictionary), 71
 - RDBMS tables for, 60
 - repeating, modifying, 142
 - single-valued, modifying, 142
 - value assistance (data dictionary), 70
 - values, setting, 132
 - Audit method
 - auditing execution of, 93
 - described, 94
 - auditing
 - Audit method, 94
 - default auditing, 93
 - described, 93
 - registry objects and, 94
 - automatic activities
 - described, 203
 - effects of Complete method, 231
 - executing, 244
 - execution queue size, 245
 - information passed to method, 245
 - mode parameter values, 246
 - performer categories for, 210
 - priority values, 203
 - resolving performers, 243
 - workflow agent, 203, 234
 - automatic activity transitions, 220
- ## B
- base permissions, 88
 - See also* object-level permissions
 - base state (lifecycle), returning to, 280
 - batch promotion, 270

- BATCH_PROMOTE (administration method), 270
 - Begin activities, 201
 - bindFile method, 145
 - binding, 166
 - See also* early binding; late binding
 - in lifecycles
 - actions on entry, 287
 - post-entry actions, 289
 - user criteria, 284
 - reference links, 321, 326
 - virtual document components, 166
 - Boolean expressions as entry criteria, 284
 - bp_actionproc.ebs, 286
 - BPM, *see* Business Process Manager (BPM)
 - Branch method, 121, 159
 - branching versions
 - defined, 121
 - implicit version labels and, 121
 - Business Process Manager (BPM), 200
- C**
- cabinets
 - destroying, 75
 - linking documents, 134
 - reference link behavior, 322
 - cache config objects, 50
 - cache..map file, 47
 - Cachequery method, 52
 - caches, 326
 - See also* consistency checking;
 - persistent client caches
 - data dictionary cache, 46, 51
 - object, 45
 - persistent client, 45
 - query, 46
 - type cache, 46, 51
 - callback attributes for connections, 330
 - Centera profiles, 130
 - change record object type, 52
 - CHANGE...OBJECT (statement), 77
 - check constraints, 69
 - CHECK_CACHE_CONFIG
 - administration method, 51
 - Checkin method, 146, 159
 - Checkinapp method, 146
 - Checkout method, 141
 - child, in foreign key, 68
 - child_id attribute, 158
 - child_label attribute, 158
 - classifiers, for application components, 70
 - client applications
 - a_controlling_app attribute, 86
 - aliases, use of, 309
 - annotations, managing, 326
 - application events, auditing, 94
 - application_code attribute, 86
 - cabinets and folders, querying, 325
 - Checkinapp method, 146
 - component classifiers, 70
 - connection brokers, runtime
 - specification, 33
 - connection callback attributes,
 - using, 330
 - connection pooling, 35
 - control of SysObjects, 86
 - digital signatures, using, 102
 - distributed operations, 328
 - dmapp.h file, 330
 - executing
 - scripts and procedures, 326
 - Smart Lists, 326
 - stored queries, 326
 - lifecycle state types, using, 277
 - lifecycles and, 276
 - lifecycles, testing, 293
 - locking strategies, 126
 - multi-repository transactions, 329
 - operations on reference links, 327
 - persistent client caches, 45, 48
 - referencing reference links, 325
 - replica operations, 327
 - repository scope, 43
 - repository sessions, 29
 - retrieving content files through
 - reference links, 328
 - roles
 - domain groups, 83
 - supporting groups, 83
 - subconnections, opening, 31
 - virtual document components,
 - obtaining path to, 181
 - XML support, 165
 - client library, *see* DMCL (client library)
 - client sessions, *see* repository sessions
 - client_check_interval attribute, 51
 - client_codepage attribute, 108
 - client_locale attribute, 109
 - client_pcaching_change attribute, 52

- code pages
 - ASCII support, 107
 - default_client_codepage attribute, 108
 - group name requirements, 111
 - server, 107
 - user name requirements, 111
- Collaboration Services license, 23
- collection objects, 58
- compatibility, workflow package, 218
- Complete method, 231
- component specifications (data dictionary), 70
- components (virtual document)
 - adding to snapshots, 177
 - adding to virtual documents, 171
 - assembly behavior, defining, 167
 - changing order, 172
 - deleting from snapshots, 178
 - removing, 172
 - selecting for snapshot, 173
- composite predicate objects, 222
- compound_integrity attribute, 164
- computed attributes
 - _is_restricted_session, 32
 - lifecycle specific, 284, 298
- concurrent users, 44
- cond expr objects, 222
- conditional assembly, 164
- Config Audit user privileges, 88
- connect failure callback attributes, 331
- Connect method, 31
- connect success callback attributes, 331
- connect_callback_enabled attribute, 330
- connection brokers
 - api config attributes for, 33
 - described, 33
 - specifying at runtime, 33
- connection callback attributes, 330
- connection config object, 30
- connection pooling, 34
- connections
 - application access control tokens, 39
 - connection config object, 30
 - login tickets, 35
 - new connection callback attributes, 331
- consistency checking
 - cache config object use, 50
 - Cachequery method, 52
 - CHECK_CACHE_CONFIG
 - administration method, 51
 - client_check_interval attribute, 51
 - consistency check rules
 - default rule, 52
 - defined, 48
 - described, 49
 - DMCL behavior, 51
 - query results, 51
 - r_last_changed_date attribute, 51
 - type and data dictionary caches, 51
- constraints (data dictionary)
 - check, 69
 - defined, 66
 - foreign key, 68
 - not null, 69
 - primary key, 67
 - unique key, 67
- constraints on explicit transactions, 53
- containment objects
 - copy_child attribute, 169
 - follow_assembly attribute, 169
 - object type for, 162
 - Updatepart method, 172
 - use_node_ver_label attribute, 167
- content assignment policies
 - overriding, 137
- content assignment policies, 137
- content files
 - adding, 135
 - assigning to storage area, 136
 - bindFile method, 145
 - content assignment policies, 137
 - default storage algorithm, 137
 - digital shredding, 104
 - internationalization, 106
 - object types accepting, 59
 - page numbers, 117, 145
 - removing from documents, 76, 145
 - renditions and, 183
 - replacing, 145
 - retrieving through reference links, 328
 - sharing, 145
 - storage options, 63
 - virtual documents and, 165
- content objects
 - described, 116
 - transformation path and, 188
- Content Server
 - communicating with, 27

