

# **EMC® Documentum® Content Server**

**Version 6.5**

**Fundamentals  
P/N 300-007-197-A01**

EMC Corporation  
*Corporate Headquarters:*  
Hopkinton, MA 01748-9103  
1-508-435-1000  
[www.EMC.com](http://www.EMC.com)

Copyright © 1992 - 2008 EMC Corporation. All rights reserved.

Published July 2008

EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

THE INFORMATION IN THIS PUBLICATION IS PROVIDED AS IS. EMC CORPORATION MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND WITH RESPECT TO THE INFORMATION IN THIS PUBLICATION, AND SPECIFICALLY DISCLAIMS IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Use, copying, and distribution of any EMC software described in this publication requires an applicable software license.

For the most up-to-date listing of EMC product names, see EMC Corporation Trademarks on [EMC.com](http://EMC.com).

All other trademarks used herein are the property of their respective owners.

# Table of Contents

---

<b>Preface</b> .....	23
<b>Chapter 1</b>	
<b>Introducing Content Server</b> .....	25
Content Server's role in the product suite .....	25
Standard Content Server services.....	25
Content management services .....	26
Storage and retrieval.....	26
Versioning.....	26
Data dictionary.....	27
Assembly and publishing.....	27
Search services .....	27
For more information.....	28
Process management features.....	28
Workflows .....	28
Lifecycles .....	29
For more information .....	29
Security features.....	29
Repository security .....	30
Accountability.....	30
For more information.....	30
Distributed services .....	31
Additional Content Server offerings.....	31
Trusted Content Services.....	31
What Trusted Content Services is.....	31
For more information.....	32
Content Services for EMC Centera .....	32
What Content Services for EMC Centera is.....	32
For more information.....	33
Content Storage Services .....	33
What Content Storage Services is.....	33
For more information.....	34
EMC Documentum products requiring activation on Content Server .....	34
Retention Policy Services.....	34
What Retention Policy Services is .....	34
For more information.....	35
EMC Documentum Collaborative Services .....	35
What EMC Documentum Collaborative Services are .....	35
For more information.....	36
Internationalization .....	36
Overview .....	36
For more information.....	36
Communicating with Content Server .....	37
Documentum client applications .....	37
Custom applications .....	37
Interactive utilities .....	37

<b>Chapter 2</b>	<b>Session and Transaction Management</b>	39
	What a session is	39
	Multiple sessions	40
	DFC Implementation	40
	Session objects	40
	Obtaining a session	40
	Session identifiers	41
	Shared and private sessions	41
	Shared sessions	41
	Private sessions	41
	Explicit and implicit sessions	41
	Session configuration	42
	The dfc.properties file	42
	The runtime configuration objects	43
	Closing repository sessions	43
	Object state when session closes	43
	Terminating a session manager	43
	For more information	44
	Inactive repository sessions	44
	Restricted sessions	44
	Connection brokers	45
	What the connection broker is	45
	Role of the connection broker	45
	For more information	45
	Native and secure connections	46
	What native and secure connections are	46
	Using secure connections	46
	For more information	47
	Connection pooling	47
	What connection pooling is	47
	How connection pooling works	47
	Simulating connection pooling in an application	48
	For more information	48
	Login tickets	48
	What a login ticket is	48
	Ticket generation	49
	Ticket format	49
	Login ticket scope	49
	Login ticket validity	50
	Ticket expiration	50
	Note on host machine time and login tickets	50
	Revoking tickets	51
	Restricting Superuser use	51
	For more information	51
	Application access control tokens	52
	What application access control tokens are	52
	Benefits of use	52
	How tokens work	52
	Format of AAC tokens	53
	Token scope	54
	Token generation	54
	Token expiration	55
	Internal methods, user methods, and tokens	55
	For more information	55
	The login ticket key	56

	What the login ticket key is .....	56
	Working with the key .....	56
	For more information.....	56
	Trusting and trusted repositories .....	57
	What trusting and trusted repositories are.....	57
	How trusting and trusted repositories are configured .....	57
	Concurrent sessions .....	58
	What concurrent sessions are .....	58
	How concurrent sessions are handled .....	58
	For more information.....	58
	Transaction management .....	58
	What a transaction is.....	58
	Internal and explicit transactions .....	59
	Constraints on explicit transactions.....	59
	Database-level locking in explicit transactions .....	60
	Managing deadlocks.....	60
	Handling deadlocks in internal transactions .....	61
	Handling deadlocks in explicit transactions.....	61
	For more information.....	62
<b>Chapter 3</b>	<b>Caching</b> .....	63
	Object type caching.....	63
	For object types with names beginning with dm, dmr, and dmi .....	63
	For custom object types and types with names beginning with dmc .....	64
	For more information.....	64
	Repository session caches.....	64
	Object cache .....	65
	Query cache .....	65
	Data dictionary caches .....	65
	Persistent caching .....	65
	What persistent caching is.....	65
	Persistent object cache storage location.....	66
	Persistent object type and data dictionary storage location .....	66
	Query cache storage location .....	66
	Using persistent client caching in an application.....	67
	For more information.....	67
	Persistent caching and multiple sessions for one user .....	68
	Currency checking .....	68
	Consistency checking.....	68
	Determining if a consistency check is needed .....	69
	When a keyword or integer defines the rule .....	69
	When a cache config object defines the rule .....	70
	Conducting consistency checks .....	71
	The default consistency check rule .....	71
	The client_pcaching_change property .....	71
	For more information.....	71
<b>Chapter 4</b>	<b>The Data Model</b> .....	73
	Introducing object types and objects .....	73
	What objects and object types are.....	73
	Supertypes, subtypes, and inheritance .....	74
	Persistence .....	74
	Type categories.....	74
	Lightweight and shareable object types.....	75

What a lightweight type is.....	75
What a shareable type is.....	76
For more information.....	76
Names of EMC Documentum object types .....	76
Content files and object types .....	77
For more information.....	77
Properties .....	77
What properties are .....	77
Property characteristics.....	78
Persistent and non-persistent.....	78
Single-valued and repeating .....	78
Datatype .....	78
Read and write or read only .....	79
Qualifiable and non-qualifiable .....	79
Local and global .....	79
Property identifiers.....	80
For more information.....	80
Property domains.....	81
The property bag .....	81
What the property bag is.....	81
Implementation.....	81
Property bag overflow .....	82
For more information.....	82
Repositories .....	82
What a repository is.....	82
Object type tables .....	83
Single-valued property tables .....	83
Repeating property tables .....	83
How standard subtype instances are stored.....	84
How lightweight subtype instances are stored.....	85
Materialization and lightweight SysObjects .....	86
Example of materialized and unmaterialized storage .....	86
The location and extents of object type tables.....	88
Object type index tables .....	89
Content storage areas.....	89
For more information.....	90
Registered tables .....	90
What registered tables are .....	90
For more information.....	90
Documentum views.....	91
The data dictionary.....	91
What the data dictionary is.....	91
Usage .....	91
Localization support.....	92
Modifying the data dictionary .....	92
Publishing the data dictionary .....	92
For more information.....	93
What the data dictionary can contain .....	93
Constraints .....	93
Unique key .....	94
Primary key .....	95
Foreign key .....	95
Not null .....	96
Check .....	96
Default lifecycles for object types .....	97
Component specifications .....	97

	Default values for properties .....	98
	Localized text .....	98
	Value assistance.....	98
	Mapping information .....	98
	The data dictionary and lifecycle states .....	98
	Retrieving data dictionary information .....	99
	Using DQL.....	99
	Using the DFC.....	100
	For more information.....	100
<b>Chapter 5</b>	<b>Object Type and Instance Manipulations and Customizations .....</b>	<b>101</b>
	Object type manipulations .....	101
	Creating new object types .....	101
	Altering object types.....	102
	Dropping object types.....	102
	For more information.....	102
	Object instance manipulations .....	103
	Object creation .....	103
	Object modification .....	103
	Object destruction.....	104
	Permissions and constraints .....	104
	Effects of the operation .....	104
	For more information.....	105
	Changing an object's object type .....	105
	Constraints on the operation .....	106
	Example .....	106
	For more information.....	106
	Business object framework .....	107
	Overview .....	107
	What the business object framework is.....	107
	Benefits.....	107
	What a BOF module is .....	107
	What comprises a module .....	108
	Module packaging and deployment.....	108
	Module interfaces on client machines.....	109
	Dynamic delivery and local caching.....	110
	BOF development mode.....	110
	Service-based objects .....	110
	What a service-based object is.....	110
	Implementation.....	111
	Type-based objects.....	111
	What a type-based object is .....	111
	Implementation.....	111
	Aspects.....	111
	What an aspect is.....	111
	Aspect properties .....	112
	Implementation of aspect properties .....	112
	Default aspects .....	113
	Simple modules.....	113
	What a simple module is.....	113
	Implementation.....	113
	For more information.....	114
<b>Chapter 6</b>	<b>Security Services .....</b>	<b>115</b>
	Security overview.....	115
	Standard security features.....	116

Summary of standard features.....	116
For more information.....	117
Trusted Content Services security features.....	118
Security features available with Trusted Content Services license.....	118
For more information.....	119
Repository security .....	119
Users and groups .....	120
Users .....	120
What a repository user is.....	120
Repository implementation.....	120
Local and global users.....	121
For more information.....	121
Groups .....	121
Standard groups.....	122
Role groups.....	122
Module role groups .....	122
Privileged group .....	122
Domain groups .....	123
How role and domain groups are used .....	123
What a dynamic group is .....	123
Mixing dynamic and non-dynamic groups in group memberships.....	124
Local and global groups.....	124
For more information.....	124
User authentication .....	124
What user authentication is .....	124
When authentication occurs .....	124
Authentication implementations.....	125
For more information.....	125
Password encryption .....	126
What password encryption is .....	126
How password encryption works .....	126
For more information.....	126
Application-level control of SysObjects .....	127
What application-level control of SysObjects is .....	127
Implementation.....	127
User privileges .....	128
Basic user privileges .....	128
Extended user privileges.....	129
For more information.....	129
Object-level permissions .....	129
What object-level permissions are.....	130
Base object-level permissions .....	130
Extended object-level permissions .....	131
Default permissions .....	132
For more information.....	132
Table permits .....	132
Folder security .....	133
What folder security is .....	133
Implementation.....	133
For more information.....	134
ACLs .....	134
What an ACL is .....	134
Implementation overview .....	134

ACL entries.....	135
Categories of ACLs.....	135
Template ACLs.....	136
For more information.....	136
Auditing and tracing .....	136
Auditing .....	136
What auditing is.....	136
What is audited .....	137
Recording audited events.....	137
Requesting auditing.....	137
For more information.....	138
Tracing .....	138
What tracing is .....	138
Tracing options.....	138
For more information.....	139
Signature requirement support.....	139
Electronic signatures.....	139
What an electronic signature is .....	140
Overview of Implementation.....	140
What addESignature does .....	141
The default signature page template and signature method .....	142
Default signature page template .....	142
Default signature creation method .....	142
How content is handled by default .....	143
Audit trail entries .....	143
What you can customize .....	144
Signature verification.....	145
General usage notes.....	145
For more information.....	145
Digital signatures .....	146
What a digital signature is.....	146
Implementation overview .....	146
For more information.....	147
Signoff method usage .....	147
What a simple sign-off is.....	147
Implementation overview .....	147
For more information.....	148
Privileged DFC.....	148
Privileged DFC registrations .....	149
How Content Server recognizes a privileged DFC instance.....	150
Using approved DFC instances only .....	150
For more information.....	150
Encrypted file store storage areas.....	150
What encrypted file store storage areas are.....	151
Implementation overview .....	151
For more information.....	151
Digital shredding .....	152
What digital shredding is.....	152
Implementation overview .....	152
<b>Chapter 7 Content Management Services .....</b>	<b>153</b>
Introducing SysObjects .....	153
Documents.....	154
What a simple document is .....	154
What a virtual document is .....	154
For more information.....	154

Document content .....	155
What a content object is.....	155
Implementation.....	155
Primary content and renditions .....	155
What primary content is.....	156
What a rendition is .....	156
Page numbers.....	156
Connecting source documents and renditions .....	156
Translations .....	157
For more information.....	157
Versioning.....	157
What versioning is.....	157
Version labels .....	158
r_version_label attribute .....	158
Numeric version labels .....	158
Symbolic version labels.....	158
The CURRENT label.....	159
Maintaining uniqueness.....	159
Version trees .....	160
Branching .....	160
Removing versions .....	161
Changeable versions.....	162
For more information.....	163
Immutability .....	163
Effects of a checkin or branch method .....	163
Effects of a freeze method .....	163
Effects of a retention policy .....	164
Attributes that remain changeable .....	164
For more information.....	165
Concurrent access control .....	165
Database-level locking .....	166
Repository-level locking.....	166
Optimistic locking .....	167
For more information.....	167
Document retention and deletion.....	167
Retention policies .....	168
Storage-based retention periods.....	169
Behavior if both a retention policy and storage-based retention apply .....	169
Deleting documents under retention.....	170
Privileged deletions .....	170
Forced deletions .....	171
Deleting old versions and unneeded renditions .....	171
Retention in distributed environments .....	171
For more information.....	172
Documents and lifecycles .....	172
Documents and full-text indexing.....	172
SysObject creation .....	173
Who owns the new object .....	173
Where the object is stored in the repository .....	173
Adding content .....	174
Renditions.....	174
Macintosh files .....	174
Where the content is stored .....	175
Content assignment policies .....	175
Default storage allocation.....	175

Explicitly assigning a storage area .....	176
Setting content properties and metadata for content-addressed storage.....	176
SysObjects and ACLs .....	178
For more information.....	178
Modifying SysObjects .....	179
Who can modify a document.....	179
Getting a document from the repository.....	179
Modifying single-valued attributes .....	180
Modifying repeating attributes .....	180
Performance tip for repeating attributes .....	181
Adding additional content .....	181
Adding additional primary content .....	182
Replacing an existing content file.....	182
Removing content from a document .....	182
Sharing a content file .....	183
Writing changes to the repository .....	183
Checkin and checkinEx methods .....	183
Save and SaveLock methods.....	183
For more information.....	184
Managing permissions .....	184
The default ACLs .....	184
Template ACLs.....	185
Assigning ACLs .....	185
Assigning a default ACL.....	185
Assigning an existing non-default ACL.....	186
Generating custom ACLs .....	186
Granting permissions to a new object without assigning an ACL .....	186
Modifying the ACL assigned to a new object.....	187
Using grantPermit when no default ACL is assigned .....	187
Modifying the current, saved ACL.....	187
Rooms and ACL assignments .....	187
Removing permissions.....	188
Removing permissions to a single document .....	189
Removing permissions to all documents .....	189
Replacing an ACL.....	189
For more information.....	189
Managing content across repositories.....	189
Architecture .....	190
For more information.....	190
Relationships between objects.....	191
What a relationship is .....	191
System-defined relationships.....	191
User-defined relationships .....	191
For more information.....	192
Managing translations .....	192
Translation relationships .....	192
For more information.....	193
Annotation relationships.....	193
Creating annotations.....	194
The annotation file.....	194
Attaching the annotation to the document.....	194
Detaching annotations from a document .....	195
Deleting annotations from the repository .....	195
Object operations and annotations .....	195
Save, Check In, and Saveasnew .....	195

	Destroy .....	195
	Object replication .....	196
	For more information.....	196
<b>Chapter 8</b>	<b>Virtual Documents</b> .....	197
	Introducing virtual documents .....	197
	What is a virtual document?.....	198
	Use of virtual documents .....	199
	Implementation.....	199
	Connecting the virtual document and its components .....	199
	r_is_virtual_doc property.....	200
	Versioning.....	200
	Deleting virtual documents and components.....	200
	Referential integrity, freezing, and component deletions.....	201
	Assembling the virtual document .....	201
	What conditional assembly is .....	201
	Snapshots .....	201
	Virtual documents and content files.....	202
	XML support.....	202
	Virtual documents and retention policies .....	203
	For more information.....	203
	Virtual document assembly and binding .....	204
	Early and late binding.....	204
	Binding rules and assembly logic.....	204
	Defining component assembly behavior.....	205
	use_node_ver_label .....	206
	follow_assembly.....	207
	Defining copy behavior.....	207
	Creating virtual documents.....	208
	Modifying virtual documents .....	209
	Required permissions .....	209
	Adding components .....	209
	Removing components .....	210
	Changing the component order .....	210
	Modifying assembly behavior .....	210
	Modifying copy behavior .....	210
	For more information.....	210
	Assembling a virtual document.....	211
	Basic procedure .....	211
	How the SELECT statement is processed.....	212
	For more information.....	213
	Snapshots .....	213
	How a snapshot is created.....	214
	Modifying snapshots .....	214
	Adding new assembly objects.....	214
	Deleting an assembly object .....	215
	Changing an assembly object.....	215
	Deleting a snapshot .....	215
	Frozen virtual documents and snapshots .....	216
	What freezing does.....	216
	Unfreezing a document.....	217
	Obtaining information about virtual documents .....	217
	Querying virtual documents .....	217
	Obtaining a path to a particular component.....	218
	The path_name property.....	218

	Using DFC .....	219
	For more information.....	219
<b>Chapter 9</b>	<b>Renditions</b> .....	221
	Introducing renditions .....	221
	What a rendition is .....	221
	Converter support .....	222
	Content Transformation Services .....	222
	Rendition formats.....	222
	Rendition characteristics .....	223
	For more information.....	223
	Automatically generated renditions.....	223
	User-generated renditions.....	224
	The keep argument.....	224
	Supported conversions on Windows platforms.....	225
	Supported conversions on UNIX platforms .....	225
	PBM image converters .....	226
	Miscellaneous converters .....	227
	Implementing an alternate converter.....	228
<b>Chapter 10</b>	<b>Workflows</b> .....	231
	Introducing workflows .....	231
	What a workflow is.....	232
	Implementation.....	232
	Template workflows .....	233
	Process Builder and Workflow Manager .....	234
	For more information.....	235
	Workflow definitions .....	235
	Process definitions.....	235
	Activity types in a process definition .....	235
	Multiple use of activities in process definitions.....	236
	How activities are referenced in workflows .....	236
	Links .....	237
	Activity definitions.....	237
	Manual and automatic activities .....	237
	Manual activities .....	238
	Automatic activities.....	238
	For more information.....	238
	Activity priorities .....	238
	Use of the priority defined in the process definition .....	239
	Use of the work queue priority values.....	239
	For more information.....	239
	Delegation and extension .....	240
	Delegation .....	240
	Extension.....	240
	Repeatable activities .....	241
	Performer choices .....	241
	Performer categories.....	241
	Categories for manual activities.....	246
	Categories for automatic activities .....	247
	Defining the actual performer.....	247
	At workflow initiation .....	247
	At activity initiation.....	248
	At the completion of a previous activity .....	248

Choosing the same performer set for multiple manual activities .....	248
Using aliases as performer names .....	248
Usage .....	248
Constraints on aliases as performer names .....	249
Alias resolution in workflows .....	249
For more information .....	250
Task subjects .....	250
Starting conditions .....	251
Port and package definitions .....	252
Port definitions .....	252
Package definitions .....	253
Empty packages .....	253
Scope of a package definition .....	253
Required Skill Levels for Packages .....	254
Package compatibility .....	254
For more information .....	255
Transition behavior .....	255
Number of completed tasks as transition trigger .....	255
Transition types .....	256
Limiting output choices in manual transitions .....	257
Setting preferences for output port use in manual transitions .....	257
Route cases for automatic transitions .....	258
Implementation without an XPath expression .....	258
Implementation with an XPath expression .....	259
Compatibility of the implementations .....	259
How the associated ports are recorded .....	260
For more information .....	260
Warning and suspend timers .....	260
Warning timers .....	260
Suspend timers .....	261
For more information .....	262
Package control .....	262
What package control is .....	262
Implementation .....	262
For more information .....	263
Process and activity definition states .....	263
Validation and installation .....	263
Validating process and activity definitions .....	263
What validation checks .....	264
Validating port connectability .....	264
For more information .....	265
Installing process and activity definitions .....	265
Implementation .....	265
For more information .....	265
Architecture of workflow execution .....	266
Workflow objects .....	266
Activity instances .....	266
For more information .....	266
Work item objects .....	267
Work items and queue items .....	267
How manual activity work items are handled .....	267
Resetting priority values .....	268
Completing work items .....	268
Signing off manual work items .....	269
For more information .....	269
Package objects .....	269
Package notes .....	270

Attachments.....	270
The workflow supervisor .....	271
The workflow agent.....	271
What the workflow agent is.....	271
Implementation.....	272
For more information.....	272
The enable_workitem_mgmt key .....	272
Instance states .....	273
Workflow states.....	273
Activity instance states .....	274
Work item states.....	275
For more information.....	276
Starting a workflow .....	276
How execution proceeds .....	276
The workflow starts.....	279
Activity execution starts.....	279
Evaluating the starting condition .....	280
Package consolidation.....	280
Resolving performers and generating work items .....	281
Manual activities .....	281
Automatic activities.....	281
Resolving aliases .....	282
For more information.....	282
Executing automatic activities .....	282
Assigning an activity for execution .....	282
Executing an activity's program.....	283
For more information.....	284
Evaluating the activity's completion.....	284
When the activity is complete.....	287
Port selection behavior for each transition type.....	287
For more information.....	287
How activities accept packages.....	288
How activity timers work.....	289
Pre-timer instantiation .....	290
Post-timer instantiation.....	290
Suspend timer instantiation.....	290
Activating pre- and post-timers .....	291
Activating suspend timers.....	291
Compatibility with pre-5.3 timers .....	291
For more information.....	291
User time and cost reporting .....	292
Reporting on completed workflows .....	292
The dm_WFReporting job .....	292
For more information.....	292
Changing workflow, activity instance, and work item states .....	293
Halting a workflow.....	293
Activity instances in halted workflows.....	293
Halting an activity instance .....	293
Resuming a halted workflow or activity .....	294
Restarting a halted workflow or failed activity.....	294
Aborting a workflow .....	294
Pausing and resuming work items .....	295
Modifying a workflow definition.....	295
Changing process definitions .....	295
Overwriting a process definition.....	295

Versioning process definitions .....	296
Reinstalling after making changes.....	296
Changing activity definitions .....	297
Overwriting an activity definition.....	297
Adding ports and package definitions .....	297
Removing ports and package information .....	297
Saving the changes .....	298
Destroying process and activity definitions .....	298
Distributed workflow .....	298
Distributed notification.....	299
Remote object routing.....	299
<b>Chapter 11 Lifecycles .....</b>	<b>301</b>
Introducing lifecycles.....	301
What a lifecycle is.....	301
Normal and exception states .....	302
Types of objects that may be attached to lifecycles.....	303
Entry criteria, actions on entry, and post-entry actions .....	304
For more information.....	304
Default lifecycles for object types.....	305
Lifecycle definitions.....	305
Repository storage.....	305
Lifecycle definition states .....	305
Lifecycle definition validation .....	306
Installation .....	306
For more information.....	306
How lifecycles work .....	307
General overview .....	307
Attaching objects.....	308
Movement between states .....	308
Promotions .....	308
Batch promotions .....	309
Demotions .....	309
Suspensions .....	309
Resumptions .....	310
Scheduled transitions.....	311
Internal supporting methods.....	311
How state changes work .....	312
For more information.....	313
Object permissions and lifecycles.....	314
Integrating lifecycles and applications .....	314
Lifecycles, alias sets, and aliases .....	314
State extensions .....	315
State types.....	315
For more information.....	316
Designing a Lifecycle .....	316
Required design decisions.....	316
For more information.....	317
Lifecycle state definitions .....	317
Attachability .....	318
Return to base state setting.....	318
Demoting from a state.....	320
Allowing scheduled transitions .....	320
Entry criteria definitions .....	320
Java programs as entry criteria .....	321

Docbasic programs as entry criteria .....	322
Boolean expressions as entry criteria .....	323
Actions on entry definitions .....	323
Using system-defined actions on entry .....	324
Java programs as actions on entry .....	324
Docbasic programs as actions on entry .....	325
Post-entry action definitions .....	326
Java programs as post-entry actions .....	326
Docbasic programs as post-entry actions .....	327
Sample implementations of Java interfaces .....	328
Including electronic signature requirements .....	329
Using aliases in actions .....	329
For more information .....	330
Lifecycle creation .....	330
Basic procedure .....	331
Using state extensions .....	331
Extension objects .....	332
Testing and debugging a lifecycle .....	332
Testing a lifecycle .....	332
Debugging a lifecycle .....	333
Java-based lifecycles .....	333
Docbasic-based lifecycles .....	333
Log file location .....	333
For more information .....	333
Custom validation programs .....	334
Overview .....	334
For more information .....	334
Modifying lifecycles .....	335
Possible changes .....	335
Uninstall and installation notifications .....	335
Obtaining information about lifecycles .....	336
Lifecycle information .....	336
Object information .....	336
For more information .....	337
Deleting a lifecycle .....	337
<b>Chapter 12 Tasks, Events, and Inboxes .....</b>	<b>339</b>
Introducing tasks and events .....	339
What a task is .....	339
What an event is .....	340
Repository implementation .....	340
Accessing tasks and events .....	340
For more information .....	340
Introducing Inboxes .....	341
What an inbox is .....	341
Accessing an Inbox .....	341
dmi_queue_item objects .....	341
Purpose .....	342
Historical record .....	342
For more information .....	342
Obtaining Inbox content .....	342
GET_INBOX administration method .....	343
getEvents method .....	343
The dm_queue view .....	343
For more information .....	343

Manual queuing and dequeuing .....	343
Queuing items.....	344
Dequeuing an inbox item .....	344
Registering and unregistering for event notifications .....	344
Registering for events .....	345
Removing a registration.....	345
For more information.....	345
Querying for registration information .....	346
<b>Appendix A Aliases</b> .....	347
Introducing aliases .....	347
Internal implementation .....	348
Defining aliases .....	348
Alias scopes .....	349
Workflow alias scopes.....	349
Non-workflow alias scopes .....	350
Determining the lifecycle scope for SysObjects .....	351
Resolving aliases in SysObjects .....	351
Resolving aliases in template ACLs.....	352
Resolving aliases in Link and Unlink methods.....	352
Resolving aliases in workflows.....	353
Resolving aliases during workflow startup.....	353
Resolving aliases during activity startup .....	353
The default resolution algorithm .....	354
The package resolution algorithm.....	354
The user resolution algorithm.....	354
When a match is found .....	355
Resolution errors .....	355
<b>Appendix B Internationalization Summary</b> .....	357
What Content Server internationalization is .....	357
Content files and metadata .....	358
Content files.....	358
Metadata.....	358
Client communications with Content Server.....	358
Constraints .....	359
For more information.....	359
Configuration requirements for internationalization .....	359
Values set during installation .....	359
The server config object.....	360
For more information.....	360
Values set during sessions .....	360
The client config object.....	360
The session config object .....	361
How values are set .....	361
Where ASCII must be used .....	362
User names, email addresses, and group names.....	362
Lifecycles .....	363
Docbasic .....	363
Federations .....	363
Object replication .....	364

Other cross-repository operations ..... 364

## List of Figures

Figure 1.	Example of database table storage for standard subtypes .....	85
Figure 2.	Database storage of unmaterialized lightweight object instances.....	87
Figure 3.	Database storage of materialized lightweight object instances.....	88
Figure 4.	Sample hierarchy.....	106
Figure 5.	Example of BOF module storage in a repository .....	109
Figure 6.	A version tree with branches .....	160
Figure 7.	Before and after pruning .....	162
Figure 8.	Containment object defines virtual document relationship .....	198
Figure 9.	Assembly object defines assembly relationship.....	202
Figure 10.	Assembly decision tree .....	205
Figure 11.	A sample virtual document showing node bindings .....	206
Figure 12.	Components of a workflow .....	233
Figure 13.	Workflow state diagram.....	273
Figure 14.	Activity instance state diagram.....	274
Figure 15.	Work item states .....	275
Figure 16.	Execution flow in a workflow .....	278
Figure 17.	Behavior if r_complete_witem equals r_total_workitem .....	285
Figure 18.	Behavior if r_complete_witem < r_total_workitem .....	286
Figure 19.	Changes to a package during port transition .....	288
Figure 20.	Package arrival.....	289
Figure 21.	A lifecycle definition with exception states .....	302
Figure 22.	Simple lifecycle definition .....	308

---

## List of Tables

Table 1.	Repeating properties table.....	83
Table 2.	Row determination in a repeating properties table.....	84
Table 3.	Standard security features supported by Content Server .....	116
Table 4.	Security features that require a Trusted Content Services license.....	118
Table 5.	Basic user privilege levels.....	128
Table 6.	Extended user privileges .....	129
Table 7.	Base object-level permissions.....	130
Table 8.	Extended object-level permissions .....	131
Table 9.	Table permits .....	132
Table 10.	Generic string property use for dm_addesignature events.....	144
Table 11.	Supported input and output formats for automatic conversion.....	225
Table 12.	Acceptable input formats for PBMPLUS converters .....	226
Table 13.	Output formats of the PBMPLUS converters.....	227
Table 14.	Acceptable input formats for UNIX conversion utilities.....	227
Table 15.	Output formats of UNIX conversion utilities .....	228
Table 16.	Performer categories for activities.....	242
Table 17.	Mode parameter values.....	284
Table 18.	Lifecycle methods.....	312



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
vertical line	A choice between two or more options

## Revision history

The following changes have been made to this document.