- compound_integrity attribute, 164
 - concurrent sessions, 44
 - data dictionary, use of, 64
 - introduction, 21, 28
 - repository sessions, opening, 31
 - supported format converters, 191
 - transactions, 52
 - user authentication, 84
 - Content Storage Services license, 137
 - content-addressed storage areas
 - metadata fields, setting, 138
 - retention periods, 128
 - retention policies, effect of, 129
 - control_flag attribute, 204
 - convert.tbl file, 185, 187
 - copy_child attribute, 169
 - CREATE...TYPE (statement), 73
 - CURRENT version label, 119
 - _current_state attribute (computed attribute), 298
 - custom ACLs
 - creating, 148
 - naming convention, 148
- D**
- data dictionary
 - attribute default values, 70
 - cache, 46, 51
 - components for object types, 70
 - Content Server use of, 64
 - dd attr info objects, 65
 - dd common info objects, 65
 - dd type info objects, 65
 - default lifecycles for types, 69
 - described, 64
 - ignore_immutable attribute, 125
 - lifecycle state information,
 - defining, 71
 - locales, supported, 64
 - localized text, 70
 - mapping information, 71
 - modifying, 65
 - object type constraints, 66, 69
 - publishing, 65
 - retrieving information, 71
 - value assistance, 70
 - data validation, *see* check constraints
 - database-level locking, 54, 126
 - dd attr info objects, 65
 - dd common info objects, 65
 - dd type info objects, 65
 - dd_change_count attribute, 52
 - deadlocks, managing, 54
 - default ACLs, assigning, 147
 - default storage algorithm for content, 137
 - default values, for attributes, 70
 - default_acl attribute, 147
 - default_app_permit attribute, 86
 - default_client_codepage attribute, 108
 - default_folder attribute, 133
 - delegation
 - control_flag attribute, effects of, 204
 - defined, 204
 - enable_workitem_mgmt (server.ini key), 205
 - deleting, *see* destroying; removing
 - deletions, forced, 130
 - demotion in lifecycles, 281
 - Dequeue method, 305
 - destroying
 - activity definitions, 260
 - lifecycles, 299
 - objects, 74 to 75
 - process definitions, 260
 - reference links, 327
 - relationships, effect on, 160
 - versions, 122
 - digital shredding, 104
 - digital signatures
 - a_is_signed attribute, 102
 - described, 101
 - lifecycle states, 291
 - Disassemble method, 178
 - Disconnect method, 32
 - discussions, described, 24
 - distributed notification, 261, 324
 - distributed repositories
 - mirror objects, 152
 - reference links, 152
 - reference objects, 153
 - distributed workflows, 260
 - dm name prefixes, 58
 - dm_acl type, 91
 - dm_addesignature events, audit trail entries, 99
 - dm_alias_set type, 213, 310
 - dm_APIDeInit, 32
 - dm_bp_batch_java method
 - log files, generating, 293

- dm_bp_schedule method, 274
- dm_bp_schedule_java method, 274
- dm_bp_transition method
 - described, 274
 - log file, generating, 293
- dm_bp_transition_java method
 - described, 274
 - log files, generating, 293
- dm_bp_validate method
 - described, 267
- dm_bp_validate_java method
 - described, 267
 - log files, generating, 293
- dm_bpactionproc (procedure), 286
- dm_bpm_timer method, 252
- dm_bpm_transition method, 249
- dm_changepriorityworkitem event, 231
- dm_completedworkitem event, 253
- dm_DistOperations job, 329
- dm_owner
 - default object-level permissions, 90
- dm_policy type, 267
- dm_queue (view), 304
- dm_reference type, 321
- dm_retention_managers group, 129 to 130
- dm_retention_users group, 129
- DM_SESSION_DD_LOCALE (keyword), 72
- dm_sig_template page modifier, 98
- dm_WfmsTimer job, 225, 252
- dm_WFReporting job
 - auditing requirements, 254
 - described, 254
- dm_WFSuspendTimer job, 225, 253
- dm_workflow objects, 228
- dmapp.h file, 330
- dmc_completed_workflow objects, 254
- dmc_completed_workitem objects, 254
- dmc_wf_package_schema type, 216
- dmc_wf_package_skill type, 232
- DMCL (client library)
 - application access control token retrieval, 43
 - connection pooling, 34
 - distributed notification, 324
 - multi-repository transactions, 329
 - object and type caches, 320, 326
 - persistent caches, 45
 - reference link refreshes, 329
 - reference links, 321 to 322
 - replica support, 322
 - repository scope, 320
 - security, 323
 - subconnections, 320
- dmcl.ini file, 30
- dmi_package objects, 232
- dmi_package type, 217
- dmi_queue_item objects, 303
- dmi_wf_attachment type, 233
- dmi_wf_timer objects, 224
- dmi_workitem objects, 229
- dmSendToList2 workflow template, 200
- dmtkgen utility, 42
- docbase_scope attribute, 44
- Docbasic
 - actions on entry (lifecycles), 286
 - entry criteria (lifecycles), 283
 - internationalization, 112
 - post-entry actions (lifecycles), 288
- documents, 116
 - See also* virtual documents
 - ACL, replacing, 151
 - ACLs, assigning, 138, 147 to 148
 - annotations, 154 to 156
 - attributes, setting, 132, 142
 - bindFile method, 145
 - Branch method, 121
 - branching, 121
 - checking out, 141
 - content files
 - adding, 135, 143
 - format specification, 186
 - Macintosh-generated, 136
 - page numbers, 117, 145
 - removing, 145
 - sharing, 145
 - specifying storage, 136
 - content objects and, 116
 - creating, 132
 - deleting with unexpired retention, 130
 - described, 116
 - dm_document type, 116
 - fetching, 141
 - forced deletions, 130
 - immutability, 123
 - keywords, assigning, 133
 - lifecycles and, 131
 - linking to cabinets/folders, 134

- locking strategies, 126
 - modifying, 140
 - permissions, revoking, 150
 - primary location default, 134
 - privileged delete, 130
 - Prune method, 122
 - renditions
 - creating, 184
 - defined, 117, 183
 - removing, 131, 190
 - retention control
 - deletion and, 128
 - effect on modification ability, 140
 - saving, 140, 146
 - translation relationships, 153
 - versions
 - described, 118
 - removing, 122, 131
 - version tree, 120
 - Documentum Administrator, 28
 - domain groups, 83
 - dormant state
 - activity instance, 236
 - work items, 238
 - workflows, 235
 - DQL (Document Query Language)
 - data dictionary, querying, 72
 - query result objects, 58
 - reference links and, 325
 - draft state (workflow definitions), 226
 - DROP TYPE (statement), 74
 - DROP_INDEX administration method, 63
 - dump operations, 113
 - Dumploginticket method, 36
 - dynamic groups
 - described, 84
 - non-dynamic group as member, 84
- ## E
- early binding
 - defined, 166
 - virtual document components, 166
 - electronic signatures, 95
 - See also* digital signatures; Signoff method
 - Addesignature behavior, 97
 - content handling, default, 99
 - customizing, 100
 - described, 95
 - lifecycle states, 291
 - PDF Fusion library and license, 98
 - signature creation method, default, 98
 - signature page template, default, 98
 - verifying, 101
 - work items, 231
 - enable_workitem_mgmt (server.ini key)
 - delegation and, 205
 - halting activities and, 237, 256
 - priority values and, 231
 - purpose, 234
 - encapsulation, specifying in format, 187
 - encryption
 - login ticket key, 36
 - password, 85
 - Encryptpass method, 85
 - End activities, 201
 - entry criteria (lifecycles)
 - Boolean expressions, 284
 - described, 266, 282
 - Docbasic programs, 283
 - Java programs, 282
 - _entry_criteria (computed attribute), 284
 - entry_criteria_id attribute, 285
 - esign_pdf method object, 98
 - events
 - auditing, 93
 - defined, 301
 - distributed notification, 324
 - Getevents method, 304
 - notifications, 306
 - registrations for
 - establishing, 306
 - obtaining information about, 307
 - removing, 307
 - trigger, for workflow activities, 215
 - exception states (lifecycles)
 - described, 264
 - resuming from exception state, 271
 - exceptional route case, 222
 - execute method, 238, 241
 - explicit transactions
 - constraints, 53, 329
 - database-level locking, 54, 126
 - deadlock management, 55
 - defined, 53
 - extended permissions, 89
 - extension (workflow activities), 204
 - extensions, *see* state extensions
 - extents, for object type tables, 62