**Revision history**

---

<b>Revision date</b>	<b>Description</b>
July 2007	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 25](#)
- [Standard Content Server services, page 25](#)
- [Additional Content Server offerings, page 31](#)
- [EMC Documentum products requiring activation on Content Server, page 34](#)
- [Internationalization, page 36](#)
- [Communicating with Content Server, page 37](#)

### Content Server's role in the product suite

Content Server is the foundation of EMC 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. In addition, you can apply one or more additional licenses to enable additional content management features.

### Standard Content Server services

This section briefly introduces the Content Server features that are part of the standard, core product.

## Content management services

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

### 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.

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.)

### 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.

## Data dictionary

The data dictionary stores information in the repository about object types and properties. 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.

## 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.

## Search services

Content Server supports a third-party search engine that provides comprehensive indexing and search capabilities. By default, all property values and indexable content are indexed, allowing users to search for documents or other objects based on property values or the content.

## For more information

- Refer to the following topics in this manual:
  - [Chapter 4, The Data Model](#), which provides a detailed description of the repository data model.
  - [Versioning, page 157](#), which describes how versions are handled by Content Server.
  - [The data dictionary, page 91](#), which 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 Administration Guide*.
  - [Chapter 8, Virtual Documents](#), which has complete information about the implementation and behavior of virtual documents.
- The *Content Server DQL Reference Manual* contains reference information, usage notes, and query examples for DQL.
- The *Content Server Administration Guide* describes the supported types of content storage areas and how to administer and manage them.
- *Administering Documentum Media Transformation Services* has full information about EMC Documentum Media Transformation Services.

## 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.

Workflows are created and managed using EMC Documentum Process Builder or Workflow Manager. Workflow Manager is the standard, basic user interface for creating and managing workflows. Process Builder is a separately licensed product that provides additional, sophisticated workflow features beyond the basic features provided by Workflow Manager.

## Lifecycles

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 more information

- Refer to the following topics in this manual:
  - [Chapter 10, Workflows](#), which describes basic workflow functionality and introduces the additional features provided by Process Builder.
  - [Chapter 11, Lifecycles](#), which describes how lifecycles are implemented.
- The *Documentum Process Builder User Guide* describes the EMC Documentum Process Builder features in detail and describes how to use Process Builder.
- The *Documentum Object Reference Manual* describes the object types that support workflows and 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.

## Repository security

Repositories are protected on two levels:

- At the repository level

Content Server supports a wide range of user authentication possibilities. When users attempt to connect to a repository, Content Server authenticates that the user is a valid user in that repository. If the user is not a valid or active user in the repository, the connection is not allowed.

- At the object level

A repository uses a security model based on Access Control Lists (ACLs) by default to protect repository objects.

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. There are seven levels of base object-level permissions and five extended object-level permissions

Content Server also provides five levels of user privileges, three extended user privileges, folder security, privileged roles, 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.

## For more information

- [Chapter 6, Security Services](#) provides a complete overview of all security options.
- [Assigning ACLs, page 185](#), describes how to assign an ACL to an object.

- Refer to the *Content Server Administration Guide* for the following topics:
  - User authentication options and how to implement each
  - ACLs , the entries you can use in an ACLs, and how to create ACLs
  - Auditing and how to start, stop, and administer auditing
  - Tracing facilities supported by Content Server and DFC

## 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*.

## Additional Content Server offerings

This section briefly introduces the add-on functionality supported in Content Server with the appropriate licenses.

### Trusted Content Services

This add-on is installed through an additional license key specified during Content Server installation.

### What Trusted Content Services is

The Trusted Content Services (TCS) license adds additional security features to Content Server. The features supported by this license are:

- Digital shredding of content files
- Strong electronic signatures
- Ability to encrypt content in file store storage areas
- Ability to create enhanced ACL entries that allow you to define more complex access rules and restrictions

## For more information

- Refer to the following topics in this manual:
  - [Trusted Content Services security features, page 118](#), which describes the security options available with TCS in more detail.
  - [Encrypted file store storage areas, page 150](#), which describes encrypted storage areas in detail.
  - [Digital shredding, page 152](#), which provides a brief description of this feature.
  - [Signature requirement support, page 139](#), which describes how electronic signatures supported by addESignature work.
  - [ACLs, page 134](#), provides more information about the permit types that you can define with a Trusted Content Services license.
- The *Content Server Administration Guide*, describes ACLs and how to add, modify, and remove entries.

## Content Services for EMC Centera

This add-on is installed through an additional license key specified during Content Server installation.

### What Content Services for EMC Centera is

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

## For more information

- Refer to the following topics in this manual:
  - [Document retention and deletion, page 167](#), which describes the various ways to implement document retention, including the storage-based retention available with the CSEC license, and how that retention behaves.  
**Note:** It is possible to apply retention to content without a CSEC license if you have installed Content Server with a Retention Policy Services license. A CSEC license is required only if you want to also store the content in a content-addressed storage area.
  - [Setting content properties and metadata for content-addressed storage, page 176](#), describes how retention settings for content-addressed storage areas are implemented internally
- Refer to the following topics in the *Content Server Administration Guide*:
  - Content-addressed storage areas and the configuration options available for those storage areas
  - How to set up and configure a content-addressed storage area

## Content Storage Services

This add-on is installed through an additional license key specified during Content Server installation.

### What Content Storage Services is

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

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

## For more information

The *Content Server Administration Guide* has more information about:

- Content compression
- Content duplication checking and prevention
- Content assignment policies, the requirements for using them, and how to create them
- Storage migration policies
- Administration procedures for content assignment policies

## EMC Documentum products requiring activation on Content Server

The products described in this section are sold separately, with separate installers, but also require a license to be entered when installing Content Server, to activate Content Server support for the product.

## Retention Policy Services

This add-on is installed through an additional license key specified during Content Server installation. Additionally, you must install the Retention Policy Services DAR file.

### What Retention Policy Services is

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

The product allows you to manage a SysObject's retention in the repository through a retention policy, a formal, defined set of phases, with a formal disposition phase at the end. The product is accessed through Documentum Administrator. The Retention Policy Services product may not be customized. You must use the product as it is delivered.

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

## For more information

- [Document retention and deletion, page 167](#), describes the various ways to implement document retention, including retention policies, and how those policies affect behaviors.
- [Virtual documents and retention policies, page 203](#), describes how applying a retention policy to a virtual document affects that document,
- The *Retention Policy Services User Guide* contains complete information about using Retention Policy Services Administrator.

## EMC Documentum Collaborative Services

EMC Documentum Collaborative Services (formerly called Collaborative Edition) are activated by a license key specified during the Content Server and Webtop installation procedures.

### What EMC Documentum Collaborative Services are

Collaborative Services allow teams to work collaboratively and securely from any EMC Documentum WDK-based client. The Services support a wide range of collaborative features, such as:

- Rooms, which 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, which are online comment threads that enable informal or spontaneous collaboration on objects.
- Contextual folders, which 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, which 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.

## For more information

- 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.

# Internationalization

Internationalization is the term that refers to how Content Server handles communications and data transfer between itself and various client applications that may or may not be using the same code page as Content Server.

## Overview

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 a national character set 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 ability to:

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

## For more information

- For more information about Unicode, UTF-8, and national character sets, refer to the Unicode Consortium's Web site at <http://www.unicode.org/>.
- [Appendix B, Internationalization Summary](#), contains a summary of Content Server's internationalization requirements.

# 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 API is the DFC, the Documentum Foundation Classes. This API is a set of Java classes and interfaces that provide full access to Content Server features. 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 DAR files) that you can deploy on the Web or desktop. You can also customize components of the Documentum client applications.

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

## Interactive utilities

To interactively communicate with Content Server, Documentum provides Documentum Administrator and the IDQL utility.

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.

IDQL is interactive utilities that let you execute DQL statements directly. The utility is primarily useful as a testing arena for statements that you may want to put in an application. It is also useful when you want to execute a quick ad hoc query against the repository.

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

# Session and Transaction Management

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

- [What a session is, page 39](#)
- [Multiple sessions, page 40](#)
- [DFC Implementation, page 40](#)
- [Inactive repository sessions, page 44](#)
- [Restricted sessions, page 44](#)
- [Connection brokers, page 45](#)
- [Native and secure connections, page 46](#)
- [Connection pooling, page 47](#)
- [Login tickets, page 48](#)
- [Application access control tokens, page 52](#)
- [The login ticket key, page 56](#)
- [Trusting and trusted repositories, page 57](#)
- [Concurrent sessions , page 58](#)
- [Transaction management, page 58](#)

## What a session is

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

## Multiple sessions

Users or applications can have multiple sessions open at the same time with one or more repositories. The number of sessions that can be established for a given user or application is controlled by the `max_session_count` entry in the `dfc.properties` file. The value of this entry is set to 10 by default and can be reset.

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

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

## DFC Implementation

This section describes how sessions are implemented within DFC.

### Session objects

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

### Obtaining a session

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

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

## Session identifiers

Each session has a session identifier in the format  $S_n$  where  $n$  is an integer equal to or greater than zero. This identifier is used in trace file entries, to identify the session to which a particular entry applies. Session identifiers not used or accepted in DFC methods calls.

## Shared and private sessions

Repository sessions are either shared or private.

### Shared sessions

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

Shared sessions are obtained by using a `IDfSessionManager.getSession` method to obtain a session.

### Private sessions

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

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

## Explicit and implicit sessions

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

Because some repositories have more than one Content Server and the servers are often running on different host machines, DFC methods let you be as specific as you like when

requesting the connection. You can let the system choose which server to use or you can identify a specific server by name or host machine or both.

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

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

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

## Session configuration

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

### The `dfc.properties` file

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

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

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

## The runtime configuration objects

There are three runtime configuration objects that govern a session:

- client config
- session config
- connection config

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

The configuration values are applicable to all sessions started through that DFC instance.

The session config and the connection config objects represent individual sessions with a repository. Each session has one session config object and one connection config object.

## Closing repository sessions

From an end user's viewpoint, a user or application's repository session is terminated when the user or application disconnects or releases the session or when another user assumes ownership of the session. Internally, a released or disconnected session is held in a connection pool, to be reused.

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

## Object state when session closes

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

## Terminating a session manager

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

- All sessions are released or disconnected
- All `beginClientControl` methods have a matching `endClientControl` executed

- All transactions opened with a `beginTransaction` have been committed or aborted.

## For more information

- The *System Administrator's Guide* has instructions for setting the connection attempt interval and the number of retries.
- The *Documentum Object Reference Manual* lists the properties in the configuration objects.
- [Connection pooling, page 47](#), describes how connection pooling is implemented.

## Inactive repository sessions

Inactive repository sessions are sessions in which the server connection has timed out but the client application has not specifically disconnected from the server. If the client application sends a request to Content Server, the 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 was started with a single-use login ticket and that session times out, the session cannot be automatically restarted by default because the login ticket cannot be 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

A restricted session is a repository session opened for a user who connects with an expired operating system password. The only operation allowed in a restricted session is changing the user's password. Applications can determine whether a session they begin is a restricted session by examining the value of the computed property `_is_restricted_session`. This property is T (TRUE) if the session is a restricted session.

---

# Connection brokers

This section provides a brief introduction to connection brokers, an integral part of a Content Server installation.

## What the connection broker is

A connection broker is a name server for the Content Server. It provides connection information for Content Servers and ACS servers and information about the proximity of network locations.

## Role of the connection broker

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

Connection brokers do not request information from 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 `dfc.properties` file used by the client. You can define primary and backup connection brokers in the file. Doing so ensures that users will rarely encounter a situation in which they cannot obtain a connection to a repository.

It is also possible for an application to set the connection broker programmatically. Setting them directly allows the application to use a connection broker that may not be included in the connection brokers specified in the `dfc.properties` file. The application must set the connection broker information before requesting a connection to a repository.

## For more information

- The *Content Server Administration Guide* provides complete information about how servers, clients, and connection brokers interact.
- The *DFC Development Guide* contains full information about setting a connection broker programmatically.

## Native and secure connections

Content Servers, connection brokers, and client applications can communicate using native (non-secure) or secure connections.

### What native and secure connections are

A native connection is a non-secure connection. A secure connection is a connection that uses the secure socket layer (SSL) protocol.

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

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

### Using secure connections

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

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

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

## For more information

- The *Content Server Administration Guide* has instructions on resetting the connection default for Content Server and how clients request a secure connection.
- The interaction between the Content Server's setting and the client's request is described in the Javadocs, in the description of the `IDfLoginInfo.setSecurityMode` method.

## Connection pooling

Connection pooling is an optional feature providing performance benefits to users when opening sessions.

### What connection pooling is

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

### How connection pooling works

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

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

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

## Simulating connection pooling in an application

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

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

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

## For more information

- The *Content Server Administration Guide*, contains instructions about enabling and configuring connection pooling.
- For details about using an `assume` method, refer to the Javadocs.

## Login tickets

This section describes login tickets and their implementation and use.

### What a login ticket is

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 current or a different repository.

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

## Ticket generation

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

## 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. The signature is generated using the login ticket key installed in the repository.

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

## 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

## Login ticket validity

This section discusses setting expiration times for a login ticket and revoking login tickets.

### Ticket expiration

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

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

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

### 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` property of the repository's `docbase` config object.

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

- [The login ticket key, page 56](#), describes the login ticket key.
- [Trusting and trusted repositories, page 57](#), describes how trusted repositories are defined and identified.
- The *Content Server Administration Guide*, has information about:
  - Setting a repository's trust mode
  - Configuring the default and maximum validity periods in a repository
  - Restricting Superuser use of global login tickets

## Application access control tokens

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

### What application access control tokens are

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.

### Benefits of use

If you configure a Content 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.

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 DFC or later is capable of appending a token or machine identifier to a connection request. Configuring the DFC to append a token is optional.

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

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

If the receiving server requires a token, the server decodes the token and determines whether the constraints are satisfied. 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, DFC has the

IDfClient.getApplicationTokenDiagnostics method, which returns the encoded values in readable text format.) The signature is generated using the repository's login ticket key.

## 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.

## Token generation

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

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 DFC to ensure that methods or applications written prior to version 5.3 can connect to servers that now require a token.

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

## 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 by the DFC on the host on which the method is executed. Token retrieval allows the DFC to append a token retrieved from storage to the connection request. The token must be generated by the `dmtkgen` utility and must be a valid token for the connection request.

## For more information

- [The login ticket key, page 56](#) describes the login ticket key.
- [Trusting and trusted repositories, page 57](#), describes how trust is determined between repositories.
- The *Content Server Administration Guide* has information about:
  - Enabling token use in a server
  - Configuring DFC to append a token or machine identifier to a connection request
  - Using `dmtkgen`
  - Implementing token use and enabling token retrieval

## The login ticket key

A login ticket key is installed automatically, when a repository is configured.

### What the login ticket key is

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` property of the `docbase` config object.

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

### Working with the key

Login ticket keys are used to generate the Content Server signatures that are part of a login ticket key or application access token. Consequently, 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.

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 more information

- The *Content Server Administration Guide* provides information about:
  - Executing the `EXPORT_TICKET_KEY` and `IMPORT_TICKET_KEY` methods
  - resetting a login ticket key

## Trusting and trusted repositories

In order to use global login tickets or application access tokens, the repositories whose servers generate the tickets or application access tokens or receive the tickets or tokens must be appropriately configured as trusted or trusting repositories. This section discusses how trusted and trusting repositories are configured.

### What trusting and trusted repositories are

A trusting repository is a repository that accepts login tickets or application access tokens or both that were generated by a Content Server from a different repository. A trusted repository is a repository that generates login tickets or application access tokens or both that are accepted by a different repository.

### How trusting and trusted repositories are configured

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` property 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 or application access token generated with an LTK that matches its LTK, regardless of the ticket or token's source repository. If the property is set to F, then the repository is running in non-trusting mode. A non-trusting repository accepts global login tickets or application access tokens generated with a matching LTK if they come from repositories specifically named as trusted repositories. The list of trusted repository names is recorded in a repository's `trusted_docbases` property 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 or tokens 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` property: RepositoryM and repositoryN. RepositoryK rejects login tickets or tokens from RepositoryA because RepositoryA is not in the list of trusted repositories. It accepts tickets or tokens from RepositoryM and RepositoryN because they are listed in the `trusted_docbases` property.

## Concurrent sessions

There are limits placed on the number of repository connections that each Content Server can handle simultaneously. This section discusses how such sessions are handled.

### What concurrent sessions are

Concurrent sessions are repository sessions that are open at the same time through one Content Server. The sessions can be for one user or multiple users.

### How concurrent sessions are handled

By default, a Content Server can have 100 connections open concurrently. The limit is configurable by setting the `concurrent_sessions` key in the `server.ini` file. You can edit this file using Documentum Administrator

Each connection to a Content Server, whether an explicit or implicit connection, counts as one connection. Content Server returns an error if the maximum number of sessions defined in the `concurrent_sessions` key is exceeded.

### For more information

- *Content Server Administration Guide*, provides instructions for setting `server.ini` file keys.

## Transaction management

This section describes transactions and how they are managed.

### What a transaction is

A transaction is one or more repository operations handled as an atomic unit. All operations in the transaction must succeed or none may succeed. A repository session

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

## Internal and explicit transactions

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 `beginTransaction` method. It is closed when the transaction is explicitly committed, to save the changes, or aborted, to close the transaction without saving the changes. An explicit transaction can include as many operations as desired. However, keep in mind that none of the changes made in an explicit transaction are committed until the transaction is explicitly committed. If an operation fails, the transaction is automatically aborted and all changes made prior to the failure are lost.

## Constraints on explicit transactions

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

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

Opening an explicit transaction starts the transaction only for the current repository. If you issue a method in the transaction that references a remote object, work performed in the remote repository by the method is not 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 perform any of the following methods that manage objects in a lifecycle: `attach`, `promote`, `demote`, `suspend`, and `resume`.
- You cannot issue a complete method for an activity if the activity is using XPath a route case condition to define the transition to the next activity.
- You cannot execute an `IDfSysObject.assemble` method that includes the `interruptFreq` argument.

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

If you want to use DFC methods in an explicit transaction, open the transaction with a DFC 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 property.

## 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 a lock method (in the IDfPersistentObject interface). A Superuser can lock any object with a database-level lock. Other users must have at least Version permission on an object to place a database lock on the object.

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

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

To test custom deadlock retry logic, Content Server provides an administration method called `SET_APIDEADLOCK`. This method plants a trigger on a particular operation. When the operation executes, the server simulates a deadlock, setting the `_isdeadlocked` computed property and rolling back any changes made prior to the method's execution. Using `SET_APIDEADLOCK` allows you to test an application's deadlock retry logic in a development environment.

## For more information

- The *Content Server DQL Reference Manual* describes the SET\_APIDEADLOCK method in detail.

## Caching

This chapter describes the caches maintained on a global and session level for sessions. It also describes the optional persistent client caching feature. The chapter includes the following topics:

- [Object type caching, page 63](#)
- [Repository session caches, page 64](#)
- [Persistent caching, page 65](#)
- [Persistent caching and multiple sessions for one user, page 68](#)
- [Currency checking , page 68](#)

## Object type caching

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

## For object types with names beginning with dm, dmr, and dmi

These object types are the types that are built-in in a Content Server installation. Their type definitions are relatively static. There are few changes that can be made to the definition of a built-in type. For these types, the mechanism is an internal process called Change Checker, that periodically checks all the object type definitions in the Content Server's global cache. If any are found to be out-of-date, the process flushes the cache and reloads the type definitions into the global cache. Changes to these types are not visible to existing sessions because the DFC caches are not updated when the change checker refreshes the global cache.

Stopping and restarting a session makes any changes in the global cache visible. If the session was a web-based client session, it is necessary to restart the web application server.

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

## For custom object types and types with names beginning with dmc

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

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

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

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

## For more information

- The *Content Server Administration Guide* has instructions for setting the database refresh interval for the server's global cache.

## Repository session caches

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

---

## Object cache

An in-memory object cache is maintained for each repository session for the duration of the repository session. If persistent caching is requested and enabled, the cache also contains a copy of every object fetched and persistently cached in previous sessions.

## Query cache

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 with a randomly generated extension on the client application host disk.

## Data dictionary caches

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

## Persistent caching

This section describes persistent caching, a feature supported by the DFC and Content Server, that provides performance benefits during session start up and when users or applications access cached objects and query results.

## What persistent caching is

Persistent caching is the ability to create, manage, and maintain persistent caches of query results, objects, and the type and data dictionary information associated with persistently cached objects. Persistent caches are maintained across client sessions for each user and repository combination. Persistent caches are implemented through DFC and supported by Content Server.

The ability to cache objects and query results is enabled by default for every repository and every client session. Documentum Webtop takes advantage of this feature.

Applications can take advantage of the feature through the DFC methods that support requests for persistent client caching.

Consistency between the repository and cached objects and query results is checked and maintained using a consistency check rule identified in the 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.

## Persistent object cache storage location

Persistently cached objects are written to a file on the client disk. For a desktop application, this is the user's local disk. For a web-based application, this is the web application server's disk. The file is written after defined intervals during a user's session and when the session terminates. The file is stored in the following directory:

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

*root* is the value of the `dfc.data.local_dir` key in the client's `dfc.properties` file.

The next time the user starts a session with the repository on the same machine, the file is loaded back into memory.

## Persistent object type and data dictionary storage location

For each persistently cached object, a persistent cache of its object type definition and data dictionary information is also maintained. These are stored in a file located in the following directory:

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

*root* is the value in the `dfc.data.local_dir` key in the client's `dfc.properties` file.

## Query cache storage location

The file that contains the cached query results is stored with a randomly generated extension on the client disk. For a desktop application, this is the user's local disk. For a web-based application, this is the web application server's disk. The files are 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 `dfc.data.local_dir` key in the client's `dfc.properties` file.

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.

## Using persistent client caching in an application

Some Documentum clients 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. At the client session level, this is controlled by the `dfc.cache.enable_persistence` key in `dfc.properties`.

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

You identify the data to cache in application methods that fetch the objects or execute the query.

- 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 methods that support persistent client caching support a variety of rule options through a method argument.

## For more information

- [Consistency checking, page 68](#), describes the consistency supported checking rule options and how they are defined and applied for cached data.
- [Consistency checking, page 68](#), describes how consistency checking is defined and implemented.

- The *Content Server Administration Guide* has more information about:
  - Defining persistent cache write intervals
  - Enabling or disabling persistent caching

## Persistent caching and multiple sessions for one user

In federated environments or any distributed environment that has multiple repositories, users can work in multiple repositories through one primary client session. For the purposes of persistent caching, each connection to a different repository is treated as a separate session. 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 DFC 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, DFC creates a query cache file specific to the queried repository and user.

## Currency checking

This section describes currency checking, the functionality that is used to ensure that objects in the persistent cache are not out of date.

## 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 DFC that tells DFC 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. DFC determines whether a consistency check is necessary.
2. DFC conducts the consistency check if needed.

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

## Determining if a consistency check is needed

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

When a method defines a consistency check rule by naming a cache config object, DFC 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, DFC 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 DFC 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 DFC 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, DFC uses the information that it has in memory.

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

## Conducting consistency checks

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

DFC 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.

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

If a query 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 DFC uses the cached query results.

## The `client_pcaching_change` property

The `client_pcaching_change` property controls whether the persistent caches, including the object caches and all query caches, are flushed when a client session is started. When a client session is started, the DFC checks the cached value of the `client_pcaching_change` property against the repository value. If the values are different, the DFC flushes all the persistent caches, including the object caches and all query caches.

The `client_pcaching_change` value must be changed in the `docbase` config object manually, by a Superuser.

## For more information

- [Determining if a consistency check is needed, page 69](#), describes how the DFC determines whether a check is needed.
- [Conducting consistency checks, page 71](#), describes how the check is conducted.

- The *Content Server DQL Reference Manual* has reference information about the CHECK\_CACHE\_CONFIG administration method.

## The Data Model

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

- [Introducing object types and objects, page 73](#)
- [Properties, page 77](#)
- [The property bag , page 81](#)
- [Repositories, page 82](#)
- [Registered tables, page 90](#)
- [Documentum views, page 91](#)
- [The data dictionary, page 91](#)
- [What the data dictionary can contain, page 93](#)
- [The data dictionary and lifecycle states, page 98](#)
- [Retrieving data dictionary information, page 99](#)

## Introducing object types and objects

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

## What objects and object types are

An object is an individual item in the repository. An object type represents a class of objects. The definition of an object type consists of a set of properties, whose values describe individual objects of the type. Object types are like templates. When you create an object in a repository, you identify which type of object you want to create. Content

Server then uses the type definition as a template to create the object and then sets the properties for the object to values specific to that object instance.

## Supertypes, subtypes, and inheritance

Most EMC 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 properties of the supertype. The subtype also has the properties defined specifically for it. For example, the `dm_folder` type is a subtype of `dm_sysobject`. It has all the properties defined for `dm_sysobject` plus two defined specifically for `dm_folder`.

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

## Persistence

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

There are some object types that are not persistent. Objects of these types are created at runtime when they are needed. For example, collection objects and query result objects are not persistent. They are used at runtime to return the results of DQL statements. When the underlying RDBMS returns rows for a `SELECT` statement, 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.