external ACLs, 92

F

failed state (activity instance), 237

federated repositories

internationalization, 112

mirror objects, 152

reference links, 152

reference objects, 153

Fetch method

described, 141

locking and, 127

file formats

conversions on Windows, 191

encapsulation specification, 187

full format specification, 186, 189

PBM Image converters, 192

renditions

described, 117, 183

removing, 190

resolution specification, 186

supported converters, 191

transformation algorithm, 187

transformation path, 188

transforming with Unix utilities, 193

files, *see* content files

finished state

activity instance, 237

work items, 238

workflows, 235

folder security, 91

folders

default ACL, 147

linking documents, 134

reference link behavior, 322

follow_assembly attribute, 169

forced deletions, 130

foreign key constraints, 68

format

alias specification, 212

application access control token, 41

login ticket, 36

formats, *see* file formats

Freeze method, 124, 179

G

GET_INBOX (administration

method), 304

Getconnection method, 31, 44

Getevents method, 304

Getfile method, renditions and, 184

global attributes, 59

grantPermit method, 149

group_class attribute, 83

groups, 83

See also dynamic groups

alias sets for, 312

code page requirements, 111

described, 83

membership constraint for

non-dynamic groups, 84

mixing non-dynamic and dynamic

groups in membership, 84

ownership of objects, 133

role, 83

standard, 83

H

Halt method

automatic resumption and, 225

use of, 255

halted state

activity instances, 237

workflows, 235

haltEx method, 237

hierarchy, object type, 58

home repository, 302

I

i_performer_flag attribute, 246

i_vstamp attribute, 51

i_vstamp attribute, locking and, 127

identifiers

object type, 77

subconnection, 44, 320

ttypename, 72

IDfWorkflowAttachment interface, 233

ignore_immutable attribute, 125

images

PMB Image converters, 192

transforming, 192

immutability

attributes, effect on, 125

described, 123

ignore_immutable attribute, 125

retention policies and, 125

- implicit version labels, 119
 - inboxes
 - Anyevents method, 303
 - contents, 302
 - Dequeue method, 305
 - described, 302
 - dm_queue view, 304
 - dmi_queue_item objects, 303
 - GET_INBOX (administration method), 304
 - Getevents method, 304
 - home repository and, 302
 - obtaining event registrations, 307
 - obtaining stamp value, 305
 - Queue method, 304
 - viewing queue, 303
 - _included_types (computed attribute), 298
 - indexes for object types, 63
 - indirect references, 322
 - inheritance, 58
 - input ports
 - Begin activities and, 202
 - defined, 216
 - packages, 242, 249
 - Insertpart method, 163, 170
 - Install method, 228
 - installed state (workflow definitions), 226
 - internal ACLs, 92
 - internal transactions
 - deadlock management, 54
 - described, 53
 - internationalization
 - api config object, 108
 - ASCII support, 107
 - ASCII use requirements, 111
 - code pages, 107
 - content files, 106
 - data dictionary support for, 70
 - databases, 107
 - described, 105
 - Docbasic, 112
 - dump and load operations, 113
 - federations, ASCII requirements, 112
 - lifecycle constraint, 112
 - locales, 105
 - metadata, 107
 - National Character Sets, 106
 - object replication, 113
 - repository sessions, values set, 108
 - required parameters, 107
 - server config object, 108
 - session config object, 109
 - Unicode, 106
 - UTF-8, 106
 - _is_restricted_session (computed attribute), 32
- ## J
- Java
 - action on entry programs (lifecycles), 286
 - entry criteria (lifecycles), 282
 - post-entry actions (lifecycles), 288
 - jobs
 - dm_DistOperations, 329
 - dm_WfmsTimer, 225, 252
 - dm_WFReporting, 254
 - dm_WFSuspendTimer, 225, 253
 - lifecycle state transition jobs, 272
- ## K
- keep_flag argument, obtaining setting, 190
 - keywords attribute, 133
- ## L
- language_code attribute, 134, 153
 - late binding
 - defined, 167
 - post-entry actions (lifecycles), 289
 - user actions (lifecycles), 287
 - user criteria (lifecycles), 284
 - virtual document components, 167
 - lifecycle states
 - actions on entry
 - defining, 285
 - Docbasic, 286
 - execution order, 285
 - Java, 286
 - system-defined, 285
 - aliases in, 276
 - attachability, 279
 - attaching objects, 269
 - base state, returning to, 280
 - batch promotion, 270
 - computed attributes for, 298
 - data dictionary and, 71

- definitions
 - described, 278
 - modifying, 297
 - demoting from, 270, 281
 - entry criteria
 - described, 282
 - Docbasic, 283
 - entry_criteria_id attribute, 285
 - Java, 282
 - movement between, 270
 - naming rules, 279
 - normal states, 264
 - post-entry actions
 - described, 287
 - Docbasic, 288
 - Java, 288
 - promoting to, 270
 - sign-offs, enforcing, 291
 - state extensions
 - adding, 294
 - described, 276
 - state type definitions, 277
 - state_class attribute, 278
 - suspending from, 271
 - transitions, scheduled, 272, 281
 - user_action_ver attribute, 287
 - user_criteria_id attribute, 283
 - user_criteria_ver attribute, 284
 - user_postproc_ver attribute, 289
- lifecycles
- a_bpaction_run_as attribute, 274
 - action_object_id attribute, 285
 - actions, described, 266
 - alias scope, defining, 313
 - alias use in actions, 291
 - aliases and alias sets, 276
 - allow_demote attribute, 281
 - attaching objects, 269
 - batch promotion, 270
 - bp_actionproc.ebs, 286
 - changing, 297
 - code page requirements, 112
 - computed attributes for, 298
 - defaults for object types, 69, 266
 - defined, 131, 263
 - See also* lifecycle states
 - definition states, 267
 - demotion, 270
 - destroying, 299
 - entry criteria, described, 266
 - exception states, 264
 - installation, 268
 - log files, 275
 - methods supporting, 273
 - notifications of uninstall/reinstall, 297
 - object types for, 265
 - object-level permissions and, 275
 - primary object type for, 265
 - programming languages, supported, 266
 - progression through, overview, 268
 - promotion, 270
 - repository storage, 267
 - resumption from exception state, 271
 - state change behavior, 274
 - state definitions, 278
 - state extensions, 276, 294
 - state types, 277
 - state-change methods, 270, 274
 - suspension from state, 271
 - system_action_state attribute, 285
 - testing and debugging, 292
 - validation
 - custom programs, 268, 295
 - Docbasic programs, 296
 - Java programs, 296
 - overview, 267
- Link method
- alias use in, 309
 - resolving aliases, 314
- linking, folder security and, 91
- links
- described, 202
- links (workflow)
- compatibility, 227
 - port compatibility, 218
- load operations, 113
- local attributes, 59
- locale_name attribute, 108
- locales
- described, 64
 - DM_SESSION_DD_LOCALE
 - keyword, 72
 - session_locale attribute, 109
 - supported, 65, 105
- location objects
- SigManifest, 98
- locking
- database level, 54, 126

- optimistic, 127
 - repository level, 126
 - strategies, 126 to 127
- log files, lifecycle, 275, 293
- login failure, auditing, 93
- login ticket key
 - application access control tokens
 - and, 41
 - described, 36
 - ticket_crypto_key attribute, 36
- login tickets
 - described, 35
 - Dumploginticket method, 36
 - expiration, configuring, 38
 - format, 36
 - login_ticket_cutoff attribute, 39
 - login_ticket_timeout attribute, 38
 - max_login_ticket_timeout
 - attribute, 38
 - revoking, 39
 - scope, 37
 - single use, 36
 - superuser use, restricting, 39
 - time difference tolerances, 38
 - timed-out sessions, reconnecting, 32
 - trusted repositories and, 37
- login_ticket_cutoff attribute, 39
- login_ticket_timeout attribute, 38
- LTK, *see* login ticket key

M

- Macintosh files, adding as content, 136
- MAKE_INDEX administration
 - method, 63
- manual activities
 - delegation, 204
 - described, 203
 - effects of Complete method, 231
 - extension characteristic, 205
 - manual transitions
 - described, 220
 - output choice behavior, 220
 - output choices, limiting, 220
 - performers
 - alias use, 212
 - categories of, 210
 - resolving, 243
 - priority values, 203
- mapping information (data dictionary), 71

- Mark method, 119
- max_login_ticket_timeout attribute, 38
- Media Transformation Services,
 - renditions, 184
- messaging, distributed, *see* distributed notification
- metadata
 - National Character Sets and, 107
 - setting in content-addressed storage, 138
- method_data attribute, in
 - dm_WFReporting job, 254
- methods
 - application access control tokens
 - and, 43
 - lifecycle, 273
 - repository scope, 43
 - scoped arguments, 320
- mirror objects
 - described, 152, 321
 - refreshing, 329
- @mirror_object_id, 322
- mode parameter values, 246

N

- names
 - activity definitions, 202
 - custom ACLs, 148
 - lifecycle states, 279
 - prefixes, system-defined, 58
- National Character Sets, 106
- new connection callback attributes, 331
- _next_state (computed attribute), 298
- non-persistent objects, 58
- normal states (lifecycle), 264
- not null constraints, 69
- notes, 24
 - See also* annotations
- notification in distributed workflows, 261
- notifications of events, 306
- NULL values
 - foreign keys and, 68
 - not null constraints, 69
 - unique keys and, 67

O

- object caches
 - consistency checking, 51

- described, 45
 - DMCL, 320, 326
 - object replication
 - internationalization, 113
 - relationships and, 160
 - object type tables
 - described, 60
 - extent size, defining, 62
 - tablespace, defining, 62
 - object types
 - attributes, defaults for, 70
 - component routines for, 70
 - content files and, 59
 - creating, 73
 - data dictionary information
 - constraints, 66
 - default lifecycle for type, 69, 266
 - mapping information, 71
 - value assistance, 70
 - (data dictionary information
 - constraints, 69
 - data dictionary information,
 - retrieving, 71
 - dd_attr_info, 65
 - dd_common_info, 65
 - dd_type_info, 65
 - default ACLs for, 147
 - defined, 57
 - dm name prefix, 58
 - identifiers, 77
 - indexes on, 63
 - owner, 73
 - persistence, 58
 - primary for lifecycles, 265
 - RDBMS tables for, 60
 - reference link sources, 321
 - removing, 74
 - subtypes, described, 58
 - supertypes, 58
 - SysObjects, 115
 - valid types for lifecycles, 265
 - object-level permissions, 88
 - See also* ACLs
 - assigning, 138
 - described, 88
 - lifecycles and, 275
 - revoking, 150
 - objects, 57
 - See also* SysObjects
 - alias use in, 309
 - attaching to lifecycles, 269
 - changing to another type, 76
 - creating, 74
 - default owner permissions, 90
 - default superuser permissions, 90
 - described, 57
 - destroying, 74 to 75
 - freezing, 124
 - global and local attributes, 59
 - ignore_immutable attribute, 125
 - immutability, 123
 - information about, obtaining through
 - API, 72
 - ownership, assigning, 133
 - persistence, 58
 - relationships
 - creating, 158
 - destroying, 159
 - user-defined, 157
 - unfreezing, 124
 - optimistic locking, 127
 - output ports
 - defined, 216
 - End activities and, 202
 - selecting conditionally, 221
 - ownership of objects, 133
- ## P
- package control (workflows), 225
 - package definitions
 - adding, 259
 - compatibility, 218
 - described, 216
 - empty, 217
 - removing, 259
 - scope, 217
 - validation of, 218
 - package object type, 217
 - packages
 - acceptance protocol, 249
 - adding to Begin activities, 241
 - consolidation of, 242
 - empty, 217
 - package objects, 232
 - package skill level, 232
 - skill level requirements, 217
 - visibility of, 217
 - page numbers, for content, 117, 145
 - parent, in foreign key, 68

- parent_id attribute, 158
 - password encryption, 85
 - path_name attribute, 181
 - paused state, for work items, 238
 - PBM Image converters, 192
 - PDF Fusion library and license, 98
 - percent sign (%) in alias specification, 310
 - performer aliases, format, 212
 - permanent_link attribute, 156, 159
 - persistence, 58
 - persistent client caches
 - cache..map file, 47
 - consistency checking, 49
 - described, 45
 - identifying data to cache, 48
 - object caches
 - consistency checking, 51
 - described, 45
 - query caches, 46 to 47
 - subconnections and, 47
 - using, 48
 - policy object type, 267
 - _policy_name attribute (computed attribute), 298
 - ports, 215
 - See also* input ports; output ports
 - adding to activities, 259
 - compatibility, 218, 227
 - described, 215
 - package definitions
 - described, 216
 - validation, 218
 - removing from activities, 259
 - required, 216
 - post-entry actions (lifecycles)
 - aliases in, 291
 - defining, 287
 - described, 266
 - Docbasic programs, 288
 - Java programs, 288
 - post-timers (workflows)
 - described, 224
 - instantiation, 252
 - pre-timers (workflows)
 - described, 224
 - instantiation, 252
 - predicate_id attribute, 223
 - prescribed activity transitions, 219
 - _previous_state (computed attribute), 298
 - primary cabinet, *see* primary location
 - primary content files, *see* content files
 - primary folder, 147
 - primary key constraints, 67
 - primary location, 133
 - priority values
 - activity (workflow), 203
 - work items, setting for, 230
 - private ACLs
 - assigning to documents, 148
 - described, 93
 - private groups, 83
 - privileged delete, 130
 - See also* forced deletions
 - procedures as reference links, 322
 - process definitions, 201
 - See also* workflow definitions
 - activity types, 201
 - described, 201
 - destroying, 260
 - installing, 228
 - links, 202
 - modifying, 257
 - re-installing, 258
 - uninstalling, 257
 - validating, 226
 - validation checks, 227
 - versioning, 258
 - promote method, testing, 293
 - Prune method, 122
 - public ACLs
 - assigning to documents, 148
 - described, 93
 - public groups, 83
 - publishing data dictionary, 65
 - Purge Audit user privileges, 88
- ## Q
- qual comp objects, 70
 - query caches, 46 to 47
 - query objects as reference links, 326
 - query result objects, 58
 - queue items
 - obtaining stamp value, 305
 - placing in inbox, 304
 - work items and, 229
 - Queue method, 304
 - queues, *see* inboxes