## Type categories

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

- Standard object type

Standard object types are the types that do not fall into one of the remaining categories.

- **Aspect property object type**

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

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

Shareable object types are the parent types of lightweight object types. Only `dm_sysobject` and its subtypes can be defined as shareable. A single instance of a shareable type object is shared among many lightweight objects.
- An object type's category is stored in the `type_category` property in the `dm_type` object representing the object type.

## Lightweight and shareable object types

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

### What a lightweight type is

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

Lightweight SysObjects are useful if you have a large number of attribute values that are identical for a group of objects. This redundant information can be shared among the LWSOs from a single copy of the shared parent object. For example, Enterprise A-Plus Financial Services receives many payment checks each day. They record the images

of the checks and store the payment information in SysObjects. They will retain this information for several years and then get rid of it. For their purposes, all objects created on the same day can use a single ACL, retention information, creation date, version, and other attributes. That information is held by the shared parent object. The LWSO has information about the specific transaction.

Using lightweight SysObjects provide the following benefits:

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

## What a shareable type is

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

## For more information

- [How lightweight subtype instances are stored, page 85](#), describes how lightweight and shareable types are associated within the underlying database tables.

## Names of EMC Documentum object types

The names of all object types that are installed with Content Server or by an EMC Documentum client product start with the letters “dm”. There are four such prefixes:

- dm, which represents object types that are commonly used and visible to users and applications.
- dmr, which represents object types that are generally read only.
- dmi, which represents object types that are used internally by Content Server and EMC Documentum client products.
- dmc, which represents object types installed to support an EMC Documentum client application. They are typically installed by a script when Content Server is installed or when the client product is installed.

The use of ‘dm’ as the first two characters in an object type name is reserved for EMC Documentum products.

## Content files and object types

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

The content associated with an object is either primary content or renditions of the primary content. The format of all primary content for any one object must have the same file format. The renditions can be in any format.

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

## For more information

- [How lightweight subtype instances are stored, page 85](#), describes how lightweight objects are stored in the repository tables.
- [Chapter 8, Virtual Documents](#), describes virtual documents and their implementation.
- [Adding content, page 174](#), describes adding content to objects in detail.
- [Chapter 9, Renditions](#), describes renditions in detail.
- The *Documentum Object Reference Manual* has the following topics:
  - Rules for naming user-defined object types and properties
  - Reference description of the dm\_lightweight object type

## Properties

This section introduces properties, the fields that make up an object type.

## What properties are

Properties are the fields that comprise an object definition. The values in those fields describe individual instances of the object type. When an object is created, its properties are set to values that describe that particular instance of the object type. For example, two

properties of the document object type are title and subject. When you create a document, you provide values for the title and subject properties that are specific to that document.

## Property characteristics

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

### Persistent and non-persistent

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

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

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

### Single-valued and repeating

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

A repeating property stores multiple values in an indexed list. Index positions within the list are specified in brackets at the end of the property name when referencing a specific value in a repeating property; for example: keywords[2] or authors[17]. Special DQL functions exist to allow you to query values in a repeating property.

### Datatype

All properties have a datatype that determines what kind of values can be stored in the property. For example, a property with an integer datatype can only store whole

numbers. A property's datatype is specified when the object type for which the property is defined is created.

## Read and write or read only

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

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

## Qualifiable and non-qualifiable

Persistent properties are either qualifiable or non-qualifiable.

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

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

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

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

## Local and global

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

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

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

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

## Property identifiers

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

The identifier is an integer value stored in the `attr_identifier` property of each type's `dm_type` object. When a property is stored in a property bag, its identifier is stored in the property bag as a base64-encoded string in place of the property's name.

A property's identifier cannot be changed.

## For more information

- [The property bag](#), page 81, describes the property bag.
- [Supertypes, subtypes, and inheritance](#), page 74, explains supertypes and inheritance.
- The *Documentum Distributed Configuration Guide*, in the instructions for creating global users, contains instructions for configuring four local user properties to behave as global properties.
- The *Documentum Object Reference Manual*, contains complete information about persistent and computed properties, including naming rules and conventions, property identifiers, valid datatypes, and the limits and defaults for each datatype.

## Property domains

Persistent properties have domains. A property's domain identifies the property's datatype and several other characteristics of the property, such as its default value or label text for it.

You can query the data dictionary to retrieve the characteristics defined by a domain.

## The property bag

This section describes the property bag, a special property used to store other properties and their values.

### What the property bag is

The property bag is a special property used to store:

- Non-qualifiable properties and their values
- Aspect properties and their values if the properties are defined with fetch optimization

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

## Implementation

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

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

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

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

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

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

## Property bag overflow

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

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

## For more information

- [What a lightweight type is, page 75](#), describes the lightweight object type.
- [Aspects, page 111](#), describes aspects and aspect properties.
- The *Documentum Object Reference* has the reference description for the property bag property.
- The *DQL Reference Manual* describes how to alter a type to add a property bag.

## Repositories

This section describes repositories, the storage location for the objects managed by the Documentum system.

## What a repository is

A repository is where persistent objects managed by Content Server are stored. A repository stores the object metadata and, sometimes, content files. An EMC

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.

Repositories are implemented as sets of tables in an underlying relational database installation.

## Object type tables

The object type tables store metadata.

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

### Single-valued property tables

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

### Repeating property tables

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

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

**Table 1. Repeating properties 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 properties, the number of rows in the `_r` table for each object is equal to the number of values in the repeating property that has the most values. The columns for repeating properties having fewer values are filled in with NULLs.

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

**Table 2. Row determination in a repeating properties 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 properties that contain a smaller number of values with NULLs.

Even an object with no values assigned to any of its repeating properties has at least one row in its type's `_r` table. The row contains a NULL value for each of the repeating properties. 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` property has at least one value—its implicit version label.

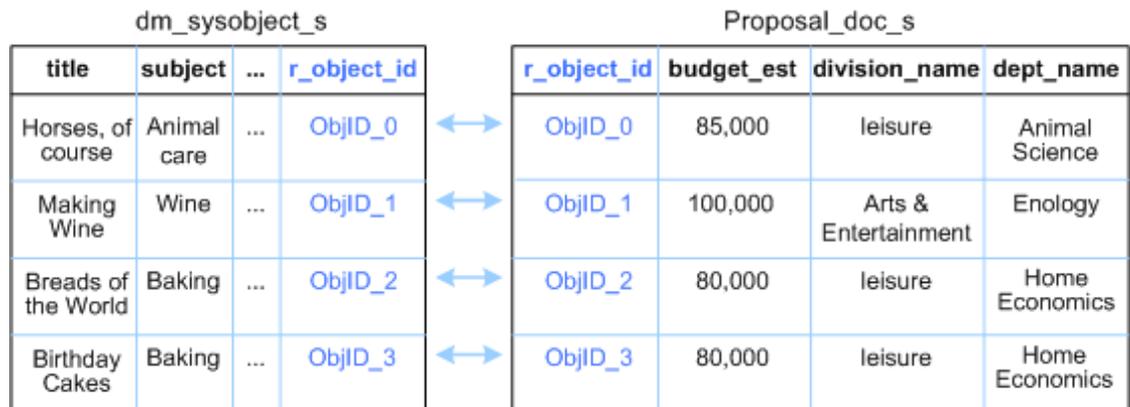
## How standard subtype instances are stored

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

row in the supertype's `_s` table. In the `_r` tables, the `r_object_id` and `i_position` values are used to join the rows.

For example, suppose you create a subtype of `dm_sysobject` called `proposal_doc`, with three properties: `budget_est`, `division_name`, and `dept_name`, all single-valued properties. Figure 1, page 85, illustrates the underlying table structure for this type and its instances. The values for the properties defined for the `proposal doc` type are stored in the `proposal_doc_s` table. Those properties that it inherits from its supertype, `dm_sysobject`, are stored in rows in the `dm_sysobject` object type tables. The rows are associated through the `r_object_id` column in each table.

**Figure 1. Example of database table storage for standard subtypes**



GEN-000105

## How lightweight subtype instances are stored

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

## Materialization and lightweight SysObjects

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

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

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

### Example of materialized and unmaterialized storage

This section illustrates by example how lightweight objects are stored and how materialization changes the underlying database records. Note that this example only uses the `_s` tables to illustrate the implementation. The implementation is similar for `_r` tables.

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

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

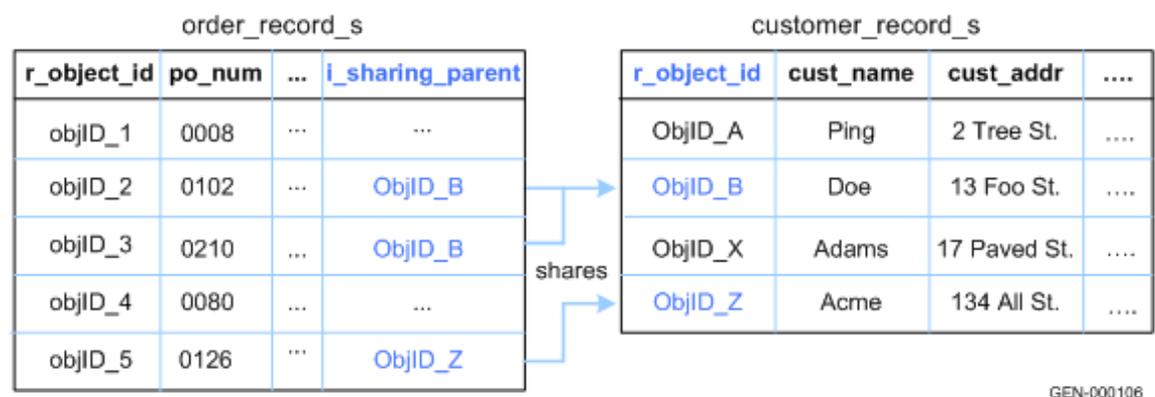
```
cust_name string(32),
cust_addr string(64),
cust_city string(32),
cust_state string(2)
cust_phone string(24)
cust_email string(100)
```

- `order_record`, with the following properties:

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

This type shares with `customer_record` and is defined for automatic materialization. Instances of the order record type will share the values of instances of the customer record object type. By default, the order record instances are unmaterialized. [Figure 2, page 87](#), shows how the unmaterialized lightweight instances are represented in the database tables.

**Figure 2. Database storage of unmaterialized lightweight object instances**



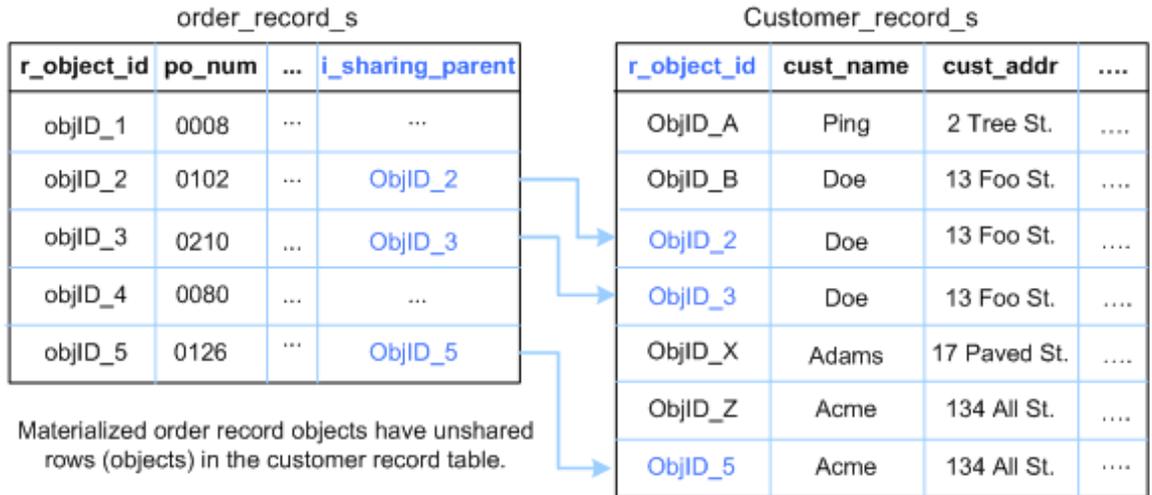
Unmaterialized order record objects share property values of customer record objects.

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

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

Because the order record object type is defined for automatic materialization, certain operations on an instance will materialize the instance. This does not create a new order record instance, but instead creates a new row in the customer record table that is specific to the materialized order record instance. [Figure 3, page 88](#), illustrates how a materialized instance is represented in the database tables.

**Figure 3. Database storage of materialized lightweight object instances**



GEN-000107

Materializing the order record instances created new rows in the customer\_record\_s table, one row for each order record object. The object ID of each customer record object representing a materialized order record object is set to the object ID of the order record object it represents, to associate the row with the order record object. Additionally, the i\_sharing\_status property of the previously shared customer record object is updated. In the order record objects, the i\_sharing\_parent property is reset to the object ID of the order record object itself.

## The location and extents of object type tables

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

On some databases, you can change the defaults when you create the repository. By setting 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` property 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.

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. Or, after the indexes are created, you can move them manually using the `MOVE_INDEX` administration method.

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

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

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

## Content storage areas

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

## For more information

- The *Documentum Object Reference* contains information about the identifiers recognized by Content Server.
- The *Content Server DQL Reference Manual* contains an expanded explanation of how NULLs are handled in Documentum.
- *Content Server Installation Guide* contains instructions for changing the default location and extents of object type tables and the locations of the index tables if it is possible on your platform.
- *Content Server Administration Guide* has a complete description of the storage area options.

## Registered tables

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

## What registered tables are

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. They are represented in the repository by objects of type dm\_registered.

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

## For more information

- The *Content Server DQL Reference Manual* has information about:
  - The REGISTER statement
  - Querying registered tables

## Documentum views

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

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

## The data dictionary

This section introduces the data dictionary and describes its use.

### What the data dictionary is

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

### Usage

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

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

## Localization support

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 properties 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 each of 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` property of the `dm_docbase_config` object.

## Modifying the data dictionary

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

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

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

## 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 property in a specific locale. A dd common info object contains the information that applies to both the property and type level across all locales for a given object type or property. For example, if a site has 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 property—one for the German locale and one for the English locale. However, there will be only one dd common info object for each object type and property because that object stores the information that is common across all locales.

Applications query the dd common, dd type info, and dd attr info objects to retrieve and use data dictionary information.

## For more information

- [What the data dictionary can contain, page 93](#), describes the kinds of information that can be stored in the data dictionary.
- [Retrieving data dictionary information, page 99](#), contains information about retrieving data dictionary information.
- The *Content Server Administration Guide* has information about:
  - Adding to or modifying the data dictionary
  - Publishing the data dictionary

## What the data dictionary can contain

This section describes the kinds of information that you can put in the data dictionary.

## Constraints

A constraint is a restriction applied to one or more property 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 a property or combination of properties for which every object of that type must have a unique value. The key can be one or more single-valued properties or one or more repeating properties. It cannot be a mixture of single-valued and repeating properties. All the properties in a unique key must be defined for the same object type.

You can include properties that allow NULL values because NULL never matches any value, even another NULL. To satisfy a unique key defined by multiple nullable properties, all the property values must be NULL or the set of values across the properties, 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 properties, all of which can contain NULLs. The properties are: A, B, and C. There are four objects of the type with the following values for A, B, and C:

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

You can define unique key constraints at either the object type level or the property level. If the key includes two or more participating properties, you must define it at the type level. If the key is a single property, it is typically defined at the property 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 property or combination of non-nullable properties 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 properties in a primary key must be single-valued properties that are defined for the same object type. Repeating properties are not allowed.

You can define primary key constraints at either the object type level or the property level. If the key includes two or more participating properties, you must define it at the type level. If the key is a single property, it is typically defined at the property 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 properties for one object type and one or more properties in another type. The number and datatypes of the properties in each set of properties must match. Additionally, if multiple properties 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 property level. If the key includes two or more participating properties, you must define it at the type level. If the key is a single property, it is typically defined at the property 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 properties must be of the same datatype.

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

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

## Not null

A NOT NULL constraint identifies a property that is not allowed to have a null value. You can only define a NOT NULL constraint at the property level. You can define NOT NULL constraints only for single-valued properties.

## 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 property's value.

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

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

## Default lifecycles for object 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 lifecycle, 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 and a way to enforce business rules concerning the appropriate lifecycle for any particular object type. Also, it allows you to write an application that will not require revision if the default changes for an object type.

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

The lifecycle defined as the default for an object type must be a lifecycle for which the type is defined as valid. Valid types for a lifecycle are defined by two properties in the dm\_policy object that defines the lifecycle in the repository. The properties are included\_type and include\_subtypes. A type is valid for a lifecycle if:

- The type is named in included\_type, or
- The included\_type property references one of the type’s supertypes and include\_subtypes is TRUE.

## 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 property 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 properties

An property's default value is the value Content Server assigns the property when new objects of the type are created unless the user explicitly sets the property 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 properties, some help text, and error messages.

## Value assistance

Value assistance provides a list of valid values for a property. A value assistance specification defines a literal list, a query, or a routine to list possible values for a property. Value assistance is typically used to provide a pick list of values for a property 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 properties, to define understandable text for each integer value. Client applications can then display the text to users instead of the integer values.

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

## 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 a DFC method.

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

Neither DQL or DFC queries return data dictionary information about new object types or added properties until that information is published, through an explicit `publishDataDictionary` method (in the `IDfSession` interface) or through the scheduled execution of the Data Dictionary Publisher job.

## Using DQL

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

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

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

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

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

## Using the DFC

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

## For more information

- The *Content Server DQL Reference Manual* provides a full description of the `DM_SESSION_DD_LOCALE` keyword.
- The Javadocs contain information about using the `IDfSession.getTypeDescription` method.

# Object Type and Instance Manipulations and Customizations

This chapter describes the manipulations that can be performed on the object type hierarchy and instances of object types. The chapter also introduces the business object framework (BOF) and the modules that comprise BOF. This chapter contains the following topics:

- [Object type manipulations, page 101](#)
- [Object instance manipulations, page 103](#)
- [Changing an object's object type, page 105](#)
- [Business object framework, page 107](#)

## Object type manipulations

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

## Creating new object types

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

New object types are created using the CREATE TYPE statement.

## Altering object types

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

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

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

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

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.

## For more information

- [Aspects, page 111](#), describes aspects and default aspects.
- The *Content Server DQL Reference Manual* has information about:
  - CREATE TYPE and the object types that are supported supertypes
  - ALTER TYPE and the possible alterations that can be made to object types
  - DROP TYPE

# Object instance manipulations

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

## Object creation

The ability to create objects is controlled by user privilege levels. Anyone can create documents and folders. To create a cabinet, a user must have the Create Cabinet privilege. To create users, the user must have the Sysadmin (System Administrator) privilege or the Superuser privilege. To create a group, a user must have Create Group, Sysadmin, or Superuser privileges.

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

In DQL, you use the CREATE OBJECT method to create a new instance of an object.

## Object modification

There are a variety of changes that users and applications can make to existing objects. The most common changes are changing property values; adding, modifying, or removing content; changing the object's access permissions; or associating the object with a workflow or lifecycle.

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

In the DFC, methods are part of the interface for individual classes. Each interface has methods that are defined for the class plus the methods inherited from its superclass. The methods associated with a class can be applied to objects of the class. For information about the DFC and its classes and interfaces, refer to *Developing DFC Applications*.

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 property values or adding content.

Creating or updating an object using DQL instead of the DFC is generally faster because DQL uses one statement to create or modify and then save the object. Using DFC

methods, you must issue several methods—one to create or fetch the object, several to set its properties, and a method to save it.

## Object destruction

Destroying an object removes it from the repository.

## Permissions and constraints

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.

## Effects of the operation

Destroying an object removes the object from the repository and also removes any relation objects that reference the object. (Relation objects are objects that define a relationship between two objects.) Only the explicit version is removed. Destroying an object does not remove other versions of the object. To remove multiple versions of an object, use a prune method.

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.

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` property to `TRUE` and removes all associated objects, such as relation objects, from the repository. The server also removes the object from all cabinets or folders and places it in the Temp cabinet. If the object is carrying the symbolic label `CURRENT`, it moves that label to the version in the tree that has the highest `r_modify_date` property value. This is the version that has been modified most recently.

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

## For more information

- User privilege levels and object-level permissions are described in [User privileges](#), page 128.
- [Removing versions](#), page 161 describes how the prune method behaves.
- The *DQL Reference Manual* contains full information about DQL and the reference information for the DQL statements, including `CREATE OBJECT`.
- For more information about the DFC and its classes and interfaces, refer to *Developing DFC Applications*, the Javadocs, or both.
- The *Content Server Administration Guide* contains a complete description of how to assign privileges and create ACLs.
- The *Documentum Object Reference Manual* contains information about relationships.
- The *Content Server Administration Guide*, contains information about the `dmclean` and `dmfilesan` jobs and how to execute the utilities manually.

## Changing an object's object type

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

## Constraints on the operation

The change is subject to the following restrictions:

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

A type identifier is a two-digit number that appears as the first two digits of an object ID. For example, the type identifier for all documents and document subtypes is 09. Consequently, the object ID for every document begins with 09.

- The new type must be either a subtype or supertype of the current type.

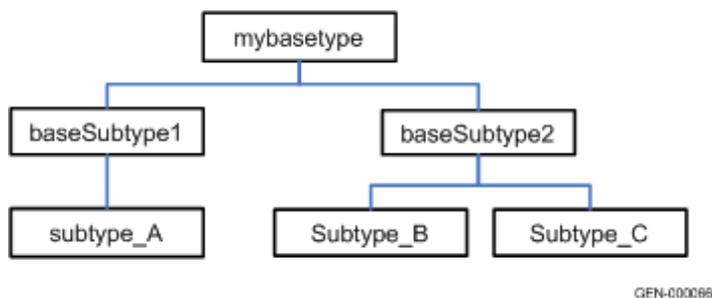
This means that type changes cannot be lateral changes in the object hierarchy. For example, if two object types, A and B, are both direct subtypes of mybasetype, then you cannot change an object of type A directly to type B.

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

## Example

Figure 4, page 106, shows an example of a type hierarchy. In this example, you can change subtype\_A to either baseSubtype\_1 or mybasetype. Similarly, you can change baseSubtype1 to either subtype\_A or mybasetype, or mybasetype to either baseSubtype1 or baseSubtype2. However, you cannot change baseSubtype1 to baseSubtype2 or Subtype\_B to Subtype\_C because these types are peers on the hierarchy. Lateral changes are not allowed. Only vertical changes within the hierarchy are allowed.

Figure 4. Sample hierarchy



GEN-00066

## For more information

- The *DQL Reference Manual* describes the syntax and use of the CHANGE OBJECT statement.

- [Changeable versions, page 162](#), describes immutability and which objects are changeable.

## Business object framework

This section describes the business object framework.

### Overview

This section provides a general introduction to the business object framework and its implementation.

### What the business object framework is

The business object framework (BOF) is a feature of DFC that allows you to write code modules to customize or add behaviors in DFC. The customizations may be applied at the service, object type, or object level.

### Benefits

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

- The customizations are independent of the client applications, removing the need to code the customization into the client applications.
- The customizations can be used to extend core Content Server and DFC functionality.
- The customizations execute well in an application server environment.

### What a BOF module is

A BOF module is unit of executable business logic and its supporting material, such as third-party software, documentation, and so forth. DFC supports four types of modules:

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

- Simple modules

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

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

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

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

## What comprises a module

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

## Module packaging and deployment

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

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

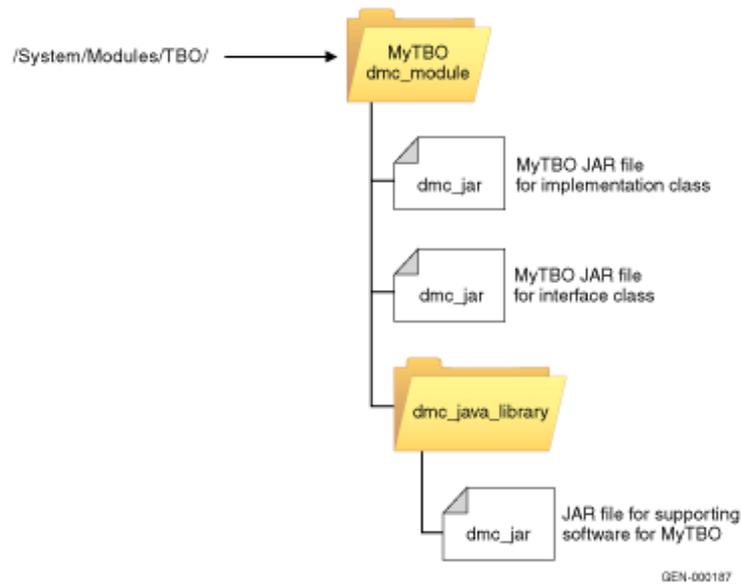
Installing a BOF module creates a number of repository objects. The top-level object is a `dmc_module` object. Module objects are subtypes of `dm_folder`. They serve as a container for the BOF module. The properties of a module object provide information about the BOF module it represents. For example,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

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

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

The jar objects representing the module's implementation and interface classes are linked directly to the `dmc_module` folder. The jar objects representing the JAR files for supporting software are linked into folders represented by `dmc_java_library` objects. The java library objects are then linked to the top-level module folder. [Figure 5, page 109](#), illustrates these relationships.

**Figure 5. Example of BOF module storage in a repository**



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

## Module interfaces on client machines

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

## Dynamic delivery and local caching

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

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

## BOF development mode

To allow you to easily test modules during development, DFC and Content Server support a development registry. This is a file that lists implementation classes to use during development. It loads the classes from the local classpath rather than being downloaded from a repository. For details about using this mode, refer to the *DFC Development Guide*.

## Service-based objects

This section describes the service-based objects, often called SBOs, a feature of the business object framework.

### What a service-based object is

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

## Implementation

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

## Type-based objects

This section describes the type-based objects, often called TBOs, a feature of the business object framework.

### What a type-based object is

A TBO associates a custom object type that extends an EMC Documentum object type with an implementation class that extends the appropriate DFC class. For example, suppose you want to add some validation behavior to a specific document subtype. You would create a TBO for that subtype with an implementation class that extends `IDfDocument`, adding the validation behavior.

## Implementation

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

## Aspects

This section describes aspects, aspect properties, default aspects, and their implementation.

### What an aspect is

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

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

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

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

## Aspect properties

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

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

Aspect properties are not fulltext-indexed by default. If you want to include the values in the index, you must use explicitly identify which properties you want indexed. You can use Documentum Composer or `ALTER ASPECT` for this.

## Implementation of aspect properties

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

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

## Default aspects

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

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

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

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

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

## Simple modules

This section describes simple modules.

### What a simple module is

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

### Implementation

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

## For more information

- The *DFC Development Guide* has instructions for how to create BOF modules of the various types and how to set up and enable the BOF development mode.
- The *Documentum Composer* documentation has instructions for packaging and deploying modules and information about deploying the interface classes to a client machine.
- The *DQL Reference Manual* describes the syntax and use of the ALTER TYPE and ALTER ASPECT statements.