R

- _r repository tables, 61
- r_alias_set_id attribute, 312
- r_chronicle_id attribute, 120
- r_complete_witem attribute, 246
- r_condition_id attribute, 222
- r_condition_name attribute, 223
- r_condition_port attribute, 223
- r_current_state attribute, 299
- r_definition_state attribute, 267
- r_frozen_flag attribute, 124, 179 to 180
- r_frzn_assembly_cnt attribute, 179 to 180
- r_has_frzn_assembly attribute, 179 to 180
- r_immutable_flag attribute, 123 to 124, 179
- r_is_virtual_doc attribute, 163, 170
- r_last_changed_date attribute, 51
- r_link_cnt attribute, 163
- r_policy_id attribute, 299
- r_predicate_id attribute, 222
- r_resume_state attribute, 299
- r_signoff_user attribute, 305
- r_total_witem attribute, 246
- r_version_label attribute, 118
- RDBMS
 - database-level locking, 126
 - Documentum tables in, 60
 - object type indexes, 63
 - _r repository tables, 61
 - registered tables, 64
 - _s repository tables, 61
- reention policies
 - virtual documents, 165
- reference links
 - binding, 321, 326
 - described, 152
 - dm_DistOperations job, 329
 - implementation, 321 to 322
 - reference_binding_label attribute, 326
 - referencing in applications, 325
 - Refresh method, 330
 - refreshing, 329
 - security, 323
 - source object types, 321
 - storage location, 322
 - type-specific behaviors, 322
 - using in DQL, 325
 - valid operations for, 327
- reference objects, 153
- reference_binding_label attribute, 326
- referential integrity, 164
- Refresh method, 330
- Register method, 306
- registered tables, 64
- registry objects, 94
- relation objects
 - annotations and, 155
 - creating, 158
 - described, 157
- relation_name attribute, 158
- relationships
 - Branch method, 159
 - Checkin method, 159
 - creating instances, 158
 - object removal, effect of, 160
 - object replication and, 160
 - relation object, 157
 - Save method and, 160
 - Saveasnew method and, 159
 - translation relationships, 153
 - user-defined, 157
- remote users, work items and, 261
- remote_pending attribute, 324
- remove methods, 143
- removeAttachment method, 233
- removeContent method, 145
- Removenote method, 156
- Removepackageinfo method, 259
- Removepart method, 172
- Removeport method, 259
- Removerendition method, 190
- removing, 145
 - See also* destroying
 - aborted workflows, 256
 - content files, 145
 - event registrations, 307
 - queued inbox items, 305
 - renditions, 190
 - repeating attribute values, 143
 - user-defined types, 74
 - versions, 122
 - virtual document components, 172
- rendition attribute, use of, 190
- renditions
 - convert.tbl file, 185, 187
 - creating, 184
 - defined, 117, 183
 - encapsulation specification, 187
 - file formats, 185, 189

- Getfile method and, 184
 - keep_flag setting, obtaining, 190
 - PBM Image converters, 192
 - Removerendition method, 190
 - removing, 131, 190
 - resolution specification, 186
 - supported format converters, 191
 - transformation algorithm, 187
 - transformation path, 188
- Repeat method, 205
- repeatable_invoke attribute, 205
- repeating attributes
- adding values, 142
 - performance tip, 143
 - removing values, 143
 - replacing values, 143
 - storage, 61
- replicas
- described, 153, 322
 - operations on, 327
 - retention policies and, 129
 - retrieving content, 328
 - security, 324
- repositories
- a_bpaction_run_as attribute, 274
 - aborted workflows, removing, 256
 - annotations, deleting, 156
 - application access control tokens, 39
 - application access control tokens, generating, 42
 - architecture, 60
 - auditing events, 93
 - data dictionary
 - described, 64
 - retrieving information, 71
 - default_acl attribute, 147
 - distributed notification, 324
 - home repository, 302
 - inboxes, 302
 - localizing, 70
 - login ticket use, 35
 - mirror objects, 321
 - object types
 - creating, 73
 - RDBMS indexes, 63
 - RDBMS tables, 60
 - removing, 74
 - reference links
 - described, 152
 - storage location, 322
 - repository objects
 - changing type, 76
 - creating, 74
 - destroying, 75
 - security, 87
 - trusted mode, for login tickets, 37
 - user authentication, 84
 - working with remote objects, 151
 - repository scope
 - described, 43
 - docbase_scope attribute, 44
 - specifying in methods, 43, 320
 - repository sessions
 - alias sets for, 311
 - caches in, 320
 - closing, 32
 - configuration objects, 30
 - connection pooling, 34
 - deadlocks, managing, 54
 - default_app_permit attribute, 86
 - described, 29
 - dmcl.ini file, 30
 - docbase_scope attribute, 44
 - explicit transactions, 52
 - Getconnection method, 44
 - inactive, 32
 - internationalization, 108
 - _is_restricted_session (computed attribute), 32
 - maximum number, 44
 - opening, 31
 - restricted, 32
 - secure connections, 34
 - session config objects, 109
 - session_codepage attribute, 109
 - session_locale attribute, 109
 - timed out sessions, reconnecting, 32
 - transactions, managing, 52
 - repository-level locking, 126
 - resetPassword (IDfSession), 32
 - resolution, specifying in format, 186
 - resolve_pkg_name attribute, 213, 316
 - resolve_type attribute, 213, 316 to 317
 - respository sessions
 - default_client_codepage attribute, 108
 - Restart method, 256
 - resume method, 272, 293
 - Resume method, 256
 - _resume_state (computed attribute), 299