## 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 115](#)
- [Repository security, page 119](#)
- [Users and groups , page 120](#)
- [User authentication, page 124](#)
- [Password encryption, page 126](#)
- [Application-level control of SysObjects, page 127](#)
- [User privileges , page 128](#)
- [Object-level permissions, page 129](#)
- [Table permits, page 132](#)
- [Folder security, page 133](#)
- [ACLs, page 134](#)
- [Auditing and tracing, page 136](#)
- [Signature requirement support, page 139](#)
- [Privileged DFC, page 148](#)
- [Encrypted file store storage areas, page 150](#)
- [Digital shredding, page 152](#)

### Security overview

This section provides a summary of the security features of EMC Documentum Content Server.

## Standard security features

An EMC Documentum Content Server installation supports numerous standard security features.

### Summary of standard features

Table 3, page 116, lists the standard security features.

**Table 3. Standard security features supported by Content Server**

Feature	Description
User authentication	User authentication is the verification that the user is a valid repository user. User authentication occurs automatically, regardless of whether repository security is active.
Password encryption	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 methods and a utility.
Application-level control of SysObjects	Application-level control of SysObjects is an optional feature that you can use in client applications to ensure that only approved applications can handle particular documents or objects.
User privileges	User privileges define what special functions, if any, a user can perform in a repository. For example, a user with Create Cabinet user privileges can create cabinets in the repository.
Object-level permissions	Object-level permissions define which users and groups can access a SysObject and which level of access those users have.
Table permits	Table permits are a set of permits applied only to registered tables, RDBMS tables that have been registered with Content Server.
Dynamic groups	Dynamic groups are groups whose membership is dynamic. If a group is dynamic, you can control its membership at runtime.

Feature	Description
ACLs	Object-level permissions are assigned using ACLs. Every SysObject in the repository has an ACL. The entries in the ACL define the access to the object.
Folder security	Folder security is an adjunct to repository security.
Auditing and tracing facilities	Auditing and tracing are optional features that you can use to monitor the activity in your repository.
Support for simple electronic signoffs and digital signatures	Content Server supports three options for electronic signatures. Support for simple signoffs, which use the IDfPersistentObject.signoff method, and for digital signatures, which is implemented using third-party software in a client application, are provided as standard features of Content Server. (Support for the third option, using the IDfSysObject.addESignature method, is only available with a Trusted Content Services license, and is not available on the Linux platform.)
Secure (SSL) communications between Content Server and the client library (DMCL) on client hosts	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.
Privileged DFC	This feature allows DFC to run under a privileged role, which gives escalated permissions or privileges for a specific operation.

## For more information

- For information about users and groups, including dynamic groups, refer to [Users and groups](#), page 120.
- [User authentication](#), page 124, describes user authentication in more detail.
- [Password encryption](#), page 126, provides more information about password encryption.
- [Application-level control of SysObjects](#), page 127, describes application level control of objects in more detail.
- For information about user privileges, object-level permissions, and table permits, refer to [User privileges](#), page 128.
- [ACLs](#), page 134, describes ACLs.

- [Folder security, page 133](#), describes folder security.
- [Auditing and tracing, page 136](#), provides an overview of the auditing and tracing facilities.
- [Signature requirement support, page 139](#), discusses all three options supporting signature requirements.
- [Privileged DFC, page 148](#), describes privileged DFC in detail.
- The *Content Server Administration Guide* has more information about the following topics:
  - Setting the connection mode for servers
  - Configuring clients to request a native or secure connection

## Trusted Content Services security features

Installing Content Server with a Trusted Content Services license adds additional security options.

### Security features available with Trusted Content Services license

[Table 4, page 118](#), lists the security features supported by a Trusted Content Services license.

**Table 4. Security features that require a Trusted Content Services license**

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

Feature	Description
Electronic signature support using the IDfSysObject.addESignature method	<p>addESignature is the way to implement an electronic signature requirement through Content Server. The method creates a formal signature page and adds that page as primary content (or a rendition) to the signed document. The signature operation is audited, and each time a new signature is added, the previous signature is verified first.</p> <p><b>Note:</b> Electronic signatures are not supported on the Linux platform.</p>
Ability to add, modify, and delete the additional types of entries in an ACL.	<p>The types of entries that you can manipulate in an ACL when you have a TCS license are:</p> <ul style="list-style-type: none"> <li>• AccessRestriction and ExtendedRestriction</li> <li>• RequiredGroup and RequiredGroupSet</li> <li>• ApplicationPermit and Application Restriction</li> </ul> <p>These types of entries 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.</p>

## For more information

- [Encrypted file store storage areas, page 150](#), describes encrypted storage areas in detail.
- [Digital shredding, page 152](#) provides a brief description of this feature.
- [Signature requirement support, page 139](#), describes how electronic signatures supported by addESignature work.
- [ACLs, page 134](#), provides more information about the permit types that you can define with a Trusted Content Services license.

## Repository security

The repository security setting controls whether object-level permissions, table permits, and folder security is enforced. The setting is recorded in the repository in the security\_mode property in the docbase config object. The property is set to ACL, which turns on enforcement, when a repository is created. Consequently, unless you have

explicitly turned security off by setting `security_mode` to `none`, object-level permissions and table permits are always enforced.

## Users and groups

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

This section provides an overview of how users and groups are implemented, to provide background understanding for a discussion of the standard security features.

### Users

This section introduces repository users.

#### What a repository user is

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

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

#### Repository implementation of users

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

The properties of a user object record information that allows Content Server to manage the user's access to the repository and to communicate with the user when necessary. For example, the properties define how the user is authenticated when the user requests

repository access. They also record the user's state (active or inactive), the user's email address (allowing Content Server to send automated emails when needed), and the user's home repository (if any).

## Local and global users

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

## For more information

- The *Content Server Administration Guide* has more information about users in general and complete instruction about creating local users.
- The *Distributed Configuration Guide* has complete information for creating and managing global users.
- The properties defined for the dm\_user object type are found in the *Documentum Object Reference Manual*.

## Groups

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

A group, 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.

Additionally, a group may be dynamic group. Membership in a dynamic group is determined at runtime. 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.

## Standard groups

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

Standard groups can be public or private. When a group is created by a user with Sysadmin or Superuser 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.

## Role groups

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

## Module role groups

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

## Privileged group

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

## Domain groups

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

## How role and domain groups are used

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

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

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

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

## What a dynamic group is

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.

## 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.

## Local and global groups

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

## For more information

- The *DQL Reference Manual* describes how to use ALTER GROUP.
- The *Content Server Administration Guide* has more information about groups in general and complete instructions about creating local groups.
- The *Distributed Configuration Guide* has complete information for creating and managing global groups.

## User authentication

### What user authentication is

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

### When authentication occurs

Content Server authenticates the user whenever a user or application attempts to open a repository connection or re-establish a timed out connection. The server checks that the user is a valid, active repository user. If not, the connection is not allowed. If the

user is a valid, active repository user, 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

## Authentication implementations

Content Server supports a variety of mechanisms for user authentication, including authentication against the operating system, against an LDAP directory server, using a plug-in module, or using a password stored in the repository.

There are several ways to configure user authentication, depending on your choice of authentication mechanism. For example, if you are authenticating against the operating system, you can write and install your own password checking program. If you use 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 the module 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 more information

- The *Content Server Administration Guide* has more information about the following topics:
  - User authentication options and procedures for implementing them
  - Setting up authentication failure limits

# Password encryption

This section introduces password encryption, the automatic process used by Content Server to protect certain passwords.

## What password encryption is

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 DFC automatically decrypts the password before passing the arguments to Content Server.

## How password encryption works

Content Server performs password encryption automatically for those passwords that it uses and those used by many of the internal jobs. Decrypting the passwords occurs automatically also. When an encrypted password is passed as an argument to a method, the DFC decrypts the password before passing the arguments to Content Server.

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

Passwords 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 more information

- The *Content Server Administration Guide* provides complete information about administering password encryption.
- Refer to the Javadocs for more information about `encryptPassword`.

# 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.

## What application-level control of SysObjects is

Application-level control of SysObjects is a feature that allows client applications to assert ownership of particular objects and, consequently, prohibit users from modifying or manipulating those objects through other applications.

## Implementation

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` property to its application code to identify the object as belonging to the application. Once set, the property can only be modified by that application or another that knows the application code.

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

When a non-Superuser user accesses an object, Content Server examines the object's `a_controlling_app` property. If the property has no value, then the user's access is determined solely by ACL permissions. If the property has a value, then the server compares the value to the values in the session's `application_code` property. 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 is not found, Content Server examines the `default_app_permit` property in the `docbase` config object. The user is granted access to the object at the level defined in that property (Read permission by default) or at the level defined by the object'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.

## User privileges

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

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

## Basic user privileges

[Table 5, page 128](#), lists the basic user privileges.

**Table 5. Basic user privilege 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 6, page 129, lists the extended user privileges.

**Table 6. Extended user privileges**

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

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

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

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

## For more information

- The *Content Server Administration Guide* contains a complete discussion and instructions on assigning privileges.

## Object-level permissions

This section introduces object-level permissions and lists the permission levels.

## What object-level permissions are

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

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

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

## Base object-level permissions

Table 7, page 130, lists the base permissions.

**Table 7. Base object-level permissions**

Level	Permission	Description
1	None	No access is permitted.
2	Browse	The user can look at property values but not at associated content.
3	Read	The user can read content but not update.
4	Relate	The user can attach an annotation to the object.
5	Version	The user can version the object, but cannot overwrite the existing version.
6	Write	The user can write and update the object.  Write permission confers the ability to overwrite the existing version.
7	Delete	The user can delete the object.

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

## Extended object-level permissions

Table 8, page 131, lists the extended permissions.

**Table 8. 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><b>Note:</b> Browse permission is not adequate to move an object.</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.
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	<p>The user can run the external procedure associated with the object.</p> <p>All users having at least Browse permission on an object are granted Execute Procedure permission by default for that object.</p>
Change Folder Links	<p>Allows a user to link an object to a folder or unlink an object from a folder.</p> <p>The permission must be defined in the ACL associated with the folder.</p>

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

## Default permissions

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

## For more information

- [Folder security, page 133](#), provides more information about folder security.
- For a description of privileges necessary to link or unlink an object, refer to the `IDfSysObject.link` and `IDfSysObject.unlink` method descriptions in the Javadocs.
- [ACLs, page 134](#), describes ACLs in more detail.

## Table permits

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

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

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

There are five levels of table permits, described in [Table 9, page 132](#).

**Table 9. Table permits**

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

Level	Permit	Description
4	Insert	The user can insert new data into the table.
8	Delete	The user can delete rows from the table.

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

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

## Folder security

This section contains a brief introduction to folder security, an adjunct to repository and object-level security.

### What folder security is

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

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

### Implementation

Folder security is turned on and off at the repository level, using the `folder_security` property in the `docbase config` object. By default, folder security is set to on when a repository is configured.

## For more information

- For complete information about folder security, including a complete list of the extra checks it imposes, refer to the security information in the *Content Server Administration Guide*.

## ACLs

This section introduces ACLs and the entries that make them up.

### What an ACL is

ACL is the acronym for access control list. ACLs are the mechanism that Content Server uses to impose object-level permissions on SysObjects. An ACL has one or more entries that identify a user or group and the object-level permissions accorded that user or group by the ACL.

Each SysObject object has an ACL. The ACL assigned to most SysObjects is used to control access to the object. Folders are the exception to this. The ACLs assigned to folders are not used to defined access to the folder. Instead, they are used by folder security and may be used as default ACLs for objects stored in the folder.

### Implementation overview

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

After an ACL is assigned to an object, the ACL 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 and manage them through DFC or DQL also.

## 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. 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 only 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

- The *Content Server Administration Guide* contains detailed descriptions of the type of entries you can place in an ACL and instructions for creating ACLs.
- [Assigning ACLs, page 185](#) describes the options for assigning ACLs to objects.
- [Appendix A, Aliases](#) has complete information about aliases.

## Auditing and tracing

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

### Auditing

This section introduces the auditing capabilities supported by Content Server.

#### What auditing is

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.

## What is audited

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

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

- All occurrences of an event on a particular object or object type
- All occurrences of a particular event, regardless of the object 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

## Recording audited events

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 property values in the audited object.

## Requesting auditing

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

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

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

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

## For more information

- The *Content Server Administration Guide* describes auditing, including a list of those events that are audited by default, how to initiate auditing, and what information is stored in an audit trail record.

## Tracing

This section introduces Content Server support for tracing, a useful troubleshooting tool.

### What tracing is

Tracing is an feature that logs information about operations that occur in Content Server and DFC. The information that is logged depends on which tracing functionality is turned on.

### Tracing options

Content Server and DFC support multiple tracing facilities. On Content Server, you can turn on tracing for a variety of server features, such as LDAP operations, content-addressed storage area operation, and operations on SysObjects. The jobs in the administration tool suite also generate trace files for their operations.

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

## For more information

- The *Content Server DQL Reference Manual* has reference information for the SET\_OPTIONS and MODIFY\_TRACE administration methods.
- The *Content Server Administration Guide* describes all the jobs in the administration tool suite.

## 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 signatures, page 139](#)
- [Digital signatures, page 146](#)
- [Signoff method usage, page 147](#)

Electronic signatures are generated and managed by Content Server. The feature is supported by two methods: IDfSysObject.addESignature and IDfSysObject.verifyESignature. Use this option if you require a rigorous signature implementation to meet regulatory requirements. You must have a Trusted Content Services license to use this option.

**Note:** Electronic signatures are not supported on the Linux platform.

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 addDigitalSignature method is executed. Use this option if you want to implement strict signature support in a client application.

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

## Electronic signatures

**Note:** This feature is not supported on the Linux platform. On supported platforms, it requires a Trusted Content Services license.

Electronic signatures are the most rigorous way that Content Server supports to fulfill a signature requirement.

## What an electronic signature is

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

## Overview of Implementation

Electronic signatures are generated by Content Server when an application or user issues an `addESignature` method. Signatures generated by `addESignature` are recorded in a formal signature page and added to the content of the signed object. The method is audited automatically, and the resulting audit trail entry is 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. 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.

## What addESignature does

When an application or user issues an `IDfSysObject.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.

## 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 you can use the electronic signature feature with no additional configuration. The only requirement for using the default functionality is that documents to be signed must be in PDF format or have a PDF rendition associated with their first primary content page.

### 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 properties in the audit trail entry to record information about the signature. [Table 10, page 144](#), lists the use of those properties for a `dm_addsignature` event.

**Table 10. Generic string property use for dm\_addesignature events**

Property	Stores
string_1	Name of the user who signed the object
string_2	The justification for the signature
string_3	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 pre_signatureHash argument of the addESignature method.  The information is formatted in the following manner:  <code>sig_number/method_name/pre_signature hash argument</code>
string_4	Hash of the primary content page 0. The information also records the hash algorithm and the format of the content. The information is formatted in the following manner:  <code>hash_algorithm/format_name/hash</code>
string_5	Hash of the signed content. The information also records the hash algorithm and the format of the content. The information is formatted in the following manner:  <code>hash_algorithm/format_name/hash</code>  If the signed content was added to the document as primary content, then value in string_5 is the same as the string_4 value.

## 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.

## Signature verification

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

- Calls the `IDfAuditTrailManager.verifyAudit` method to verify the 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 properties 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 `verifyESignature` against the replica will act on the source document.
- Using `addESignature` to sign a document requires at least `Relate` permission on the document.
- Using `verifyESignature` to verify a signature requires at least `Browse` permission on the signed document.

## For more information

- The *Content Server Administration Guide* has complete information about customizing electronic signatures and tracing the use of electronic signatures.

## Digital signatures

This section provides an overview of digital signatures.

### What a digital signature is

Digital signatures are electronic signatures, in formats such as PKCS #7, XML Signature, or PDF Signature, that are generated by client applications.

### Implementation overview

Digital signatures 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 EMC Documentum Professional Services or EMC Documentum Developer Support.

Content Server supports digital signatures with an property on SysObjects and the addDigitalSignature method. The property is a Boolean property called a\_is\_signed. to indicate whether the object is signed. The addDigitalSignature method generates an audit trail entry recording the signing. The event name for the audit trail entry is dm\_adddigisignature. 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 property to T.
4. Check the document in to the repository.
5. Issue the addDigitalSignature method to generate the audit trail entry.

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

## For more information

- For information about methods to request auditing for the `dm_adddigsignature` event, refer to the `IDfAuditTrailManager` interface in the Javadocs.
- The *Content Server Administration Guide* provides more information about Content Server signatures on audit trail entries.

## Signoff method usage

This section introduces simple sign-offs, the least rigorous way to satisfy a signature requirement.

### What a simple sign-off is

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

### Implementation overview

Simple sign-offs are implemented using a `IDfPersistentObject.signoff` method. The method accepts a user authentication name and password as arguments. When the method is executed, 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 more information

- The *Content Server Administration Guide* provides instructions for creating a custom signature validation program.
- Refer to the Javadocs for the IDfPersistentObject.signoff usage notes.

## Privileged DFC

**Note:** This feature is currently supported only for internal use by Documentum client products. It is not supported for use by external applications.

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

Supporting privileged DFC is a set of privileged group, privileged roles, and the ability to define TBOs and simple modules as privileged modules. The privileged groups are groups whose members are granted a particular permission or privileged automatically. You can add or remove users from these groups. The privileged roles are groups defined as role groups that may be used by DFC to give the DFC an escalated permission or privilege required to execute a privileged module. Only DFC can add or remove members in those groups. A privileged module is a module that is defined as a module that uses one or more escalated permissions or privileges to execute.

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

**Note:** In some workstation environments, it may also be necessary to manually modify the Java security policy files to use privileged DFC. For details, refer to the *Content Server Administration Guide*.

You can disable the use of escalated privileges by a DFC instance. This is controlled by a key in the dfc.properties file.

## Privileged DFC registrations

Three objects are used to register a DFC instance for privileged roles:

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

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

The PKI credentials for a DFC are stored by default in a file named `dfc.keystore` in the same directory as the `dfc.properties` file. You can change the file's location and name if you wish, by setting a key in the `dfc.properties` file.

The first time a DFC instance is initialized, it creates its own PKI credentials and publishes its identity to the global registry. For subsequent startups, DFC instance checks for the presence of its credentials. If they are not found or are not accessible—for instance, when a password has changed—the DFC recreates the credentials and republishes its identity to the global registry if privileged DFC is enabled in the `dfc.properties` file. Republishing the credentials causes the creation of another client registration object and public key certificate object for the DFC instance.

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

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

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

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

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

## How Content Server recognizes a privileged DFC instance

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

## Using approved DFC instances only

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

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

## For more information

- The *Content Server Administration Guide* contains procedures and instructions for configuring privileged DFC.

## Encrypted file store storage areas

Encrypted file store storage areas are an optional security feature. They are available only if you have installed Content Server with a Trusted Content Services license.

## What encrypted file store storage areas are

An encrypted file store storage area is a file store storage area that contains encrypted content files. If you installed Content Server with a Trusted Content Services license, 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.

## Implementation overview

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` property of the storage area object (`dm_filestore` object). It is encrypted using the repository encryption key.

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

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.

**Note:** The encryption key is 192 bits in length and is used with the Triple DES-EDE-CBC algorithm.

## For more information

- The *Content Server Administration Guide* has more information about the following topics:
  - The repository encryption key
  - Dump and load operations

# Digital shredding

Digital shredding is a feature available with a Trusted Content Services license.

## What digital shredding is

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.

## Implementation overview

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

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

# 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 Administration Guide*. This chapter includes the following topics:

- [Introducing SysObjects, page 153](#)
- [Documents, page 154](#)
- [Document content , page 155](#)
- [Versioning, page 157](#)
- [Immutability, page 163](#)
- [Concurrent access control, page 165](#)
- [Document retention and deletion, page 167](#)
- [Documents and lifecycles, page 172](#)
- [Documents and full-text indexing, page 172](#)
- [SysObject creation, page 173](#)
- [Modifying SysObjects, page 179](#)
- [Managing permissions, page 184](#)
- [Managing content across repositories, page 189](#)
- [Relationships between objects, page 191](#)
- [Managing translations, page 192](#)
- [Annotation relationships, page 193](#)

## 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 EMC Documentum 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 `dm_sysobject`. 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.

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

### What a simple document is

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, and all have the same file format.

### What a virtual document is

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

Using virtual documents 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.

### For more information

- [Chapter 8, Virtual Documents](#), describes virtual documents in detail.

# Document content

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

## What a content object is

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

## Implementation

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.

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

## Primary content and renditions

All content associated with a document is either primary content or a rendition.

## What primary content is

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

## What a rendition is

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 Content Transformation Services. Content Transformation Services is an optional product that lets you manage rich media such as jpeg and various audio and video formats. With Content Transformation Services, you can create thumbnails and other renditions for the rich media formats. [Chapter 9, Renditions](#), contains an in-depth description of renditions generated by Content Server and briefly describes renditions generated by Content Transformation Services. Renditions generated by Content Transformation Services are described in detail in the Content Transformation Services documentation.

## 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.

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

## 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.

## Translations

Content Server contains support for managing translations of original documents using relationships.

## For more information

- [Page numbers, page 156](#), has more information about page numbers.
- [Chapter 9, Renditions](#), has information about creating renditions using converters and managing renditions.
- *Administering Documentum Media Transformation Services* describes EMC Documentum Media Transformation Services.
- [Managing translations, page 192](#), has more information about setting up translation relationships.
- The *Content Server Administration Guide* has complete information about the content checking and duplication feature and about the features supported by the Content Storage Services license.

## Versioning

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

## What versioning is

Versioning is an automated process that creates a historical record of a document. Each time you check in or branch a document or other SysObject, Content Server creates a new version of the object without overwriting the previous version. All the versions

of a particular document are stored in a virtual hierarchy called a version tree. Each version on the tree has a numeric version label and optionally, one or more symbolic version labels.

## Version labels

Version labels are used to uniquely identify a version within a version tree.

### r\_version\_label attribute

Version labels are recorded in the `r_version_label` attribute defined for the `dm_sysobject` object type. This is a repeating attribute. The first index position (`r_version_label[0]`) is reserved for an object's numeric version label. The remaining positions are used for storing symbolic labels.

### Numeric version labels

The numeric version label is a number that uniquely identifies the version within the version tree. The numeric version label is generally assigned by the server and is always stored in the first position of the `r_version_label` attribute (`r_version_label[0]`). By default, the first time you save an object, the server sets the numeric version label to 1.0. Each time you check out the object and check it back in, the server creates a new version of the object and increments the numeric version label (1.1, 1.2, 1.3, and so forth). The older versions of the object are not overwritten. If you want to jump the version level up to 2.0 (or 3.0 or 4.0), you must do so explicitly while checking in or saving the document.

**Note:** If you set the numeric version label manually the first time you check in an object, you can set it to any number you wish, in the format *n.n*, where *n* is zero or any integer value.

### Symbolic version labels

A symbolic version label is either system- or user-defined. Using symbolic version labels lets you provide labels that are meaningful to applications and the work environment.

Symbolic labels are stored starting in the second position (`r_version_label[1]`) in the `r_version_label` attribute. To define a symbolic label, simply define it in the argument list when you check in or save the document.

An alternative way to define a symbolic label is to use an `IDfSysObject.mark` method. A mark method assigns one or more symbolic labels to any version of a document. For example, you can use a mark method, in conjunction with an unmark method, to move a symbolic label from one document version to another.

A document can have any number of symbolic version labels. Symbolic labels are case sensitive and must be unique within a version tree.

## The CURRENT label

The symbolic label `CURRENT` is the only symbolic label that the server can assign to a document automatically. When you check in a document, the server assigns `CURRENT` to the new version unless you specify a label. If you specify a label (either symbolic or implicit), then you must also explicitly assign the label `CURRENT` to the document if you want the new version to carry the `CURRENT` label. For example, the following checkin call assigns the labels `inprint` and `CURRENT` to the new version of the document being checked in:

```
IDfId newSysObjId = sysObj.checkin(false, "CURRENT,inprint");
```

If you remove a version that carries the `CURRENT` label, the server automatically reassigns the label to the parent of the removed version.

## Maintaining uniqueness

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

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

**Note:** Symbolic labels are case sensitive. Two symbolic labels are not considered the same if their cases differ, even if the word is the same. For example, the labels `working` and `Working` are not the same.

## 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.

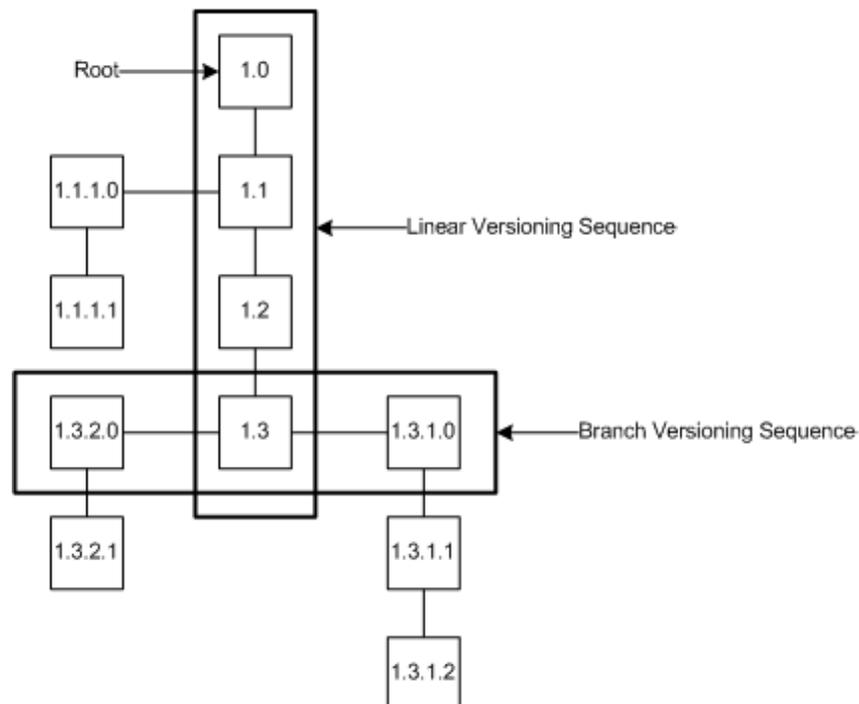
To identify which version tree a document belongs to, the server uses the document's `i_chronicle_id` attribute value. This attribute contains the object ID of the document's original version, the root of the version tree. Each time you create a new version, the server copies the `i_chronicle_id` value 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 numeric 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, page 160](#), shows a version tree that contains branches.

**Figure 6. A version tree with branches**



The numeric version labels on versions in branches always have two more digits than the version at the origin of the branch. For example, looking at the example, version 1.3 is the origin of two branches. These branches begin with the numeric version labels 1.3.1.0 and 1.3.2.0. If 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 `IDfSysObject.branch` method instead of the checkout method when you get the document from the repository.

When you use a branch method, the server copies the specified document and gives the copy a branched version number. The method returns the `IDfID` object representing the new version. The parent of the new branch is marked immutable (unchangeable).

After you branch a document version, you can make changes to it and then check it in or save it. If you use a checkin method, you create a subsequent version of your branched document. If you use a save method, you overwrite the version created by the branch method.

A branch method is particularly helpful if you want to check out a locked document.

## Removing versions

Content Server provides two ways to remove a version of a document. If you want to remove only one version, use a `IDfPersistentObject.destroy` method. If you want to remove more than one version, use a `IDfSysObject.prune` method.

With a prune method, you can prune an entire version tree or only a portion of the tree. By default, prune removes any version that does not belong to a virtual document and does not have a symbolic label.

To prune an entire version tree, identify the first version of the object in the method'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.

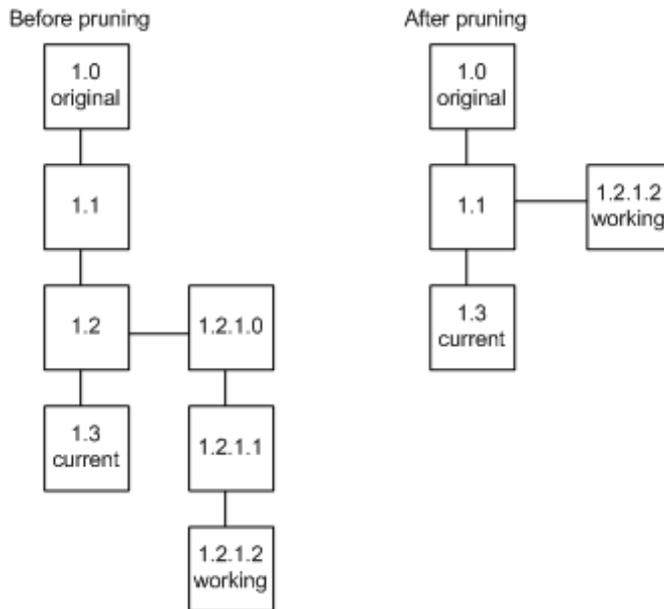
To prune only part of the version tree, specify the object ID of the version at the beginning of the portion you want to prune. For example, to prune the entire tree shown in [Figure 6, page 160](#), specify the object ID for version 1.0. To prune only version 1.3 and its branches, specify the object 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 7, page 162](#). 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 7. 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, page 160](#), you can modify the following versions:

- 1.3
- 1.3.1.2
- 1.3.2.1
- 1.1.1.1

The other versions are immutable. However, you can create new, branched versions of immutable versions.

## For more information

- [Version trees, page 160](#), describes version trees in detail.
- [Removing versions, page 161](#), contains more information about removing versions.
- Refer to the Javadocs for detailed information about the mark and unmark methods.
- [Immutability, page 163](#), describes immutability in more detail.

## Immutability

Immutability is a characteristic that defines an object as unchangeable. An object is marked immutable if one of the following occurs:

- The object is versioned or branched.
- An `IDfSysObject.freeze` method is executed against the object.
- The object is associated with a retention policy that designates controlled documents as immutable.

## 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 an `IDfWorkflow.haltAll` 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.

- `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, the method sets other attributes also and offers the option of freezing the components of any snapshot associated with the object.

To unfreeze an object, use an `IDfSysObject.unfreeze` method. Unfreezing an object resets the `r_frozen_flag` attribute to FALSE. If the object has not been previously versioned, then unfreezing it also resets the `r_immutable_flag` to FALSE. The method has an argument that, if set to TRUE, unfreezes the components of a snapshot associated with the object.

## 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 numeric 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:

- `a_archive`
- `i_isdeleted`
- `i_vstamp`
- `r_access_date`
- `r_alias_set_id`
- `r_aspect_name`
- `r_current_state`

- `r_frozen_flag`
- `r_frzn_assembly_cnt`
- `r_policy_id`
- `r_immutable_flag`
- `i_reference_cnt`
- `r_policy_id`
- `r_resume_state`

A data dictionary attribute defined for the `dm_dd_info` type provides additional control over immutability for objects of type `dm_sysobject` or any subtypes of `SysObject`. The attribute is called `ignore_immutable`. When set to `TRUE` for a `SysObject`-type attribute, the attribute is changeable even if the `r_immutable_flag` for the containing object instance is set to `TRUE`.

## For more information

- [Attributes that remain changeable, page 164](#), lists the attributes that can be changed in a frozen object.
- [What freezing does, page 216](#), describes the additional attributes that are set when a virtual document is frozen.
- [Unfreezing a document, page 217](#), describes how unfreezing affects a virtual document.
- The *Content Server DQL Reference Manual* contains instructions for using the `ALTER TYPE` statement to set or change data dictionary attributes.

## 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 an explicit method or statement issued by a user or application. For example, the DQL BEGINTRAN statement starts an explicit transaction. The database lock is released when the explicit transaction is committed or aborted.

A system administrator or Superuser can lock any object with a database-level lock. Other users must have at least Write permission on an object to place a database lock on the object. Database locks are set using the `IDfPersistentObject.lock` method.

Database locks provide a way to ensure that deadlock does not occur in explicit transactions and that save operations do not fail due to version mismatch errors.

If you use database locks, using repository locks 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 check out 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 have Superuser user privileges.

Repository locks are released by check-in methods (`IDfSysObject.checkin` or `IDfSysObject.checkinEx`). A check-in method creates a new version of the object, removes the lock on the old version, and gives you the option to place a lock on the new version.

If you use a save method to save your changes, you can choose to keep or relinquish the repository lock on the object. Save methods, which overwrite the current version of an object with the changes you made, have an argument that allows you to direct the server to hold the repository lock.

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

## 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.

## For more information

- [Object-level permissions, page 129](#), introduces the object-level permissions.
- [User privileges , page 128](#), introduces user privileges.

## 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

Retention policies are part of the larger retention services provided by Retention Policy Services. These services allow you to manage the entire life of a document, including its disposition after the retention period expires. Consequently, documents associated with an active retention policy are not automatically deleted when the retention period expires. Instead, they are held in the repository until you impose a formal disposition or use a privileged delete to remove them.

Using retention policies requires a Retention Policy Services license. If Content Server is installed with that license, you can define and apply retention policies through Retention Policy Services Administrator (an administration tool that is similar to, but separate from, Documentum Administrator). Retention policies can be applied to documents in any storage area type.

Using retention policies is the recommended way to manage document retention.

- Content-addressed storage area retention periods

If you are using content-addressed storage areas, you can configure the storage area to enforce a retention period on all content files stored in that storage area. The period is either explicitly specified by the user when saving the associated document or applied as a default by the Centera host system.

## Retention policies

A retention policy defines how long an object must be kept in the repository. The retention period can be defined as an interval or a date. For example, a policy might specify a retention interval of five years. If so, then any object to which the policy is applied is held for five years from the date on which the policy is applied. If a date is set as the retention period, then any object to which the policy is applied is held until the specified date.

A retention policy is defined as either a fixed or conditional policy. If the retention policy is a fixed policy, the defined retention period is applied to the object when the policy is attached to the object. For example, suppose a fixed retention policy defines a retention period of five years. If you attach that policy to an object, the object is held in the repository for five years from the date on which the policy was applied.

If the retention policy is a conditional policy, the retention period is not applied to the object until the event occurs. Until that time, the object is held under an infinite retention; that is the object is retained indefinitely. After the event occurs, the retention period defined in the policy is applied to the object. For example, suppose a conditional retention policy requires employment records to be held for 10 years after an employee leaves a company. This conditional policy is attached to all employment records. The records of any employee are retained indefinitely until the employee leaves the company.

At that time, the conditional policy takes effect and the employee's records are marked for retention for 10 years from the date of termination.

You can apply multiple retention policies to an object. In general, the policies can be applied at any time to the object.

The date an object's retention expires is recorded in an object's `i_retain_until` property. However, if there are conditional retention policies attached to the object, the value in that property is null until the condition is triggered. If there are multiple conditional retention policies attached to the object, the property is updated as each condition is triggered if the triggered policy's retention period is farther in the future than the current value of `i_retain_until`. However, Content Server ignores the value, considering the object under infinite retention, until all conditions are triggered.

A policy can be created for a single object, a virtual document, or a container such as a folder. If the policy is created for a container, all the objects in the container are under the control of the policy.

An object can be assigned to a retention policy by any user with Read permission on the object or any user who is a member of either the `dm_retention_managers` group or the `dm_retention_users` group. These groups are created when 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.

## Behavior if both a retention policy and storage-based retention apply

If a retention policy is assigned to a document 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 property `i_retain_until` is set to the date furthest in the future. For example, suppose a document created on April 1, 2005 is stored in a content-addressed storage area and assigned to a retention policy. The retention policy specifies that it must be held for five years. The expiration date for the policy is May 31, 2010. The content-addressed storage area has a default retention period of eight years. The expiration date for the storage-based retention period is May 31, 2013. Content Server will not allow the document to be deleted (without using a forced deletion) until May 31, 2013. The `i_retain_until` property is set to May 31, 2013.

If the retention policy is a conditional retention policy, the property value is ignored until the event occurs and the condition is triggered. At that time, the property is set to the retention value defined by the conditional policy. If multiple conditional retention policies apply, the property is updated as each is triggered if the triggered policy's retention period is farther in the future than the value already recorded in `i_retain_until`. However, Content Server ignores the value in `i_retain_until` all the policies are triggered. Until all conditional policies are triggered, the object is held in infinite retention.

## 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.

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.

You can remove unneeded renditions using the Rendition Management administration tool.

## Retention in distributed environments

In a distributed environment, a document's content may be saved to a repository from a remote location, through a BOCS server, using either a synchronous or an asynchronous write operation. In a synchronous write operation, the content is saved to the appropriate storage area immediately. In an asynchronous write operation, the content is parked on the BOCS server and written to the storage area at a later time. In both options, the document's metadata is saved to the repository immediately.

Regardless of whether the content is written synchronously or asynchronously, if the document is under retention, retention is enforced as soon as the document metadata is saved to the repository.

## For more information

- [Retention policies, page 168](#), describes retention policies. For more information about retention policies and their use, refer to Documentum Administrator online help.
- The *Content Server Administration Guide* has information about:
  - Content-addressed storage areas
  - How retention periods in a content-addressed storage area are defined and how they are implemented internally
  - Enabling the use of a Centera profile for a forced deletion
  - The Version Management tool
  - The Rendition Management tool
- [Removing versions, page 161](#), describes using destroy or prune in detail.

## Documents and lifecycles

Lifecycles represent the stages of a document's life. A lifecycle consists of a linear sequence of states. Each state has associated entry criteria and actions that must be performed before an object can enter the state.

After you create a document, you can attach it to any lifecycle that is valid for the document's object type. Only a user with the Change State extended permission can move the document from one state to another.

## Documents and full-text indexing

All object of type SysObject or SysObject subtypes are full-text indexed. The values of the object's properties and content, if it has content, are indexed. Properties defined for any aspects associated with an object are not indexed unless those properties are defined for indexing in the aspect definition.

It is not possible to turn off indexing of the properties unless you disable indexing completely. You can turn off indexing of content however. Content indexing is controlled by the `a_full_text` property.

By default, the property is T (TRUE) and both the content and metadata (property values) are indexed. If an object's `a_full_text` property is F (FALSE), then only the object's metadata is indexed.

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

## SysObject creation

The most commonly created `SysObject` is a document or a document subtype. End users typically use a Documentum client application, such as `WebPublisher` or `Webtop`, to create documents or to import documents created in an external editor. For information about using an EMC Documentum client product to create documents, or other `SysObjects`, refer to the documentation for that product. For information about creating documents or other `SysObjects` programmatically, refer to the *DFC Developer's Guide*. This section provides some important conceptual information about certain properties and the management of new `SysObjects`.

## Who owns the new object

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 change the object owner, you must be a `Superuser`, the current owner of the object, or a user with `Change Owner` permission.

## Where the object is stored in the repository

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 sometimes 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. It is possible to specify a different location programmatically by setting the `default_folder` property or linking the object to a different location.

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

## Adding content

When users create a SysObject using a Documentum client, adding content is a seamless operation. Creating a new document using Web Publisher 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 you add to an object determines that object's primary format. The format is set and recorded in the `a_content_type` property of the object. Thereafter, all content added to the object as primary content must have the same format as that first primary content file.

**Note:** If you discover that an object's `a_content_type` property is set incorrectly, it is not necessary to re-add the content. You can check out the object, reset the property, and save (or check in) the object.

## Renditions

Renditions are typically copies of the primary content in a different format. You can add as many renditions of primary content as needed.

## Macintosh files

You cannot use DQL to add a file created on a Macintosh machine to an object. You must use a DFC method. Older Macintosh-created files have two parts: a data fork (the actual text of the file) and a resource fork. The DFC, in the `IDfSysObject` interface, includes methods that allow you to specify both the content file and its resource fork when adding content to a document.

## Where the content is stored

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.

### 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 check in operation.

### 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

## 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.

## Setting content properties and metadata for content-addressed storage

Content-addressed storage areas allow you to store metadata, including a value for a retention period, with each piece of content in the system. Each of the storage system metadata fields that you want to set when content is stored is identified in the ca store object and in the content object representing the content file.

When a content file is saved to content-addressed storage, the metadata values are stored first in the content object and then copied into the storage area. Only those metadata fields that are defined in both the content object and the ca store object are copied to the storage area.

In the content object, the properties that record the metadata are:

- `content_attr_name`
- `content_attr_num_value`
- `content_attr_value`
- `content_attr_date_value`
- `content_attr_data_type`

These are repeating properties. When a `setContentAttrs` method or a `SET_CONTENT_ATTRS` administration method is issued, the name and value pairs identified in the parameter argument are stored in these content properties. The name is placed in `content_attr_name` and the value is stored in the property 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 properties that identify the metadata are:

- `a_content_attr_name`

This is a list of the metadata fields in the storage area to be set

- `a_retention_attr_name`

This identifies the metadata field that contains the retention period value.

When `setContentAttrs` executes, the metadata name and value pairs are stored first in the content object properties and then the plug-in library is called to copy them from the content object to the storage system metadata fields. Only those fields that are identified in both `content_attr_name` in the content object and in either `a_content_attr_name` or `a_retention_attr_name` in the storage object are copied to the storage area.

If `a_retention_attr_required` is set to T (TRUE) in the `ca` store object, the user or application must specify a retention period for the content when saving the content. That is accomplished by including the metadata field identified in the storage object's `a_retention_attr_name` property in the list of name and value pairs when setting the content properties.

If `a_retention_attr_required` is set to F (FALSE), then the content is saved using the default retention period, if one is defined for the storage area. However, the user or application may overwrite the default by including the metadata field identified in the storage object's `a_retention_attr_name` property when setting the content.

The value for the metadata field identified in `a_retention_attr_name` can be a date, a number, or a string. For example, suppose the field name is "retain\_date" and content must be retained in storage until January 1, 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.

## SysObjects and ACLs

An ACL, or access control list, specifies access permissions for an object. The standard entries can give a user or group any of the basic access permissions, extended permissions or both. With a Trusted Content Services license, you can also add entries that restrict access for specific users or groups and entries that specify permissions recognized only by specific applications.

Each object of type SysObject or SysObject subtype has one ACL that controls access to that object. The server automatically assigns a default ACL to a new SysObject if you do not explicitly assign an ACL to the object when you create it. If a new object is stored in a room and is governed by that room, the ACL assigned to the object is default ACL for that room.

The ACL associated with an object is identified by two properties of the SysObject: acl\_name and acl\_domain. The name is the name of the ACL and acl\_domain records the owner of the ACL.

## For more information

- [Chapter 9, Renditions](#), contains information about creating and adding renditions.
- [Assigning ACLs, page 185](#) contains information about setting permissions for a SysObject.
- The *Content Server Administration Guide* has information about:
  - The implementation and use of the options for determining where content is stored
  - The behavior and implementation of content assignment policies and creating them
  - How the default storage algorithm behaves
  - Configuring a storage area to require a retention period for content stored in that area

# Modifying SysObjects

This section contains information about modifying existing documents and other SysObjects.

## Who can modify a document

The ability to modify a SysObject is controlled by object-level permissions and whether the object is under the control of a retention policy. Additionally, the ability to modify an object may be affected by application-defined roles.

Object-level permissions are defined in ACLs. Each SysObject has an associated ACL object that defines the access permissions for that object. The entries in the ACL define who can access the object and the operations allowed for those having access. Users with Superuser privileges can always access a SysObject because a Superuser always has at least Read permission on SysObjects and has the ability to modify ACLs.

If the object is under the control of a retention policy, users cannot overwrite the content regardless of their permissions. Documents controlled by a retention policy may only be versioned or copied. Additionally, some retention policies set documents under their control as immutable. If that is so, then users can change only some of the document's attributes.

Application-level control of access to a particular object is accomplished using role groups. These groups control which applications users can use to modify a document.

You cannot modify the content of objects that are included in a frozen (unchangeable) snapshot or that have the `r_immutable_flag` attribute set to TRUE. Similarly, most attributes of such objects are also unchangeable.

## Getting a document from the repository

Before a user or application can modify a SysObject, the object must be obtained from the repository. There are three options for obtaining an object from the repository:

- A lock method
- A checkOut method
- A fetch method

These methods retrieve the object's metadata from the repository. Retrieving the object's content requires a separate 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.

Checking out a document places a repository lock on the object. A repository lock ensures that while you are working on a document, no other user can make changes to that document. Checking out a document also offers you two alternatives for saving the document when you are done. You need `Version` or `Write` permission to check out a document.

Use a `fetch` method when you want to read but not change an object. The method does not place either a repository or database lock on the object. Instead, the method uses optimistic locking. Optimistic locking does not restrict access to the object; 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.

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, retrieve the content file first.

## Modifying single-valued attributes

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

In DFC, most attributes have a specific `set` method that sets the attribute. For example, if you wanted to set the `subject` attribute of a document, you call the `setSubject` method. There is also a generic `set` method that you can use to set any attribute.

## Modifying repeating attributes

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

When you add a value, you can append it to the end of the values in the repeating property or you can replace an existing value. If you remove a value, all the values at

higher index positions within the property are adjusted to eliminate the space left by the deleted value. For example, suppose a keywords property has 4 values:

```
keywords[0]=engineering
keywords[1]=productX
keywords[2]=metal
keywords[3]=piping
```

If you removed productX, the values for metal and piping are moved up and the keywords property now contains the following:

```
keywords[0]=engineering
keywords[1]=metal
keywords[2]=piping
```

## Performance tip for repeating attributes

How long it takes the server to append or insert a value for a repeating property increases in direct proportion to the number of values in the property. Consequently, if you want to define a repeating property for a type and you expect that property to hold hundreds or thousands of values, it is recommended that you create an RDBMS table to hold the values instead and then register the table. When you query the type, you can issue a SELECT that joins the type and the table.

## Adding additional content

When you add primary content to an object, you can append the new file to the end of the object's list of primary content files or you can insert the file into the list. There are a variety of methods in the IDfSysObject interface for adding content. Which you use depends on the operation you wish to perform. For example, if you want to append an existing content file, you can use `appendFile`. If you want to insert the content into the list of primary content, use an `insertFile`.

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

Although the DQL `CREATE...OBJECT` and `UPDATE...OBJECT` statements support a clause that allows you to add content, DQL requires Superuser privileges and, if executed through IDQL, bypasses any TBOs or SBOs that you may have associated with the object or operation. DQL executed in a program does not bypass these modules.

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.

## 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.

You cannot remove a content page if the content has a rendition with the `keep` flag set to `true` and the page is not the last remaining page in the document.

## 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 checkinEx methods

Use `checkin` or `checkinEx` to create a new version of a object. You must have at least Version permission for the object. The methods work only on checked-out documents.

The `checkinEx` method is specifically for use in applications. It has four arguments an application can use for its specific needs. (Refer to the Javadocs for details.)

Both methods return the object ID of the new version.

### Save and SaveLock methods

Use a `save` or `saveLock` method when you want to overwrite the version that you checked out or fetched. To use either, you must have at least Write permission on the

object. A save method works on either checked-out or fetched objects. A saveLock method works only on checked-out objects.

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

## For more information

- [Attributes that remain changeable, page 164](#), contains a list of the changeable properties in immutable objects.
- [Application-level control of SysObjects, page 127](#), describes application-level control of SysObjects.
- [Concurrent access control, page 165](#), describes the types of locks and locking strategies in detail.
- The CREATE OBJECT and UPDATE OBJECT statements are described in the *DQL Reference Manual*.

## Managing permissions

Access permissions for an object of type SysObject or its subtypes are controlled by the ACL associated with the object. Each object has one associated ACL. An ACL is assigned to each SysObject when the SysObject is created. That ACL can be modified or replaced as needed, as the object moves through its life cycle.

## The default ACLs

If a user or application creates and saves a new object without explicitly assigning an ACL or permissions to the object, Content Server assigns a default ACL. The ACL designated as the default ACL is recorded in the Content Server's server config object, in the default\_acl property. The designated ACL can be any of the following 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.
- In a newly configured repository, the `default_acl` property is set to the value identifying the user's ACL as the default ACL. You can change the setting through Documentum Administrator.

## Template ACLs

A template ACL is identified by a value of 1 in the `acl_class` property of its `dm_acl` object. Template ACL typically use aliases in place of actual user or group names in the access control entries in the ACL. When the template is assigned to an object, Content Server resolves the aliases to actual user or group names.

Template ACLs are used to make applications, workflows, and lifecycles portable. For example, an application that uses a template ACL could be used by a variety of departments within an enterprise because the users or groups within the ACL entries are not defined until the ACL is assigned to an actual document.

## Assigning ACLs

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

### Assigning a default ACL

A Content Server automatically assigns the designated default ACL to new objects if the user or application does not explicitly assign a different ACL or does not explicitly grant permissions to the object. Consequently, a user or application can ensure the assignment of the default ACL simply by saving the object without assigning an ACL to the object or without granting any permissions on the object.

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

## 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 is recommended. 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. There are four common situations that generate a custom ACL:

- [Granting permissions to a new object without assigning an ACL, page 186](#)
- [Modifying the ACL assigned to a new object, page 187](#)
- [Using `grantPermit` when no default ACL is assigned, page 187](#)
- [Modifying the current, saved ACL, page 187](#)

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 assign a custom ACL to multiple objects. Any custom ACL that the server creates for you is owned by you.

## 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 identified in the `default_acl` property of the server config object. It copies that ACL, makes the indicated changes, and then assigns the custom ACL to the object.

## Modifying the ACL assigned to a new object

The server creates a custom ACL when you create a `SysObject`, associate an ACL with the object, and then modify the access control entries in the ACL before saving the object. To identify the ACL to be used as the basis for the custom ACL, use a `useACL` method.

The server copies the specified ACL, applies the changes to the copy, and assigns the new ACL to the document.

## Using `grantPermit` when no default ACL is assigned

The server creates a custom ACL when you create a new document, direct the server not to assign a default ACL, and then use a `grantPermit` method to specify access permissions for the document. In this situation, the object's owner is not automatically granted access to the object. If you create a new document this way, be sure to set the owner's permission explicitly.

To direct the server not to assign a default ACL, you issue a `useacl` method that specifies `none` as an argument.

The server creates a custom ACL with the access control entries specified in the `grantPermit` methods and assigns the ACL to the document. Because the `useacl` method is issued with `none` as an argument, the custom ACL is not based on a default ACL.

## Modifying the current, saved ACL

If you fetch an existing document and change the entries in the associated ACL, the server creates a new custom ACL for the document that includes the changes. The server copies the document's current ACL, applies the specified changes to the copy, and then assigns the new ACL to the document.

## Rooms and ACL assignments

`SysObjects` that are created in moved to a collaborative room, and therefore are governed by the room, are assigned the default ACL for that room. Similarly, if you move an object to a room, the object's current ACL is removed and the room's default ACL is applied.

If the object is moved out of the room, Content Server removes the default room ACL and assigns a new ACL:

- If the user moving the object out of the room is the object's owner, Content Server assigns the default ACL defined in the docbase config to the object.
- If the user moving the object out of the room is not the object's owner, Content Server assigns the object owner's default ACL to the object.

## 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.

Each execution of revokePermit removes a specific entry. To preserve backwards compatibility with pre-5.3 Content Server 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.

If the user or group has access through another entry, the user or group retains that access permission. For example, suppose janek has access as an individual and also as a member of the group engr in a particular ACL. If you issue a revokePermit method for janek against that ACL, you remove only janek's individual access. The access level granted through the engr group is retained.

## Removing permissions to a single document

To remove permissions to a document, call the IDfSysobject revokePermit method against the document. The server copies the ACL, changes the copy, and assigns the new ACL to the document. The original ACL is not changed. The new ACL is a custom ACL.

## 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 revokePermit method against the ACL. 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. Users typically replace an ACL using facilities provided by a client interface. To replace the ACL programmatically, reset the object attributes acl\_name, acl\_domain, or both. These two attributes identify the ACL assigned to an object.

## For more information

- The *Content Server Administration Guide* describes the types of ACLs, types of entries, and how to create ACLs and the entries.

## 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 by opening a session with the remote repository or by working with the mirror or replica object in the current repository that refers to the remote object. Mirror objects and replica objects are implemented as reference links.

## Architecture

A reference link is a pointer in one repository to an object in another repository. A reference link is a combination of a `dm_reference` object and a mirror object or a `dm_reference` object and a replica object.

A mirror object is an object in one repository that mirrors an object in another repository. The term mirror object describes the object'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.

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.

## For more information

- The *Distributed Configuration Guide* has complete information about:
  - Reference links and the underlying architecture that supports them
  - How operations on mirror objects and replicas are handled
  - Object replication

# Relationships between objects

This section provides an overview of relationships, a feature that allows you to establish a formal relationship between two objects in a repository.

## What a relationship is

A relationship is a formal association between two objects in the repository. One object is designated as the parent and one object is designated as the child. Before you can connect two objects in a relationship, the relationship must be described in the repository. Types of relationships are defined in `dm_relation_type` objects and instances of relationship types are recorded in `dm_relation` objects.

The definition of the relationship, recorded in the `dm_relation_type` object, names the relationship and defines some characteristics, such as the security applied to the relationship and the behavior if one of the objects involved in an instance of the relationship is deleted from the repository.

A relation object identifies the two objects involved in the relationship and the type of relationship. Relation objects also have some attributes that you can use to manage and manipulate the relationship.

## System-defined relationships

Installing Content Server installs a set of system-defined relationships. For example, annotations are implemented as a system-defined relationship between a `SysObject`, generally a document, and a note object. Another system-defined relationship is `DM_TRANSLATION_OF`, used to create a relationship between a source document and a translated version of the document.

You can obtain the list of system-defined relationships by examining the `dm_relation_type` objects in the repository. The relation name of system-defined relationships begin with `dm_`.

## User-defined relationships

You can create custom relationships. Additionally, the `dm_relation` object type may be subtyped, so that you can create relationships between objects that record business-specific information if needed.

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.

## For more information

- The *Documentum Object Reference Manual* has complete information about relationships, including instructions for creating relationship types and relationships between objects.
- [Managing translations, page 192](#), describes how the system-defined translation relationship can be used.
- [Annotation relationships, page 193](#), describes annotations and how to work with them.

## 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.

## Translation relationships

You can also use the built-in relationship functionality to create a translation relationship between two SysObjects. Such a relationship declares one object (the parent) the original and the second object (the child) a translation of the original. Translation relationships

have a security type of child, meaning that security is determined by the object type of the translation. A translation relationship has the relation name of DM\_TRANSLATION\_OF.

When you define the child in the relationship, you can bind a specific version of the child to relationship or bind the child by version label. To bind a specific version, you set the child\_id property of the dm\_relation object to object ID of the child. To bind by version label, you set the child\_id attribute to the chronicle ID of the version tree that contains the child, and the child\_label to the version label of the translation. The chronicle ID is the object ID of the first version on the version tree. For example, if you want the APPROVED version of the translation to always be associated with the original, set child\_id to the translation's chronicle ID and child\_label to APPROVED.