- retention policies
 - deleting documents under
 - control, 130
 - described, 128
 - description, 128
 - dm_retention_managers group, 129
 - dm_retention_users group, 129
 - document versions and, 129
 - privileged delete, 130
 - r_immutable_flag and, 125
 - replicas and, 129
 - storage-based retention, interaction with, 129
 - SysObjects, modification
 - constraints, 140
 - Retention Policy Services, 24
 - return_to_base attribute, 280
 - revert ports, 216
 - revokePermit method, 149
 - role groups, 83
 - rooms, described, 23
 - route cases
 - described, 221
 - evaluation, 249
 - non-XPath implementation, 222
 - r_condition_id and r_predicate_id
 - compatibility, 223
 - selected ports, recording in repository, 223
 - XPath implementation, 222
- S**
- _s repository tables, 61
 - Save method
 - described, 146
 - effect on relationships, 160
 - Saveasnew method, 159
 - scope
 - alias, 311
 - application access control tokens, 41
 - login tickets, 37
 - method, 43, 320
 - method arguments, 320
 - repository, 43, 320
 - scripts as reference links, 322
 - secure connections, described, 34
 - security
 - cached query files, 47
 - digital shredding, 104
 - digital signatures, 101
 - folder, 91
 - permissions, revoking, 150
 - reference links, 323
 - replicas, 324
 - secure connections, use of, 34
 - signature requirements support, 95
 - user privileges and permissions, 87
 - security_mode attribute, 87
 - SELECT (statement)
 - processing algorithm, 174
 - server config object, 30, 108
 - server.ini file
 - enable_workitem_mgmt key, 234
 - server_os_codepage attribute, 108
 - session code page
 - dump and load, with, 113
 - session_codepage attribute, 109
 - session config objects, 30, 109
 - session_locale attribute, 109
 - sessions, *see* repository sessions
 - Set method, 132
 - SET_APIDEADLOCK administration
 - method, 55
 - Setdoc method, 170
 - Setoutput method, 220
 - Setpriority method, 230
 - shredding, digital, *see* digital shredding
 - SigManifest location object, 98
 - sign-offs, simple, 102
 - signature creation method, default, 98
 - signature page templates, default, 98
 - signature requirements, support for, 95
 - See also* digital signatures; electronic signatures; sign-offs, simple
 - signatures, digital, *see* digital signatures
 - signatures, electronic, *see* electronic signatures
 - Signoff method
 - auditing execution, 93
 - simple sign-off, use in, 102
 - workflow task sign-off, 305
 - sigpage.doc, 98
 - sigpage.pdf, 98
 - Smart Lists, as reference links, 326
 - snapshots
 - assembly object type, 163
 - components
 - adding, 177
 - deleting, 178

- creating, 176
 - described, 164, 175
 - Disassemble method, 178
 - freezing, 179
 - modifying, 177
 - path_name attribute, 181
 - unfreezing, 179
 - source_docbase attribute, 324
 - source_event_id attribute, 325
 - source_event_stamp attribute, 325
 - SSL (secure socket layer) protocol, 34
 - standard groups
 - described, 83
 - starting condition (activities)
 - described, 215
 - evaluating, 242
 - state extensions
 - adding to lifecycles, 294
 - described, 276
 - state types, for lifecycles, 277
 - state_class attribute, 278
 - _state_type (computed attribute), 298
 - states
 - work items, 237
 - workflow, 235
 - workflow definition objects, 226
 - Step activities
 - described, 201
 - required ports, 216
 - storage areas
 - assigning content to, 136
 - digital shredding, 104
 - retention periods, 129
 - subconnections
 - connection pooling and, 35
 - described, 31, 320
 - identifiers, 44, 320
 - persistent client caching and, 47
 - subtypes
 - described, 58
 - owner, 73
 - removing, 74
 - supertypes, 58
 - Superuser user privilege
 - a_full_text attribute and, 142
 - application access control tokens
 - and, 40
 - login tickets and, 39
 - workflow supervisor and, 233
 - superusers
 - default object-level permissions, 90
 - supervisor, workflow, 233
 - supported format converters, 191
 - suspend method, 271
 - suspend timers
 - described, 225
 - dm_WFSuspendTimer job, 253
 - implementation, 251
 - instantiation, 252
 - symbolic linking, 166
 - symbolic version labels, 119
 - Sysadmin user privilege
 - workflow supervisor and, 233
 - SysObjects
 - ACLs, assigning, 147
 - alias use in, 309
 - annotations, adding, 156
 - application-level control, 86
 - content files
 - adding, 143 to 144
 - replacing, 145
 - default_folder attribute, 133
 - described, 115
 - keywords attribute, 133
 - language_code attribute, 134
 - lifecycle scope, defining, 313
 - modifying, 140
 - ownership, assigning, 133
 - primary folder, defined, 147
 - r_alias_set_id attribute, 312
 - r_chronicle_id attribute, 120
 - r_is_virtual_doc attribute, 163, 170
 - r_link_cnt attribute, 163
 - r_version_label attribute, 118
 - removeContent method, 145
 - repeating attributes, modifying, 142
 - resolving aliases, 313
 - retention control, effects of, 128, 140
 - saving, 140
 - state information, obtaining, 298
 - version labels, 118
 - version tree, 120
 - system ACLs
 - assigning to documents, 148
 - public ACLs and, 93
- T**
- tablespaces for object type tables, 62
 - target_docbase attribute, 324

- task subjects, 213
 - task_subject attribute, 213
 - task_subject attribute (activities), 214
 - task_subject attribute (queue items), 214
 - template ACLs, 93
 - template signature pages, 98
 - template workflows
 - alias usage, 212
 - described, 199
 - dmSendToList2, 200
 - terminated state (workflows), 235
 - ticket_crypto_key attribute, 36
 - timers, for workflow, 223
 - tokens, *see* application access control
 - tokens
 - tracing, overview, 94
 - transactions
 - database-level locking, 126
 - deadlocks, managing, 54
 - defined, 52
 - explicit, 53, 329
 - locking strategies, 126 to 127
 - TRANSCODE_CONTENT administration
 - method, 184
 - transformations
 - algorithm for, 187
 - convert.tbl file, 187
 - loss values, 187
 - path description, 188
 - PBM Image converters, 192
 - renditions, 183
 - supported converters, 191
 - using UNIX utilities, 193
 - transition condition objects, 222
 - transition types, for activities, 219
 - transition_eval_cnt attribute, 219
 - transition_flag attribute, 221
 - transition_max_output_cnt attribute, 220
 - transitions, activity, 219
 - translation relationships, 153
 - trigger condition (activities), 215
 - trigger events (activities), 215
 - trust_by_default attribute, 37
 - Trusted Content Services
 - ACLs and, 92, 148, 150
 - trusted_docbases attribute, 37
 - typename identifiers, 72
 - type caches, in DMCL
 - connections and, 320
 - consistency checking, 51
 - described, 46
 - mirror objects and, 326
 - type identifiers, 77
 - type_change_count attribute, 52
- ## U
- Unaudit method, auditing, 93
 - Unfreeze method, 124, 179
 - Unicode, 106
 - unique key constraint, 67
 - Unlink method
 - alias use in, 309
 - resolving aliases, 314
 - unlinking, folder security and, 91
 - Unlock method, 127
 - Unregister method, 307
 - Updatepart method, 172
 - use_node_ver_label attribute, 167
 - user authentication, 84
 - user privileges
 - Config Audit, 88
 - list of, 87
 - Purge Audit, 88
 - View Audit, 88
 - user_action_id attribute, 287
 - user_action_service attribute, 285
 - user_action_ver attribute, 287
 - user_criteria_id attribute, 283
 - user_criteria_ver attribute, 284
 - user-defined types, removing, 74
 - user_delegation attribute, 204
 - user_postproc_id attribute, 289
 - user_postproc_ver attribute, 289
 - users
 - alias sets for, 312
 - alias_set_id attribute, 312
 - code page requirements, 111
 - default ACL, 147
 - described, 82
 - login failure, auditing, 93
 - object-level permissions, 88
 - user privileges, list of, 87
 - workflow_disabled attribute, 82
 - UTF-8, 106
 - utilities
 - dmtkgen, 42