## For more information

- The *Documentum Object Reference Manual* has more information about:
  - Recommended language and country codes
  - Properties defined for the relation object type

## Annotation relationships

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 IDfNote.addNoteEx method to associate the note with the document. A single document can have multiple annotations. Conversely, a single annotation can be attached to multiple documents.

When you attach an annotation to a document, the server creates a relation object that records and describes the relationship between the annotation and the document. The relation object'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.

## 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.
4. Use an `addNoteEx` method to attach the annotation to the desired document.

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 the note. 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 `IDfNote.addNoteEx` 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 attribute 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 an `IDfNote.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 have SysAdmin or Superuser privileges or be 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 have SysAdmin or Superuser privileges 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 copy a document, the `permanent_link` attribute for the relation object associated with the annotation must be set to `TRUE`. This flag is `FALSE` by default.

### Destroy

Destroying an object that has attached annotations automatically destroys the relation objects that attach the annotations to the object. The note objects that are the annotations are not destroyed.

**Note:** The dm\_clean utility automatically destroys note objects that are not referenced by any relation object, that is, any that are not attached to at least one object.

## Object replication

If the replication mode is federated, then any annotations associated with a replicated object are replicated also.

## For more information

- The *Documentum Object Reference Manual* has a complete description of relation objects, relation type objects, and their attributes.
- The Javadocs have more information about the addNote and removeNote methods in the IDfSysObject interface.

# 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 DFC.

The chapter covers the following topics:

- [Introducing virtual documents, page 197](#)
- [Virtual document assembly and binding, page 204](#)
- [Defining component assembly behavior, page 205](#)
- [Defining copy behavior, page 207](#)
- [Creating virtual documents, page 208](#)
- [Modifying virtual documents, page 209](#)
- [Assembling a virtual document, page 211](#)
- [Snapshots, page 213](#)
- [Frozen virtual documents and snapshots, page 216](#)
- [Obtaining information about virtual documents, page 217](#)

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 Services](#), before you begin working with virtual documents.

## Introducing virtual documents

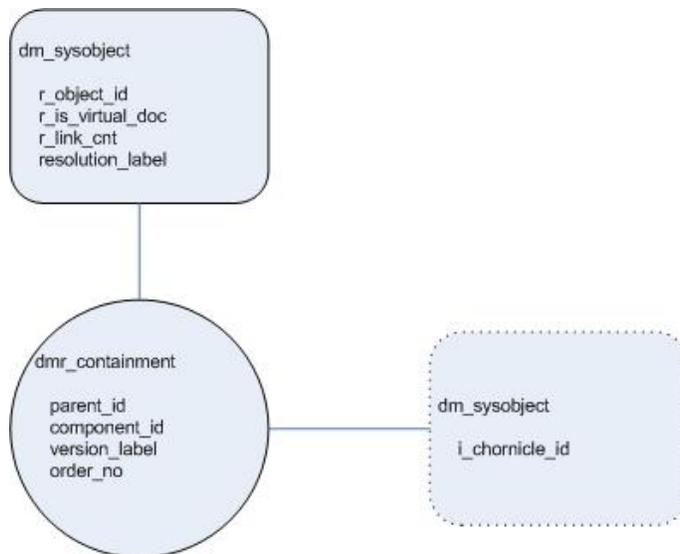
This section describes virtual documents, a feature supported by Content Server that allows you to create documents with varying formats.

## What is a virtual document?

A virtual document is a hierarchically organized structure composed of component documents. The components of a virtual document are of type `dm_sysobject`, or a subtype of `dm_sysobject` (but excluding cabinets and folders). Most commonly, the components are of type `dm_document` or a subtype. The child components of a virtual document can be simple documents (that is, non-virtual documents), or they can themselves be virtual documents. Content server does not impose any restrictions on the depth of nesting of virtual documents.

The root of a virtual document is version-specific and identified by an object identity (on Content Server an `r_object_id`). The child components of a virtual document are not version-specific, and are identified by an `i_chronicle_id`. The relationship between a parent component and its children are defined in containment objects (`dmr_containment`), each of which connects a parent object to a single child object. The order of the children of the parent object is determined by the `order_no` property of the containment object.

**Figure 8. Containment object defines virtual document relationship**



The version of the child component is determined at the time the virtual document is *assembled*. A virtual document is assembled when it is retrieved by a client, and when a snapshot of the virtual document is created. The assembly is determined at runtime by a binding algorithm governed by metadata set on the `dmr_containment` objects.

## Use of virtual documents

Virtual documents provide a way to combine multiple documents in multiple formats into a single document. Each component exists as an independent object in the repository. Virtual documents allow users to:

- Share document components in multiple virtual documents. When a changed component is checked in, the change is reflected in all virtual documents that include the component.
- Combine different types of related content into the same document (as an organizational tool).
- Increase flexibility of user access (multiple users can simultaneously check out and maintain different parts of the virtual document).
- Save snapshots of the virtual document that reflect the state of all components at the time the snapshot is created.

For some content types, such as Microsoft Word files and XML files used in XML applications, virtual documents are patched as they are retrieved to a client, and flattened into a single document. In other cases, the individual components of the virtual documents are retrieved as separate files.

## Implementation

This section briefly describes how virtual documents are implemented within the EMC Documentum system.

### Connecting the virtual document and its components

The components of a virtual document are associated with the containing document by containment objects. Containment objects contain information about the components of a virtual document. Each time you add a component to a virtual document, a containment object is created for that component. Containment objects store the information that links a component to a virtual document. For components that are themselves virtual documents, the objects also store information that the server uses when assembling the containing document.

You can associate a particular version of a component with the virtual document or you can associate the 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.

The components of a virtual document are ordered within the document. By default, the order is managed by the server. The server automatically assigns order numbers when you add or insert a component.

If you bypass the automatic numbering provided by the server, you can use your own numbers. The `insertPart`, `updatePart`, and `removePart` methods allow you to specify order numbers. However, if you define order numbers, you must also perform the related management operations. The server does not manage user-defined ordering numbers.

The number of direct components contained by a virtual document is recorded in the document's `r_link_cnt` property. Each time you add a component to a virtual document, the value of this property is incremented by 1.

## **r\_is\_virtual\_doc property**

`r_is_virtual_doc`

The `r_is_virtual_doc` property is (surprisingly) an integer property that helps determine whether EMC Documentum client applications treat the object as a virtual document. If the property is set to 1, the client applications always open the document in the Virtual Document Manager. The property is usually set to 1 when you use the Virtual Document Manager to add the first component to the containing document. Programmatically, you can set it using the `IDfSysObject.setIsVirtualDocument` method. You can set the property for any `SysObject` subtype except folders, cabinets, and their subtypes.

However, clients will also treat an object as a virtual document if `r_is_virtual_doc` is set to 0, and `r_link_cnt` is greater than 0. So, a document is not a virtual document only when both properties are set to 0. If either property is not 0, the object is treated as a virtual document.

## **Versioning**

You can version a virtual document and manage its versions just as you do a simple document.

## **Deleting virtual documents and components**

Deleting a virtual document version also removes the containment objects and any assembly objects associated with that version.

---

## Referential integrity, freezing, and component deletions

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` property in the server's server config object. By default, this property is `TRUE`, which prohibits users from destroying any object contained in a virtual document.

If you set this property to `FALSE`, users can destroy components of unfrozen virtual documents. However, users can never destroy components of frozen virtual documents, regardless of the setting of `compound_integrity`.

You must have SysAdmin or Superuser privileges to set the `compound_integrity` property.

## Assembling the virtual document

Content Server supports conditional assembly and snapshots for virtual documents. Both are features that allow you to see the document as an assembled whole.

### What conditional assembly is

Assembling a virtual document selects a set of the document's components for publication or some other operation, such as viewing or copying. Conditional assembly lets you identify which components to include. You can include all the components or only some of them. If a component'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 properties in the containment objects.

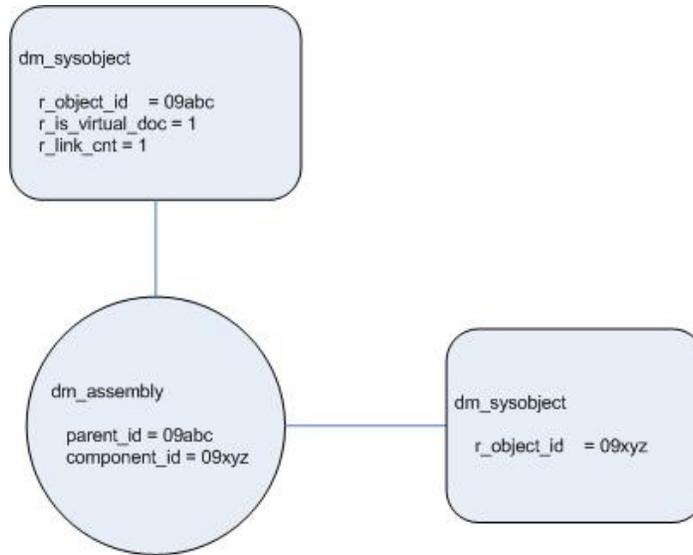
### Snapshots

Snapshots provide a way of persistently storing the results of virtual document assembly. The snapshot records the exact components of the virtual document at the time the snapshot was created, using version-specific object identities to represent each node.

Snapshots are stored in the repository as a set of assembly objects (`dm_assembly`) associated with a `dm_sysobject`. Each assembly object in a snapshot represents one node

of the virtual document, and connects a parent document with a specific version of a child document.

**Figure 9. Assembly object defines assembly relationship**



## 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, the document is created as a virtual document. Other documents referenced in the content of the XML document as entity references or links are automatically brought into the repository and stored as directly contained components of the virtual document.

The connection between the parent and the components is defined in two properties of containment objects: `a_contain_type` and `a_contain_desc`. DFC uses the `a_contain_type` property to indicate whether the reference is an entity or link. It uses the `a_contain_desc` to record the actual identification string for the child.

These two properties are also defined for the `dm_assembly` type, so applications can correctly create and handle virtual document snapshots using the 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, which define rules for handling and chunking the XML content.

## Virtual documents and retention policies

You can associate a virtual document with a retention policy. Retention policies control an object's retention in the repository. They are applied using Retention Policy Services.

If a virtual document is subject to a retention policy, you cannot add, remove, or rearrange its components.

## For more information

- [Versioning, page 157](#) describes how versioning is handled for documents.
- [Defining component assembly behavior, page 205](#), has more information about the properties that control conditional assembly for contained virtual documents.
- [Assembling a virtual document, page 211](#), has more information about the process of assembling a virtual document.
- [Snapshots, page 213](#), has information about creating and working with snapshots.
- [Virtual document assembly and binding, page 204](#), describes how early and late binding work.
- The *XML Application Development Guide* describes XML applications and how to create and manage them. You can find this document through Documentum's Support web site. (Instructions for obtaining documents from the web site are found in the *Content Server Release Notes*)

## Virtual document assembly and binding

A virtual document is assembled when it is retrieved by a client, and when a snapshot of the virtual document is created and stored in the repository.

### Early and late binding

Each virtual document node can be early or late bound.

- In early binding, the binding label is set on the containment object when the node is created and stored persistently. (The binding level is stored in the `version_label` property of the `dmr_containment` object.)
- In late binding, the version of the node is determined at the time the virtual document is assembled, using a “preferred version” or late binding label passed at runtime. (If the `version_label` property of the `dmr_containment` object is empty or null, then the node is late bound.)

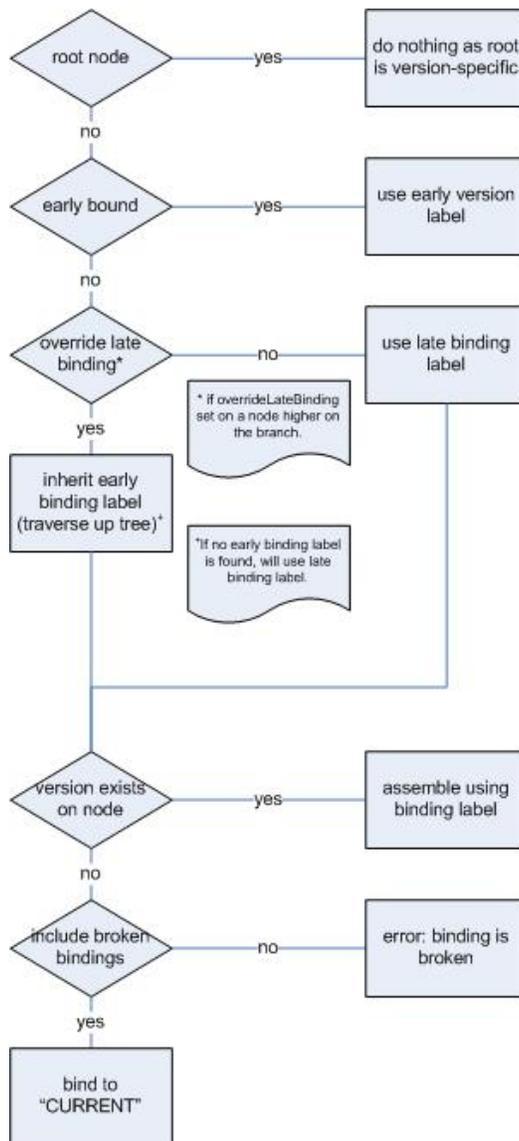
### Binding rules and assembly logic

The logic that controls the assembly of the virtual document at the time it is retrieved is determined by settings on the containment objects.

API term	Content Server property	Description
binding	version_label	The early binding label of the virtual document node. If empty, then the node is late bound.
overrideLate-Binding	use_node_vers_label	Override the late binding value for all descendants of this node, using the early bound label of this node.
includeBroken-Bindings	none (provided by client API at runtime)	A broken binding occurs when there is no version label on the node corresponding to the <code>lateBindingValue</code> ; if broken nodes are include, uses the <code>CURRENT</code> version of the node.

The following diagram shows the decision process when assembling a virtual document node.

Figure 10. Assembly decision tree



## Defining component assembly behavior

There are two properties in containment objects that control how components that are themselves virtual documents behave when the components are selected for a snapshot. The properties are `use_node_ver_label` and `follow_assembly`. In an application, they are set by arguments in `appendPart`, `insertPart`, and `updatePart` methods.

## use\_node\_ver\_label

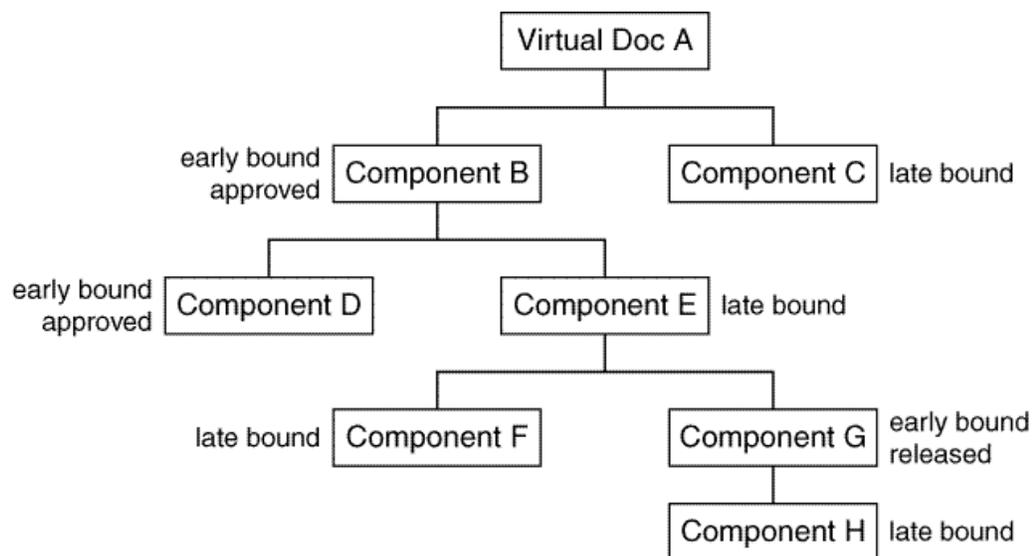
The `use_node_ver_label` property determines how the server selects late-bound descendants of an early-bound component.

If a component is early bound and `use_node_ver_label` in its associated containment object is set to `TRUE`, the server uses the component'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 11. A sample virtual document showing node bindings**



GEN-000926

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` property.

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 a user copies a virtual document, the server can make a copy of each component or it can create an internal reference or pointer to the source component. (The pointer or reference is internal. It is not an instance of a `dm_reference` object.) Which option is used is controlled by the `copy_child` property in a component's containment object. It is an integer property with three valid settings:

- 0, which means that the copy or reference choice is made by the user or application when the copy operation is requested

- 1, which directs the server to create a pointer or reference to the component
- 2, which directs the server to copy the component

Whether the component is copied or referenced, a new containment object for the component linking the component to the new copy of the virtual document is created.

Regardless of which option is used, when users open the new copy in the Virtual Document Manager, all document components are visible and available for editing or viewing (subject to the user's access permissions).

## Creating virtual documents

Virtual documents are typically created using the Virtual Document Manager, a user interface accessible through Webtop. They may also be created programmatically. The basic steps to create a virtual document programmatically are:

1. Obtain the object that you want to use as a virtual document.

Folders and cabinets cannot be virtual documents.

2. Set the object's `r_is_virtual_doc` property.

Setting this property is optional. If users are never going to open or work with the document, setting this property is not necessary. However, setting it ensures that if users do work with the document, the document behaves appropriately.

3. Add components to the object.

Two methods add components to a virtual document: `IDfSysObject.appendPart` and `IDfSysObject.insertPart`. The `appendPart` method adds components to the end of the ordered list of components that make up the virtual document. The `insertPart` method inserts components into the ordered list of components at any location. Note that neither method sets the `r_is_virtual_doc` property. They only increment the `r_link_cnt` property.

4. Save or check in the object.

The permissions required to write the object to the repository vary depending on how it was obtained:

- If you created a new object, use a save method to put the object in the repository.
- 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 one of the checkout methods to obtain the object, use one of the checkin methods 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. Virtual documents are typically modified using Virtual Document Manager. However, it is possible to do so programmatically. This section briefly introduces the permissions needed to change a virtual documents and the methods used to do so programmatically.

### Required permissions

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 `IDfSysObject.appendPart` or `insertPart` method. The `appendPart` method 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.

The `insertPart` method 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 an `IDfSysObject.removePart` method.

## Changing the component order

To change the component order in a virtual document, the application must first remove the component and then use `appendPart` or `insertPart` to place the component in the new desired position.

## 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 properties in the component's containment object: `use_node_ver_label` and `follow_assembly`. The values of these properties are set initially when the component is added to the document. To change their values, and the server's subsequent behavior, use an `IDfSysObject.updatePart` argument.

## Modifying copy behavior

How a component is handled when the containing document is copied is determined by the `copy_child` property in the component's associated containment object. This property is set initially when the component is added to the virtual document. To reset this property, use an `IDfSysObject.updatePart` method.

## For more information

- Refer to the Javadocs for information about the methods used to add or remove components or update a virtual document component.
- [Defining component assembly behavior, page 205](#), describes how the `use_node_ver_label` and `follow_assembly` properties affect assembly behavior.
- [Defining copy behavior, page 207](#), describes the valid settings for the `copy_child` property.

# 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.

## Basic procedure

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.
2. Use the object IDs to get the components from the repository.

After obtaining the components, the application can manipulate the components as needed.

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.

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.

The use\_node\_ver\_label and follow\_assembly properties affect any components that are themselves virtual documents. Both control how Content Server chooses the descendants of such components for inclusion.

## How the SELECT statement is processed

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.
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.

## For more information

- [Defining component assembly behavior, page 205](#), has more information about defining assembly behavior for virtual documents.
- The *DQL Reference Manual* describes the SELECT statement in detail.

## 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.

## How a snapshot is created

Creating a snapshot of a virtual document requires at least Version permission for the virtual document. Typically, snapshots are created through Virtual Document Manager. However, it is possible to create them programmatically. The basic steps are:

1. Use an `IDfSysObject.assemble` method to select the components for the snapshot and place them in a collection.

**Note:** If you include the `interruptFreq` argument in the `assemble` method, you cannot execute the method inside an explicit transaction. (An explicit transaction is a transaction explicitly opened by an application or user.)

2. Execute a `IDfSession.getLastCollection` method to obtain the ID of the collection holding the components.
3. Execute a `IDfCollection.next` method to generate assembly objects for the components.
4. When the `next` method returns a NULL value, execute a `IDfCollection.close` method to close the collection.

The collection must be explicitly closed to 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.

## 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

If you add an assembly object to a snapshot programmatically, be sure to set the following properties of the new assembly object:

- `book_id`, which identifies the topmost virtual document containing this component. Use the document'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 property has an integer datatype. You can query the `order_no` values for existing components to decide which value you want to assign to a new component.

You can add components that are not actually part of the virtual document to the document'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` property 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` property) to delete an assembly object.

To delete a single assembly object or several assembly objects, use a destroy method. Do not use `destroy` to delete each object individually in an attempt to delete the snapshot.

## Changing an assembly object

You can change the values in the properties of an assembly object. However, if you do, be very sure that the new values are correct. Incorrect values can cause errors when you attempt to query the snapshot. (Snapshots are queried using the `USING ASSEMBLIES` option of the `SELECT` statement's `IN DOCUMENT` clause.)

## Deleting a snapshot

Use a `IDfSysObject.disassemble` method to delete a snapshot. This method destroys the assembly objects that make up the snapshot. You must have at least Version permission for a virtual document to destroy its snapshot.

## Frozen virtual documents and snapshots

A frozen virtual document or snapshot is a document that has been explicitly marked as immutable by an `IDfSysObject.freeze` method. Users cannot modify the content or properties of a frozen virtual document or of the frozen snapshot components. Nor can they add or remove snapshot components.

Issuing the freeze method automatically freezes the target virtual document. Freezing the associated snapshot is optional. If the document has multiple snapshots, only the snapshot actually associated with the virtual document itself can be frozen. (The other snapshots, associated with simple documents, are not frozen.)

If you want to freeze only the snapshot, you must freeze both the virtual document and the snapshot and then explicitly unfreeze the virtual document.

Users are allowed to modify any components of the virtual document that are not part of the frozen snapshot. Although users cannot remove those components from the document, they can change the component's content files or properties.

### What freezing does

Freezing sets the following properties of the virtual document to TRUE:

- `r_immutable_flag`  
This property indicates that the document is unchangeable.
- `r_frozen_flag`  
This property indicates that the `r_immutable_flag` was set by a Freeze method (instead of a Checkin method).

If you freeze an associated snapshot, the `r_has_frzn_assembly` property is also set to TRUE.

Freezing a snapshot sets the following properties for each component in the snapshot:

- `r_immutable_flag`
- `r_frzn_assembly_cnt`  
The `r_frzn_assembly` count property contains a count of the number of frozen snapshots that contain this component. If this property is greater than zero, you cannot delete or modify the object.

## Unfreezing a document

Unfreezing a document makes the document changeable again.

Unfreezing a virtual document sets the following properties of the document to FALSE:

- `r_immutable_flag`

If the `r_immutable_flag` was set by versioning prior to freezing the document, then unfreezing the document does not set this property to FALSE. The document remains unchangeable even though it is unfrozen.

- `r_frozen_flag`

If you chose to unfreeze the document's snapshot, the server also sets the `r_has_frzn_assembly` property to FALSE.

Unfreezing a snapshot resets the following properties for each component in the snapshot:

- `r_immutable_flag`

This is set to FALSE unless it was set to TRUE by versioning prior to freezing the snapshot. In such cases, unfreezing the snapshot does not reset this property.

- `r_frzn_assembly_cnt`

This property, which contains a count of the number of frozen snapshots that contain this component, is decremented by 1.

## 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 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 property (*path\_name*) and methods in the *IDfSysObject* interface provide this information.

### The *path\_name* property

The *path\_name* property 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* property 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 property values for these assembly objects would be:

- Mydoc, for the Mydoc component
- Mydoc/BrotherDoc, for the BrotherDoc component
- Mydoc/BrotherDoc/Nephew1Doc, for the Nephew1Doc component
- Mydoc/BrotherDoc/Nephew2Doc, for the Nephew2Doc component
- Mydoc/SisterDoc, for the SisterDoc component

The path\_name property is set during the execution of the next method during assembly processing. If the path is too long for the property'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 property to identify the virtual document in which the path in path\_name is found.

## Using DFC

In DFC, use the IDfSysObject.vdmPath and IDfSysObject.vdmPathDQL methods to 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 more information

- The *Content Server DQL Reference Manual* describes the ID function and its use.

- [How a snapshot is created, page 214](#), describes assembly processing.

## 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:

- [Introducing renditions, page 221](#)
- [Automatically generated renditions, page 223](#)
- [User-generated renditions, page 224](#)
- [The keep argument, page 224](#)
- [Supported conversions on Windows platforms, page 225](#)
- [Supported conversions on UNIX platforms, page 225](#)
- [Implementing an alternate converter, page 228](#)

## Introducing renditions

This section describes renditions and the support for renditions provided by Content Server and EMC Documentum.

### 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 EMC 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.

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

## Content Transformation Services

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

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

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

## 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 EMC 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 Administration Guide* for instructions on obtaining a list of formats and how to modify or add a format.

## Rendition characteristics

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

- Resolution characteristics
- Encapsulation characteristics
- Transformation loss characteristics

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

## For more information

- [Supported conversions on Windows platforms, page 225](#), describes the supported format conversions on Windows platforms.
- [Supported conversions on UNIX platforms, page 225](#), describes the supported format conversions on UNIX platforms.
- [Implementing an alternate converter, page 228](#), contains information about using a converter that you have written or that is not on our supported list.
- *Administering Documentum Media Transformation Services* has information about Media Transformation Services and how to create renditions using `TRANSCODE_CONTENT`.
- The *Content Server DQL Reference Manual* contains reference information for `TRANSCODE_CONTENT`.

## Automatically generated renditions

When an application issues a `getFile` method, the application can specify the rendition of the file that it wants to obtain. If the document has a rendition in that format, Content Server will deliver it to the application. If there is not rendition in that format, but

Content Server can create one, it will do so and deliver that automatically generated rendition to the user.

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

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.

## User-generated renditions

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

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

## The keep argument

When a rendition is added using an `addRenditionEx2` method, the `keep` argument allows you to define how you want to handle the rendition when the containing object is versioned. If you set `keep` to T (TRUE), 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 `keep` is set to T, the `keep` property in the content object representing the rendition is set to 3. To determine whether `keep` is set to T or F, examine that property. If it is set

to 3, the flag was set to T. If the property 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 11, page 225](#), lists the acceptable ASCII text input formats and the obtainable output formats.

**Table 11. 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. This directory contains the following subdirectories:

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

Additionally, Content Server uses UNIX utilities to perform various miscellaneous conversions.

You can also purchase and install document converters. 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., and purchase a separate license.

You can also purchase and install Frame converters from Adobe Systems Inc. If you install the Frame converters in the 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 12, page 226](#), lists the acceptable input formats and [Table 13, page 227](#), lists the acceptable output formats for the PBMPLUS package.

**Table 12. 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
rast	SUN raster image file

Input format	Description
tiff	TIFF graphic file
xbm	xbitmap file (x.11 Windowing system definition)

**Table 13. 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

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 14, page 227](#), lists the acceptable input formats and [Table 15, page 228](#), lists the acceptable output formats for these tools.

**Table 14. 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

Input format	Description
ps	PostScript file
text	ASCII text file
troff	UNIX text file

**Table 15. 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 property called `user_converter_location`. To use an alternate converter, set this property to the name of the location object that identifies the directory where the executable or script for the alternate convertor 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 231](#)
- [Workflow definitions, page 235](#)
- [Validation and installation, page 263](#)
- [Architecture of workflow execution , page 266](#)
- [Package notes, page 270](#)
- [Attachments, page 270](#)
- [The workflow supervisor, page 271](#)
- [The workflow agent, page 271](#)
- [The enable\\_workitem\\_mgmt key, page 272](#)
- [Instance states, page 273](#)
- [Starting a workflow, page 276](#)
- [How execution proceeds, page 276](#)
- [User time and cost reporting, page 292](#)
- [Reporting on completed workflows, page 292](#)
- [Changing workflow, activity instance, and work item states, page 293](#)
- [Modifying a workflow definition, page 295](#)
- [Destroying process and activity definitions, page 298](#)
- [Distributed workflow, page 298](#)

## Introducing workflows

This section provides a brief introduction to EMC Documentum workflows.

## What a workflow is

A workflow is a sequence of activities that represents a business process such as an insurance claims procedure or an engineering development process. Workflows can describe simple or complex business processes. 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

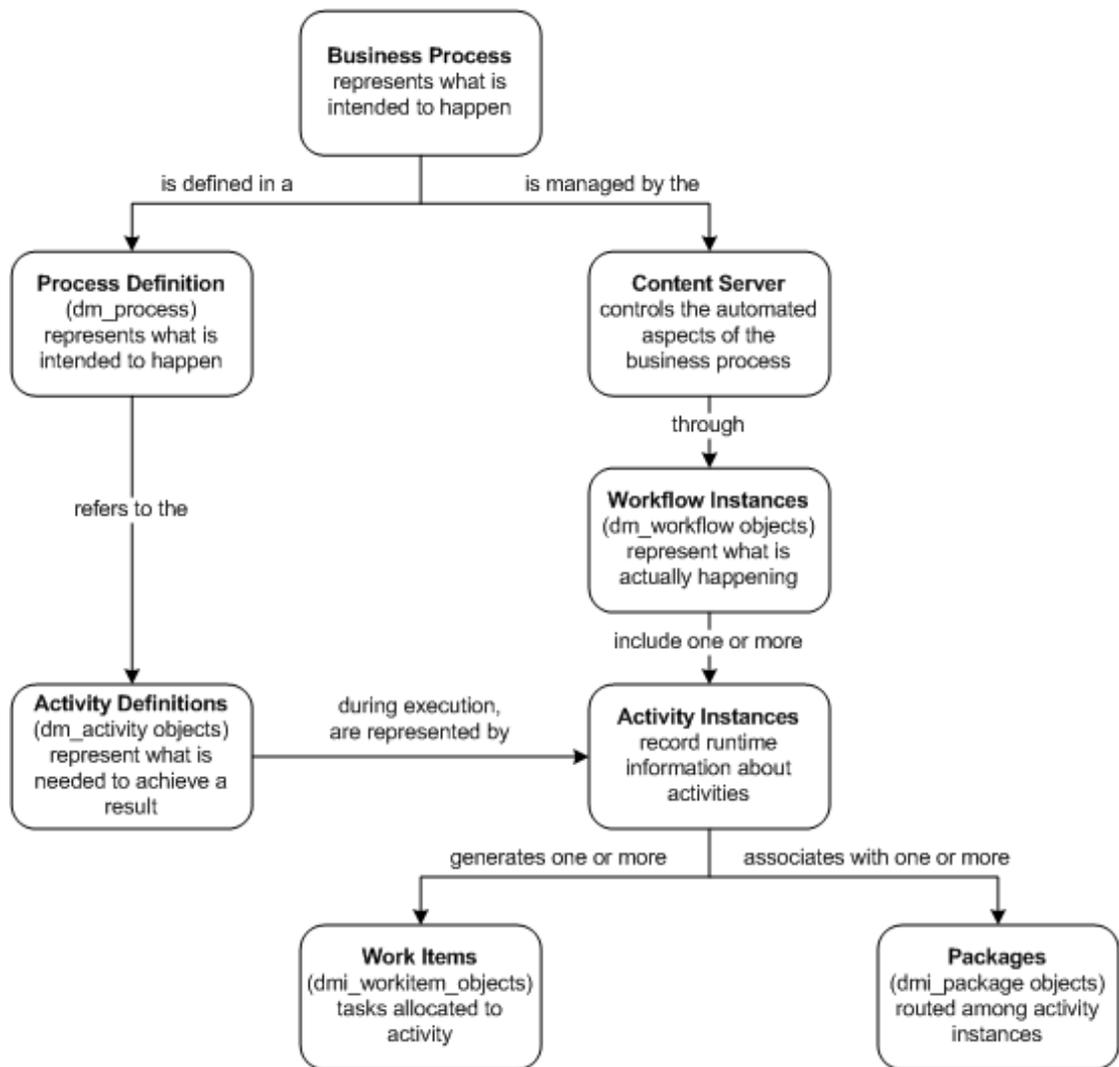
Workflows are implemented as two separate parts: a workflow definition and a runtime instantiation of the definition.

The workflow definition is the formalized definition of the business process. A workflow definition has two major parts, the structural, or process, definition and the definitions of the individual activities. The structural definition is stored in a `dm_process` object. The definitions of individual activities are stored in `dm_activity` objects. Storing activity and process definitions in separate objects allows activity definitions to be used in multiple workflow definitions. When you design a workflow, you can include existing activity definitions in addition to creating any new activity definitions needed.

When a user starts a workflow, the server uses the definition in the `dm_process` object to create a runtime instance of the workflow. Runtime instances of a workflow are stored in `dm_workflow` objects for the duration of the workflow. When an activity starts, it is instantiated by setting properties in the workflow object. Running activities may also generate work items and packages. Work items represent work to be performed on the objects in the associated packages. Packages generally contain one or more documents.

[Figure 12, page 233](#), illustrates how the components of a workflow definition and runtime instance work together.

Figure 12. Components of a workflow



Users can repeatedly perform the business process, and because it is based on a stored definition, the essential process is the same each time. Separating a workflow's definition from its runtime instantiation allows multiple workflows based on the same definition to be run concurrently.

## 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).

## Process Builder and Workflow Manager

Content Server supports two user interfaces for creating and managing workflows: Process Builder and Workflow Manager.

Workflow Manager (WFM) supports basic workflow functionality. Process Builder, which requires an additional license, supports the basic functionality and additional features not supported by WFM. The additional features supported in Process Builder are:

- Ability to define INITIATE and EXCEPTION activities
- Ability to associate a form with an activity.
- Global package definitions
- Ability to associate structured data with workflows, to allow metadata to be recorded and managed as part of the workflow
- Ability to associate correlation sets with a workflow, to allow the workflow engine to communicate with external applications at runtime
- Ability to conditionally define a performer for certain performer categories
- 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 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.

## For more information

- [Workflow definitions, page 235](#), provides more information about workflow definitions, including activity, port, and package definitions.
- [Using aliases as performer names, page 248](#), describes how aliases are used in workflows.

## 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 properties that identify the activities that make up the business process, a set of properties that define the links connecting the activities, and a set of properties that define the structured data elements and correlation sets that may be associated the workflow. It also has properties that define some behaviors for the workflow when an instance is running.

**Note:** Structured data elements and correlation sets for a workflow may only be defined using Business Process Manager. Refer to that documentation for more information about these features.

## 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.

Each activity in a workflow is defined as one of the following kinds of activities:

- **Initiate**

Initiate activities link to a Begin activity. These activities record how a workflow may be started; for example, a workflow may have two Initiate activities, one that allows the workflow to be started manually, from Webtop and one that allows the workflow to be started by submitting a form. Initiate activities may only be linked to Begin activities. Initiate activities may only be defined for a workflow using Process Builder.
- **Begin**

Begin activities start the workflow. A process definition must have at least one beginning activity.
- **Step**

Step activities are the intermediate activities between the beginning and the end. A process definition can have any number of Step activities.
- **End**

An End activity is the last activity in the workflow. A process definition can have only one ending activity.
- **Exception**

An exception activity is associated with an automatic activity, to provide fault-handling functionality for the activity. Each automatic activity may have one exception activity.

## Multiple use of activities in process definitions

You can use activity definitions more than once in a workflow definition. For example, suppose you want all documents to receive two reviews during the development cycle. You might design a workflow with the following activities: Write, Review1, Revise, Review2, and Publish. The Review1 and Review2 activities can be the same activity definition.

An activity that can be used more than once is called a repeatable activity. Whether an activity is repeatable is defined in the activity's definition.

## 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` property. If the activity is used only once in the workflow structure, you can use the name assigned to the activity when the activity was defined (recorded in the activity's `object_name` property). However, if the activity is used more than once in the workflow, 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:** An input port on a Begin activity participates in a link, it may only connect to an output port of an Initiate activity. Similarly, an output port of an Initiate activity may only connect to an input port of a Begin activity.

Output ports on End activities are not allowed to participate in links.

Each link in a process definition has a unique name.

## Activity definitions

Activity definitions describe tasks in a workflow. Documentum implements activity definitions as `dm_activity` objects. The properties of an activity object describe the characteristics of the activity, including:

- How the activity is executed
- Who performs the work
- What starts the activity
- The transition behavior when the activity is completed

The definition also includes a set of properties that define the ports for the activities, the packages that each port can handle and the structured data that is accessible to the activity.

## Manual and automatic activities

An activity is either a manual activity or an automatic activity.

## Manual activities

A manual activity represents a task performed by an actual person or persons. Manual activities can allow delegation or extension.

Any user can create a manual activity.

## Automatic activities

An automatic activity represents a task whose work is performed, on behalf of a user, by a script defined in a method object.

Automatic activities cannot be delegated or extended. Additionally, you must have Sysadmin or Superuser privileges to create an automatic activity.

If the method executed by the activity is a Java method, you can configure the activity so that the method is executed by the `dm_bpm` servlet. This is a Java servlet dedicated to executing workflow methods. To configure the method to execute in this servlet, you must set the `a_special_app` property of the method object to a character string beginning with "workflow". Additionally, the classfile of the Java method must be in a location that is included in the classpath of the `dm_bpm_servlet`.

If a Java workflow method is not executed by the `dm_bpm_servlet`, it is executed by the Java method server.

**Note:** The `dm_server_config.app_server_name` for the `dm_bpm_servlet` is `do_bpm`. The URL for the servlet is in the `app_server_uri` property, at the corresponding index position as `do_bpm` in `app_server_name`.

## For more information

- [Delegation and extension, page 240](#), describes delegation and extension.
- The *Content Server Administration Guide* contains instructions for creating a method for an automatic activity.

## Activity priorities

Priority values are used to designate the execution priority of an activity. Any activity may have a priority value defined for it in a process definition that contains the activity. An activity assigned to a work queue may have an additional priority assigned that is specific to the work queue. The uses of these two priority values are different.

---

**Note:** A work queue can be chosen as an activity performer only if the workflow definition was created in Process Builder.

## Use of the priority defined in the process definition

When you create a workflow definition in either WFM or Process Builder, you can set a priority for each activity in the workflow. The priority value is recorded in the process definition, in the `r_act_priority` property. This value 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 the priority values in `r_act_priority` 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 an `IDfWorkitem.setPriority` method.

## Use of the work queue priority values

In Process Builder, 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 Process Builder 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. Every work item on a work queue is assigned a priority value at runtime, when the work item is generated.

The priority assigned by a 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

- For more information about how the workqueue policy is handled at runtime, refer to Process Builder documentation.

## Delegation and extension

Delegation and extension are features that you can set for manual activities.

### Delegation

Delegation allows the server or the activity's performer to delegate the work to another performer. If delegation is allowed, it can occur automatically or be forced manually.

Automatic delegation occurs when the server checks the availability of an activity'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` property 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` property of the activity object that generated the work item. If `control_flag` is set to 0 and automatic delegation fails, the work item is assigned to the workflow supervisor. If `control_flag` is set to 1, the work item is reassigned to the original performer. The server does not attempt to delegate the task again. In either case, the workflow supervisor receives a `DM_EVENT_WI_DELEGATE_F` event.

Manual delegation occurs when an `IDfWorkitem.delegateTask` method is explicitly issued. Typically, only the work item's performer, the workflow supervisor, or a Superuser can execute the method. However, if the `enable_workitem_mgmt` key in the `server.ini` file is set to T (TRUE), any user can issue a `delegateTask` 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

Extension allows the activity's performer to identify a second performer for the activity after he or she completes the activity the first time. If extension is allowed, when the original performers complete 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 an `IDfWorkitem.repeat` method.

If extension is disallowed, only the workflow supervisor or a Superuser can extend the work item.

Activities with multiple performers performing sequentially (user category 9), cannot be extended.

## 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` property controls this feature. It is `TRUE` by default. To constrain an activity's use to only once in a workflow's structure, the property must be set to `FALSE`.

## Performer choices

When you define a performer for an activity, you must first choose a performer category. Depending on the chosen category, you may also be required to identify the performer also. If so, you can either define the actual performer at that time or configure the activity to allow the performer to be chosen at one of the following times:

- When the workflow is started
- When the activity is started
- When a previous activity is completed

If you choose to define the performer during the design phase, Process Builder allows you to either name the performer directly for many categories or define a series of conditions and associated performers. At runtime, the workflow engine determines which condition is satisfied and selects the performer defined as the choice for that condition.

## 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 16, page 242](#), 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` property.

**Table 16. 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 workflow_disabled set to TRUE, 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 workflow_disabled set to TRUE, 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
Performer of previous activity (2)	If the activity is created in Process Builder, choosing this category allows you to define which activity is used as the previous activity whose performer is selected to perform the activity.  If the activity is created in WFM, the server gets the performer from the last finished activity that satisfied the trigger condition of the current activity.  If the selected performer has workflow_disabled set to TRUE, the server gives the work item to the user's delegated user.  If the activity is an automatic activity, you must be a superuser to assign this performer.	manual activities only

User category (integer value)	How performers are selected	Valid category for
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>You may define conditions to choose the performer for this category if you are creating the activity using Process Builder.</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>You may define conditions to choose the performer for this category if you are creating the activity using Process Builder.</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 property.</p> <p>You may define conditions to choose the performer for this category if you are creating the activity using Process Builder.</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 property.</p> <p>You may define conditions to choose the performer for this category if you</p>	manual activities only

User category (integer value)	How performers are selected	Valid category for
	are creating the activity using Process Builder.	
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> <p>You may define conditions to choose the performer for this category if you are creating the activity using Process Builder.</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</p>	manual activities only

User category (integer value)	How performers are selected	Valid category for
	<p>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> <p>Activities with this performer type cannot be extensible.</p>	
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>You may define conditions to choose the performer from the queue users if you are creating the activity using Process Builder.</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 Process Builder 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 16, page 242](#). 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 Process Builder 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 Process Builder 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 Process Builder, you can select multiple activities and make a performer choice that is applied to all the selected activities.

## Using aliases as performer names

This section describes how aliases are used to represent activity performers.

### Usage

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.

## 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 properties 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 Process Builder 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* property 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* property 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` property 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.

### For more information

- [Appendix A, Aliases](#), for a complete description of aliases and how they may be used in workflows and other contexts.

## Task subjects

The task subject is a message that provides a work item performer with information about the work item. The message is defined in the activity definition, using references to one or more properties. At runtime, the actual message is constructed by substituting the actual property values into the string. For example, suppose the task subject is defined as:

```
Please work on the {dmi_queue_item.task_name} task
(from activity number {dmi_queue_item.r_act_seqno})
of the workflow {dmi_workflow.object_name}.
The attached package is {dmi_package_r_package_name}.
```

Assuming that `task_name` is "Review", `r_act_seqno` is 2, `object_name` is "Engr Proposal", and `r_package_name` is "First Draft", at 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` property of the activity definition. The text can be up to 255 characters and can contain references to the following object types and properties:

- `dm_workflow`, any property
- `dmi_workitem`, any property

At runtime, references to `dmi_workitem` are interpreted as references to the work item associated with the current task.

- `dmi_queue_item`. any property except `task_subject`

At runtime, references to `dmi_queue_item` are interpreted as references to the queue item associated with the current task.

- `dmi_package`, any property

The format of the object type and property references must be:

```
{object_type_name.property_name}
```

The server uses the following rules when resolving the string:

- The server does not place quotes around resolved object type and property references.
- If the referenced property is a repeating property, the server retrieves all values, separating them with commas.
- If the constructed string is longer than 512 characters, the server truncates the string.
- If an object type and property reference contains an error, for example if the object type or property does not exist, the server does not resolve the reference. The unresolved reference appears in the message.

The resolved string is stored in the `task_subject` property of the task's associated queue item object. Once the server has created the queue item, the value of the `task_subject` property in the queue item will not change, even if the values in any referenced properties change.

## 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 properties in activity definitions.

### 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
- Exception

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.

An exception port is an output port that links an automatic activity to the input port of an Exception activity. Exception ports do not participate in transitions. The port is triggered only when the automatic activity fails. You must create the workflow definition using Process Builder to define exception ports and Exception activities.

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, the port must either be linked to the output port of an Initiate activity or 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. (Initiate activities may only be defined through Process Builder.)

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. An 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 properties of the activity definition.

Each port must have at least one associated package definition, and may have multiple package definitions. When an activity is completed and a transition to the next activity occurs, 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 a manual activity's route case conditions for an automatic transition, the schema is used to validate the path. 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 Process Builder, 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.

## 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 Process Builder to create the workflow, a package definition is global. When you define a package in Process Builder, 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:** Process Builder 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.

## 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, Process Builder 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 Process Builder.

## 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.

## For more information

- [How activities accept packages, page 288](#), describes how the implementation actually moves packages from one activity to the next.
- [Transition types, page 256](#), describes automatic transitions and route cases in detail.
- The Documentum Process Builder documentation or online help describes how to use those features, such as visibility and skill levels for packages, that are only available through Documentum Process Builder.

## 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

## 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 Process Builder 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` property 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)

- 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.

## 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 Process Builder 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` property.

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` property. If the number of selected ports exceeds the value in the property, 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` property 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 properties 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 property, the property must have at least one value or the condition generates an error when evaluated.

The expression can reference properties from the workflow object, the work item object, or any package, visible or invisible, associated with the activity. If you are using Process Builder 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` property 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` property, a repeating property, 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` property 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 have both the `r_condition_id` and the `r_predicate_id` properties 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` properties. These are repeating properties. The `r_condition_name` property 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` property in the activity definition.

## For more information

- [When the activity is complete, page 287](#), provides more information about how activity transitions work.

## 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 Process Builder. 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 Process Builder, 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 properties 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.

## Suspend timers

**Note:** Suspend timers are only supported if you are using Process Builder 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 properties 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 more information

- The *Content Server Administration Guide* has instructions about activating jobs.
- [How activity timers work, page 289](#), has information about how warning and suspend timers execute.

## Package control

This section describes package control, an optional feature.

### What package control is

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 property of the generated package or wf attachment object. If package control is enabled, Content Server sets the r\_component\_name property to a single blank even if the component names are specified in the methods.

### Implementation

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` property to 1.

## For more information

- The *Content Server Administration Guide* describes how to enable or disable package control at the repository level.

## 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

Activity and process definitions must be validated and installed before users can start a workflow based on the definitions.

## 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

## 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` properties have valid values.

Validating a process definition verifies that:

- The referenced activities have unique names within the process
- There is at least one Begin activity and only one End activity
- There is a path from each activity to the End activity
- All referenced `dm_activity` objects exist and are in the validated or installed state and that they are local objects
- All activities referenced by the link definitions exist
- The ports identified in the links are defined in the associated activity object
- There are no links that reference an input port of a Begin step and no links that reference an output port of an End step
- The ports are connectable and that each port participates in only one link

## 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 more information

- [Package compatibility, page 254](#), describes the rules for package compatibility.

## 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, do not use the information in this section.

The process and activity definitions of a workflow definition must be installed before a workflow can be started from the definition.

## Implementation

A process or activity definition must be in the validated state before you install it.

You can install activity definitions individually, before you install the process definition, or concurrently with the process definition. You cannot install a process definition that contains uninstalled activities unless you install the activities concurrently. If you install only the process, the activities must be in the installed state.

Installing activity definitions and process definitions requires either:

- Relate permission on the process or activity definition
- Sysadmin or Superuser privileges

## For more information

- [Reinstalling after making changes, page 296](#), contains instructions for re-installing a modified workflow definition that has running instances.
- Refer to the Javadocs for information about the methods that install process and activity 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 property). This person functions much like the owner of an object, with the ability to change the workflow's properties and change its state.

## Activity instances

A workflow object contains properties that describe the activities in the workflow. These properties are set automatically, based on the workflow definition, when the workflow object is created. They are repeating properties, and the values at the same index position across the properties represent one activity instance.

The properties that make up the activity instance identify the activity, its current state, its warning timer deadlines (if any), and a variety of other information. As the workflow executes, the values in the activity instance properties change to reflect the status of the activities at any given time in the execution.

## For more information

- [The workflow supervisor, page 271](#), describes the workflow supervisor.
- The *Documentum Object Reference Manual* provides a full list of the properties 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 properties 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 properties 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` property is set.

## 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` property identifies its peer queue item, and the `item_id` property in the queue item object identifies its underlying, associated work item. Work items for automatic activities do not have peer queue item objects.

## 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.

Users typically acquire a work item by selecting and opening the associated Inbox task. Internally, an `acquire` method is executed when a user acquires a work item. Acquiring a work item sets the work item'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 an `IDfWorkitem.delegateTask` method. Adding a note to a package is implemented using an `IDfPackage.appendNote` method. Removing a note is implemented using a `removeNote` method. And sending the work item to additional performers is implemented using an `IDfWorkitem.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.

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, a `complete` method is executed when a work item is designated as completed.

Executing a `complete` method updates the workflow's `r_last_performer` and `r_complete_witem` properties. Updating the `r_complete_witem` property 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` property 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` properties. 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 more information

- The *Documentum Object Reference Manual* lists the properties in the `dmi_workitem` and `dmi_queue_item` object types.
- [The `enable\_workitem\_mgmt` key, page 272](#), describes workflow security.
- [Signature requirement support, page 139](#), describes the options for signing off work items.

## Package objects

Packages contain the objects on which the work is performed. Packages are implemented as `dmi_package` objects. A package object's properties:

- Identify the package and its contained objects
- Record the activity with which the package is associated
- Record when the package arrived at the activity
- Record information about any notes attached to the package

(At runtime, an activity'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.

## 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 properties, 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` property of the workflow object.

## The workflow agent

This section introduces the workflow agent.

### What the workflow agent is

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.

## Implementation

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 more information

- The *Content Server Administration Guide* has instructions on tracing or disabling the agent, as well as instructions on changing the number of worker sessions and the sleep interval.

## 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.

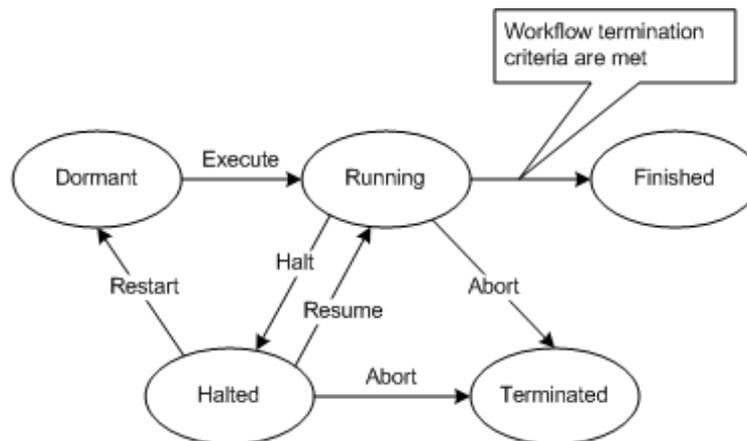
### Workflow states

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` property of the `dm_workflow` object.

The state transitions are driven by API methods or by the workflow termination criterion that determines whether a workflow is finished.

Figure 13, page 273, illustrates the states.

**Figure 13. 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 `IDfWorkflow.abort` method. If the workflow terminates normally, its state is set to finished. If the workflow is manually terminated with the `abort` method, its state is set to terminated.

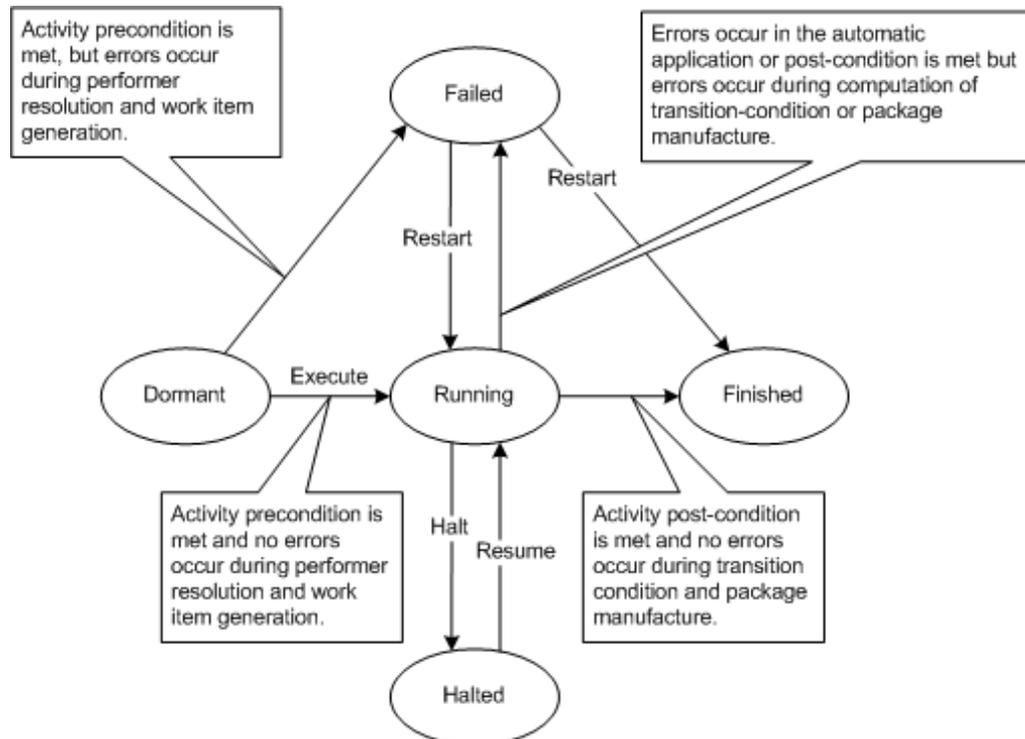
A supervisor can halt a running workflow, which changes the workflow's state to halted. From a halted state, the workflow's supervisor can restart, resume, or abort the workflow.

## 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` property of the `dm_workflow` object, as part of the activity instance.

Figure 14, page 274, illustrates the activity instance states and the operations or conditions that move the instance from one state to another.