V

Validate method, 227
 validated state (workflow definitions), 226
 validation, lifecycle definitions, 267, 295
 ValidationProc Docbasic program, 296
 validity periods

- application access control tokens, 42
- login tickets, 38

 value assistance, 70
 Vdmpath method, 182
 Vdmpathdql method, 182
 verification of electronic signatures, 101
 Verifiesignature method, permissions needed, 101
 versions

- Branch method, 121
- branching, 121
- Destroy method and, 122
- implicit version label, 119
- label uniqueness, 120
- Mark method and, 119
- Prune method and, 122
- r_chronicle_id attribute, 120
- r_version_label attribute, 118
- removing, 76, 122, 131
- retention policies and, 129
- symbolic version label, 119
- SysObjects and, 118
- version tree, 120

 View Audit user privileges, 88
 view on inboxes, 304
 virtual documents

- assembling, 173
- assembly objects, 163
- changes, saving, 171
- components
 - adding, 171
 - appending, 170
 - assembly behavior, defining, 167, 172
 - determining paths to, 181
 - early binding, 166
 - inserting, 170
 - late binding, 167
 - ordering, 163, 172
 - removing, 172
 - selecting for snapshot, 173
- compound_integrity attribute, 164
- conditional assembly, 164

- containment objects
 - copy_child attribute, 169
 - described, 162
 - Updatepart method, 172
- content files and, 165
- copy behavior, defining, 169
- creating, 169
- described, 116, 162
- freezing, 179
- permissions to modify, 171
- querying, 180
- r_is_virtual_doc attribute, 163, 170
- r_link_cnt attribute, 163
- referential integrity, 164
- retention policies, 165
- snapshots
 - assembly objects, 163
 - creating, 176
 - described, 164, 175
 - disassembling, 178
- unfreezing, 179
- Vdmpath method, 182
- Vdmpathdql method, 182
- versioning, 163

W

warning timers

- described, 224
- dm_WfmsTimer job, 252
- implementation, 251

 Webtop Workflow Reporting tool, 254
 wf attachment objects, 233
 wf package schema object type, 216
 wf package skill object type, 232
 WfmsTimer job, 252
 Windows platforms, format conversions supported, 191
 workflow definitions

- described, 202

 work items

- completing, 231
- delegating, 204
- described, 229
- inboxes and, 301
- overview of use, 230
- package skill level, 217
- pausing, 257
- priority, setting, 230
- queue items and, 229

- remote users and, 261
- resuming paused, 257
- signing off, 231, 305
- states of, 237
- user time and cost reporting, 253
- workflow agent
 - activities, assigning, 244
 - activity priority value use, 203
 - batch size, 244
 - described, 234
- workflow definitions
 - activities, naming, 202
 - activity definitions
 - modifying, 259
 - multiple use of, 201
 - alias use in, 199, 309
 - architecture, 201
 - Begin activities, 201
 - End activities, 201
 - modifying, 257
 - package compatibility, 218
 - package control, enabling, 225
 - process definitions
 - modifying, 257
 - validation checks, 227
 - states, 226
 - Step activities, 201
 - templates, 199
 - validating, 226
- workflow instances
 - aborting, 256
 - activity instances, 229
 - completion, evaluating, 246
 - evaluating transition
 - conditions, 249
 - halting, 255
 - attachments, 233
 - halting
 - described, 255
 - effect on activities, 255
 - notes, for packages, 232
 - reinstalling definition, effects of, 258
 - reports about, 254
 - resolving performers
 - aliases, 244
 - automatic activities, 243
 - manual activities, 243
 - workflow_disabled attribute, 243
- restarting
 - failed activities, 256
 - workflow, 256
- resuming
 - halted workflows, 256
 - paused activities, 256
- starting, 238, 241
- states of, 235
- workflow objects, 228
- workflow_disabled attribute, 82, 243
- workflows, 197
 - See also* activities; workflow definitions; workflow instances
 - activities
 - definitions, 202
 - delegation, 204
 - execution, 241
 - extension, 205
 - instances of, 229
 - names of, 202
 - repeatable, 205
 - types of, 201
 - aliases
 - default resolution, 316
 - package resolution algorithm, 316
 - resolving at activity start, 316
 - resolving at startup, 315
 - scopes of, 311
 - user resolution algorithm, 317
 - attachments, 233
 - defined, 197
 - distributed, 260
 - input ports, 216
 - links, 202
 - manual activities, 203
 - output ports, 216
 - package definitions, 216
 - revert ports, 216
 - route cases, 221
 - runtime architecture, 228
 - runtime execution, 238
 - starting, 238, 241
 - supervisor, 233
 - timers, for activities, 223
 - user time and cost reporting, 253
 - work item objects, 229
 - workflow agent, 234
 - workflow_disabled attribute, 82

X

XML files

activity packages, as, 216

support for, 165

XPath expressions

route cases and, 222

validation, 222