**Figure 14. 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.

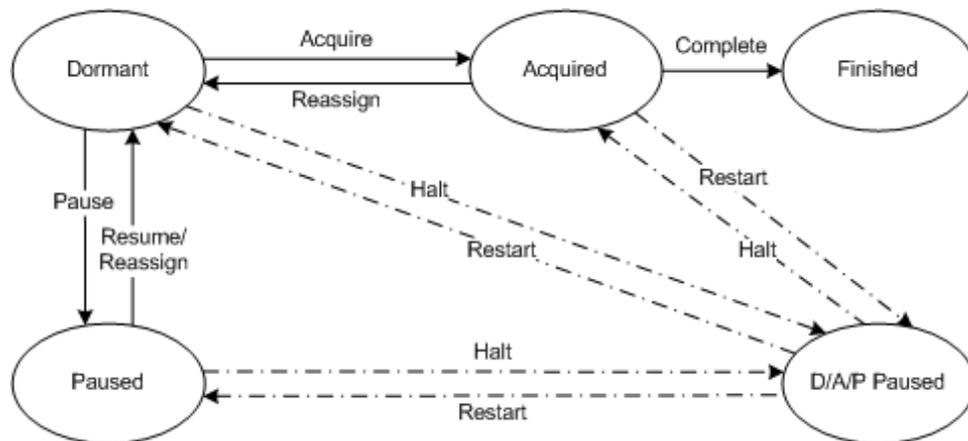
A running activity can be halted. Halting an activity sets its state to halted. By default, only the workflow supervisor or a user with Sysadmin or Superuser privileges can halt or resume an activity instance. However, if `enable_workitem_mgmt`, a `server.ini` key, is set to T (TRUE), any user can halt 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.

## Work item states

A work item exists in one of the following states: dormant, paused, acquired, or finished. [Figure 15, page 275](#), shows the work item states and the operations that move the work item from one state to another.

**Figure 15. Work item states**



A work item's state is recorded in the `r_runtime_state` property of the `dmi_workitem` object.

When the server generates a work item for a manual activity, it sets the work item'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. 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.

## For more information

- [Changing workflow, activity instance, and work item states, page 293](#), describes all options for changing the state of a halted workflow.
- [How activity timers work, page 289](#), describes how suspension intervals are implemented.
- [Completing work items, page 268](#), describes completing a task.

## 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.

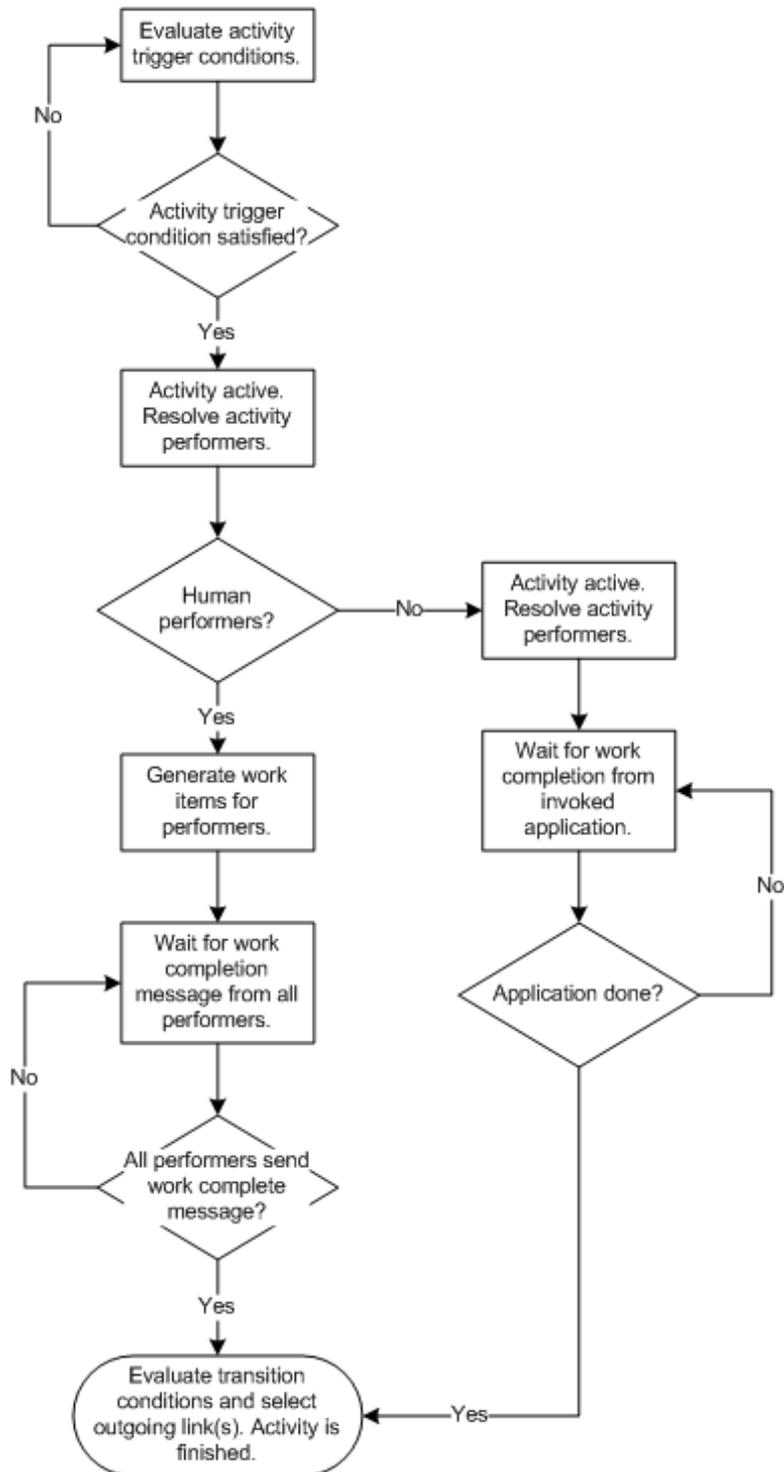
Saving the new workflow object requires Relate permission on the process object (the workflow definition) used as the workflow's template. The execute method must be issued by the workflow's creator or supervisor or a user with Sysadmin or Superuser privileges. If the user is starting the workflow through a Documentum client interface, such as Webtop, the user must be defined as a Contributor also.

## How execution proceeds

This section describes how a typical workflow executes. It describes what happens when a workflow is started and how execution proceeds from activity to activity. It also describes how packages are handled and how a warning timer behaves during workflow execution.

Figure 16, [page 278](#), illustrates the general execution flow described in detail in the text of this section.

Figure 16. 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 property for those activity instances that have pre-timers defined
- Examines the starting condition of each Begin activity and, if the starting condition is met:
  - Sets the r\_post\_timer property for the activity instance if a post timer is defined for the activity
  - Resolves performers for the activity
  - Generates the activity's work items
  - Sets the activity's state to active
- Records the workflow's start time

After the execute method returns successfully, the workflow's execution has begun, starting with the Begin activities.

## Activity execution starts

For Begin activities, execution begins when an execute method is executed for the workflow. The starting condition of a typical Begin activity with no input ports is always considered fulfilled. If a Begin activity has input ports, the application or user must use an addPackage method to pass the required packages to the activity through the workflow. When the package is accepted, the server evaluates the activity's starting condition just as it does for Step and End activities.

For Step and End activities, execution begins when a package arrives at one of the activity's input ports. If the package is accepted, it triggers the server to evaluate the activity's starting condition.

**Note:** For all activities, if the port receiving the package is a revert port and the package is accepted, the activity stops accepting further packages, and the server ignores the starting condition and immediately begins resolving the activity's performers.

After the server determines that an activity's starting condition is satisfied, it consolidates packages if necessary. Next, the server determines who will perform the work and generates the required work items. If the activity is an automatic activity, the server queues the activity for starting.

## 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` properties in the activity definition. When a workflow is created, these values are copied to the `r_trigger_threshold` and `r_trigger_event` properties in the workflow object.

When an activity's input port accepts a package, the server increments the activity instance's `r_trigger_input` property in the workflow object and then compares the value in `r_trigger_input` to the value in `r_trigger_threshold`.

If the two values are equal and no trigger event is required, the server considers that the activity has satisfied its starting condition. If a trigger event is required, the server will query the `dmi_queue_item` objects to determine whether the event identified in `r_trigger_event` is queued. If the event is in the queue, then the starting condition is satisfied.

If the two values are not equal, the server considers that the starting condition is not satisfied.

The server also evaluates the starting condition each time an event is queued to the workflow.

After a starting condition that includes an event is satisfied, the server removes the event from the queue. (If multiple activities use the same event as part of their starting conditions, the event must be queued for each activity.)

When the starting condition is satisfied, the server consolidates the accepted packages if necessary and then resolves the performers and generates the work items. If it is a manual activity, the server places the work item in the performer'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` property.

## 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` property in conjunction with the `performer_name` property, if needed, to determine the activity's performer. After the performer is determined, the server generates the necessary work items and peer queue items.

If the server cannot assign the work item to the selected performer because the performer has `workflow_disabled` set to `TRUE` in his or her user object, the server attempts to delegate the work item to the user listed in the `user_delegation` property of the performer's user object.

If automatic delegation fails, the server reassigns the work item based on the setting of the `control_flag` property in the definition of the activity that generated the work item.

**Note:** When a work item is generated for all members of a group, users in the group who are workflow disabled do not receive the work item, nor is the item assigned to their delegated users.

If the server cannot determine a performer, a warning is sent to the performer who completed the previous work item and the current work item is assigned to the supervisor.

### Automatic activities

The server uses the value in the `performer_type` property in conjunction with the `performer_name` property, if needed, to determine the activity's performer. The server passes the name of the selected performer to the invoked program.

The server generates work items but not peer queue items for work items representing automatic activities.

## Resolving aliases

When the `performer_name` property contains an alias, the server resolves the alias using a resolution algorithm determined by the value found in the activity's `resolve_type` property.

If the server cannot determine a performer, a warning is sent to the workflow supervisor and the current work item is assigned to the supervisor.

## For more information

- [Table 16, page 242](#), lists the valid values for `performer_type` and the performer selection they represent.
- [Delegation, page 240](#), describes how delegation works in detail.
- [Executing automatic activities, page 282](#), describes how automatic activities are executed.
- [Resolving aliases in workflows, page 353](#), describes the resolution algorithms for performer aliases.

## Executing automatic activities

The master session of the workflow agent controls the execution of automatic activities. The workflow agent is an internal server facility.

## 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` property 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.

## 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. It also fetches some related objects, such as the workflow. If any of the objects cannot be fetched or if the fetched workflow is not running, the worker session sets `a_wq_name` to a message string that specifies the problem and drops the task without processing it. Setting `a_wq_name` also ensures that the task will not be picked up again.

After all the fetches succeed and after verifying the ready state of the activity, the worker thread executes the method associated with the activity. The method is always executed as the server regardless of the `run_as_server` property setting in the method object.

**Note:** If the activity is already locked, the worker session assumes that another workflow agent is executing the activity. The worker session simply skips the activity and no error message is logged. (This can occur in repositories with multiple servers, each having its own workflow agent.)

If an activity fails for any reason, the selected performer receives a notification.

The server passes the following information to the invoked program:

- Repository name
- User name (this is the selected performer)

- Login ticket
- Work item object ID
- Mode value

The information is passed in the following format:

```
-docbase_name repository_name -user user_name -ticket login_ticket  
-packageId workitem_id mode mode_value
```

The mode value is set automatically by the server. [Table 17, page 284](#), lists the values for the mode parameter:

**Table 17. 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 method's program can use the login ticket to connect back to the repository as the selected performer. The work item object ID allows the program to query the repository for information about the package associated with the activity and other information it may need to perform its work.

## For more information

- [The workflow agent, page 271](#), describes the workflow agent.
- The *Content Server Administration Guide* provides instructions for recovering work items for execution by an alternate workflow agent in case of a Content Server failure.

## Evaluating the activity's completion

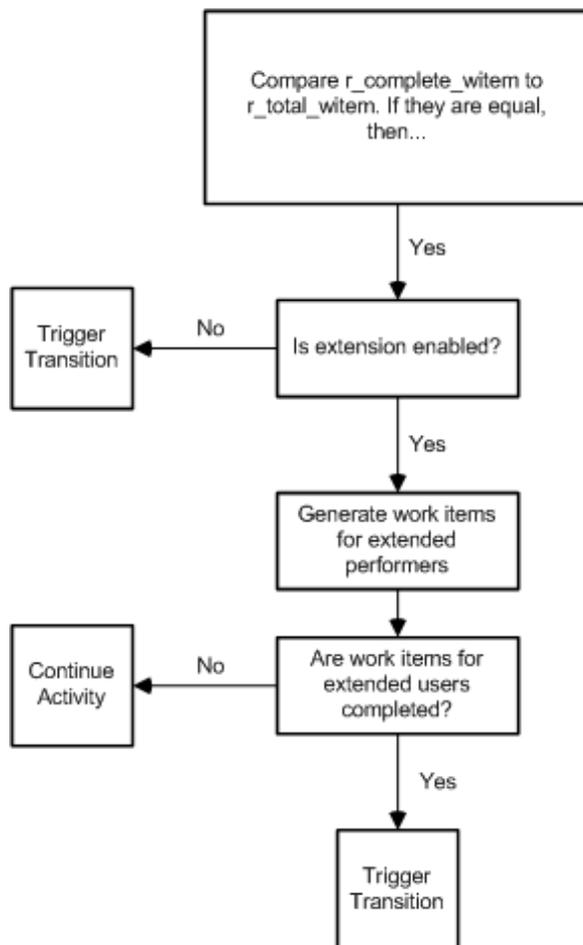
When a performer completes a work item, the server increments the `r_complete_witem` property in the workflow object and then evaluates whether the activity is complete. To do so, the server compares the value of the `r_complete_witem` property to the value in the workflow's `r_total_workitem` property. The `r_total_witem` property records the total number of work items generated for the activity. The `r_complete_witem` property records how many of the activity'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` property of all generated work items
- Generates another set of work items for the user or users designated as the second-round performers and removes the first round of work items
- Sets `i_performer_flag` to indicate that the activity is in the extended mode and no more extension is allowed

Figure 17, page 285, illustrates the decision process when the properties are equal.

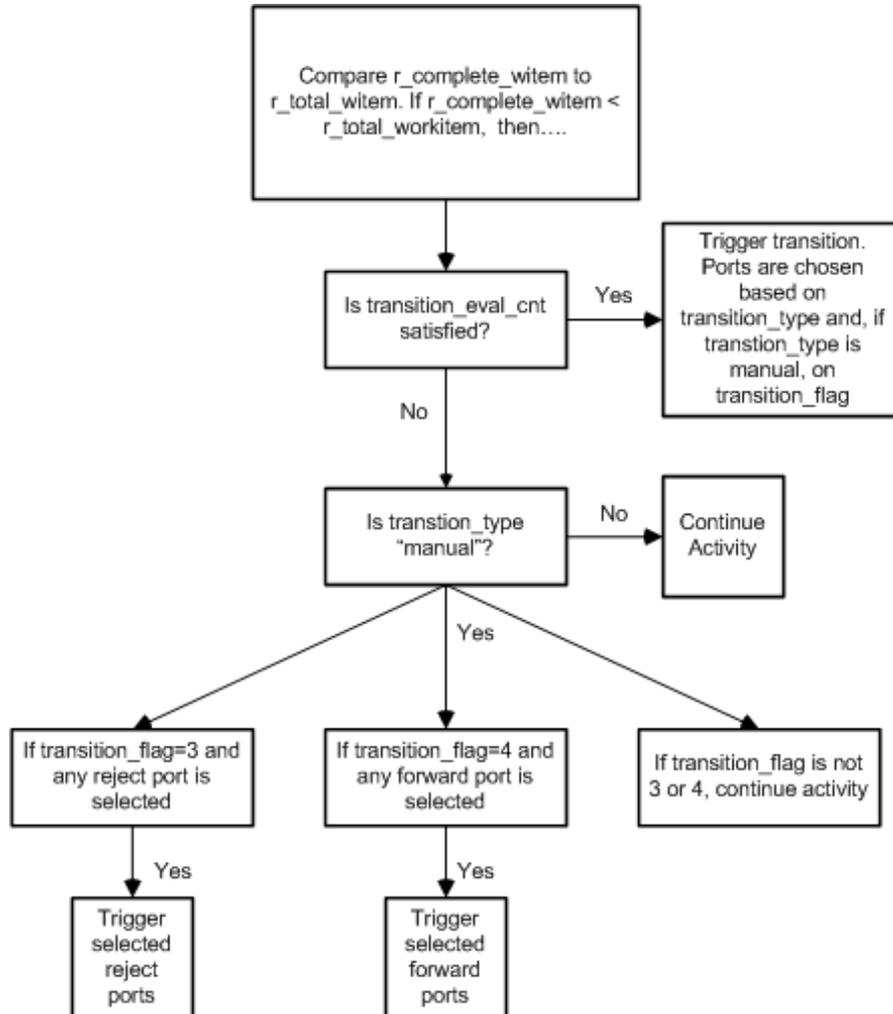
**Figure 17. Behavior if `r_complete_witem` equals `r_total_workitem`**



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` property to determine whether to trigger a transition. The `transition_eval_cnt` property specifies how many work items must be completed

to finish the activity. The `transition_flag` property defines how ports are chosen for the transition. Figure 18, page 286, illustrates the decision process when `r_complete_witem` and `r_total_workitem` are not equal.

**Figure 18. 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.

### Port selection behavior for each transition type

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 one of the Setoutput methods. The number of choices may be limited by the activity's definition. For example, the activity definition may only allow a performer to choose two output ports. How the selected ports are used is also specified in the activity's definition. For example, if multiple ports are selected, the definition may require the server to send packages to the selected revert ports and ignore the forward selections.

If the transition type is automatic, the route cases are evaluated to determine which ports will receive packages. If the activity's `r_condition_id` property is set, the server evaluates the route cases. If the activity's `r_predicate_id` property is set, the server invokes the `dm_bpm_transition` method to evaluate the route cases. The `dm_bpm_transition` method is a Java method that executes in the Java method server. The server selects the ports associated with the first route case that returns a TRUE value.

After the ports are determined, the server creates the needed package objects. If the package creation is successful, the server considers that the activity is finished. At this point, the cycle begins again with the start of the next activity's execution.

### For more information

- [Limiting output choices in manual transitions, page 257](#), describes how output port selection choices are limited.
- [Setting preferences for output port use in manual transitions, page 257](#), describes how processing the selected port choices is defined.
- [Route cases for automatic transitions, page 258](#), describes how route cases are defined.

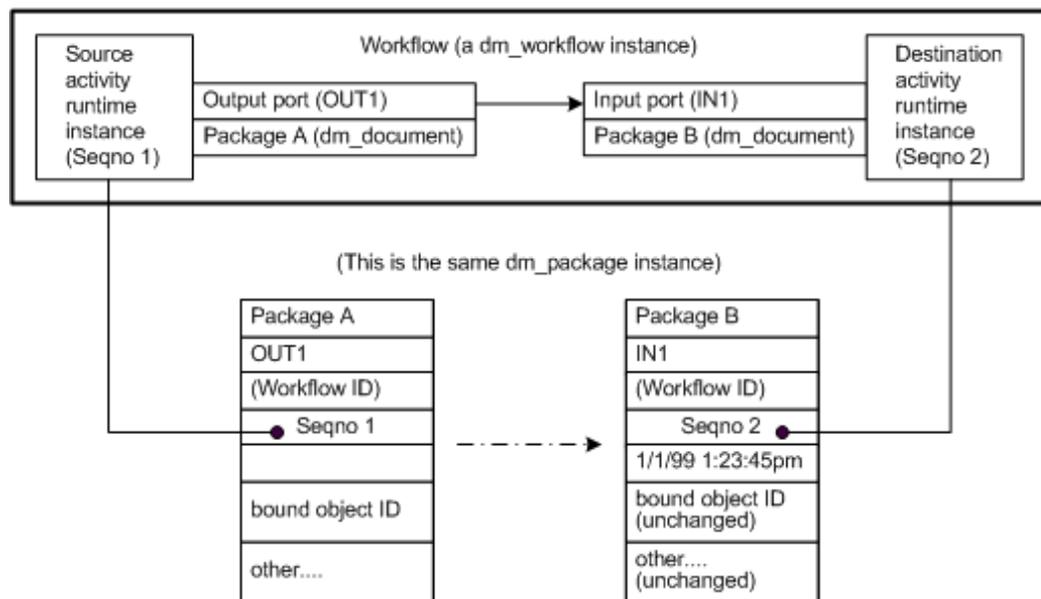
## How activities accept packages

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` properties of those packages.

Figure 19, page 288, illustrates this process.

**Figure 19. 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

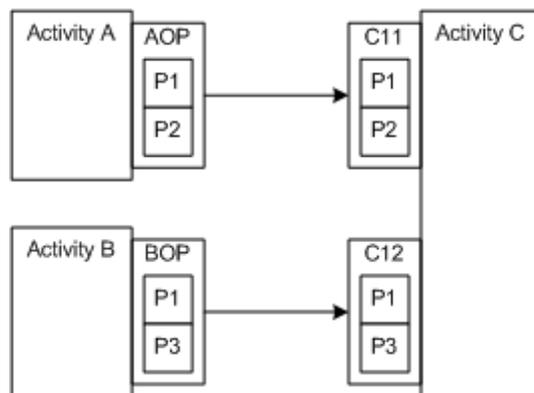
In addition, at the destination activity, the server performs some bookkeeping tasks, including:

- 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 20, page 289](#), Activity C has two input ports: C11, which accepts packages P1 and P2, and C12, 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, C11 accepts the packages. In that case, the packages arriving from Activity B are ignored.

**Figure 20. 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.

## 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 and later client libraries and BPM 5.3, and Process Builder. 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` property 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 more information

- [Warning and suspend timers, page 260](#), describes each kind of timer.

## 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` properties. If the `dm_completedworkitem` event is audited, the values are also recorded in the `string_3` property 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 `dm_WFReporting` job

The data is generated by the `dm_WFReporting` job, which invokes the `dm_WFReporting` method. The method examines audit trail entries for all workflow events for completed workflows. It collects the information from these events and generates objects of type `dmc_completed_workflow` and `dmc_completed_workitem`. Each object represents the data for one completed workflow or one completed work item in a completed workflow. The Webtop Workflow Reporting tool uses the information in the objects generated by the job to create its reports.

## For more information

- The *Content Server Administration Guide* has information about:
  - Activating a job
  - Starting auditing
- 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 DFC to halt the workflow, use a halt or haltEx method.

You can resume, restart, or abort a halted workflow.

## 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` property of the workflow. This is a repeating property, 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` property.

## Resuming a halted workflow or activity

Use an `IDfWorkflow.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.

## Aborting a workflow

Aborting a workflow terminates the workflow and sets the `r_runtime_state` property 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 properties 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 properties (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 properties 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 properties 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 properties (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 301](#)
- [Default lifecycles for object types, page 305](#)
- [Lifecycle definitions, page 305](#)
- [How lifecycles work, page 307](#)
- [Object permissions and lifecycles, page 314](#)
- [Integrating lifecycles and applications, page 314](#)
- [Designing a Lifecycle, page 316](#)
- [Lifecycle state definitions, page 317](#)
- [Lifecycle creation, page 330](#)
- [Custom validation programs, page 334](#)
- [Modifying lifecycles, page 335](#)
- [Obtaining information about lifecycles, page 336](#)
- [Deleting a lifecycle, page 337](#)

## Introducing lifecycles

This section provides an introduction to lifecycles, the EMC Documentum feature that automates management of a document's life cycle.

## What a lifecycle is

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 states representing the draft, review, rewrite, approved, and obsolete stages of an SOP's life. 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

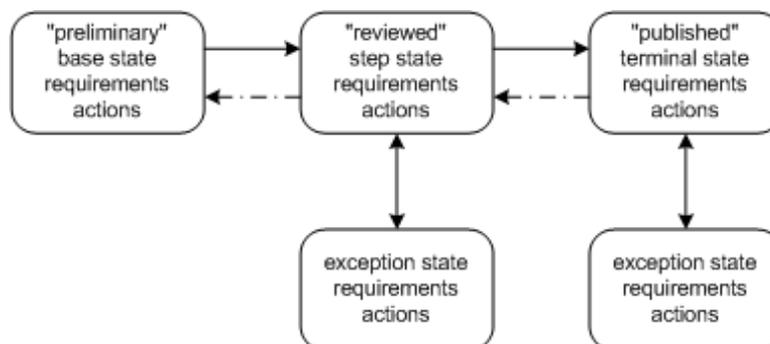
There are two kinds of states: normal and exception. Normal states are the states that define the typical stages of an object's life. Exception states represent situations outside of the normal stages of an object's life. All lifecycles must have normal states. Exception states are optional. Each normal 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 21, page 302, shows an example of a lifecycle with exception states. Like normal states, exception states have their own requirements and actions.

**Figure 21. 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 properties: `included_type` and `include_subtypes`. These are repeating properties. The `included_type` property records, by name, the object types that can be attached to a lifecycle. The `include_subtypes` property is a Boolean property that records whether subtypes of the object types specified in `included_type` may be attached to the lifecycle. The value at a given index position in `include_subtypes` is applied to the object type identified at the corresponding position in `included_type`.

An object can be attached to a lifecycle if either

- The lifecycle's `included_type` property 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` property is set to `TRUE`

For example, suppose a lifecycle definition has the following values in those properties:

```
included_type[0]=dm_sysobject
included_type[1]=dm_document

include_subtypes[0]=F
include_subtypes[1]=T
```

For this lifecycle, users can attach any object that is the `dm_sysobject` type. However, the only `SysObject` subtype that can be attached to the lifecycle is a `dm_document` or any of the `dm_document` subtypes.

The object type defined in the first index position (`included_type[0]`) is called the primary object type for the lifecycle. Object types identified in the other index positions in `included_type` must be subtypes of the primary object type.

## 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.

## For more information

- [Entry criteria definitions, page 320](#), contains information about defining entry criteria.
- [Actions on entry definitions, page 323](#), contains information about defining actions on entry.
- [Post-entry action definitions, page 326](#), contains information about defining post-entry actions.

---

## 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

This section briefly describes how lifecycle definitions are handled.

## Repository storage

The definition of a lifecycle is stored in the repository as a `dm_policy` object. The properties of the object define the states in the lifecycle, the object types to which the lifecycle may be attached, whether state extensions are used, and whether a custom validation program is used.

The state definitions within a lifecycle definition consist of a set of repeating properties. The values at a particular index position across those properties represent the definition of one state. The sequence of states within the lifecycle is determined by their position in the properties. The first state in the lifecycle is the state defined in index position [0] in the properties. The second state is the state defined in position [1], the third state is the state defined in index position [2], and so 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.

## Lifecycle definition states

Lifecycles 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` property of the policy object.

## Lifecycle definition 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` property are subtypes of the primary attachable type.
- All objects referenced by object ID in the policy definition exist.
- For Java-based lifecycles, that all 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.

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.

## For more information

- [Lifecycle state definitions, page 317](#), contains a detailed list of the information that makes up a state definition.
- [Custom validation programs, page 334](#), contains more information on writing a custom validation program.

# 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` property
- Sets the `r_alias_set_id` to the object ID of the alias set associated with the lifecycle, if any
- Executes any actions defined for the state
- Sets the `r_current_state` property to the number of the state

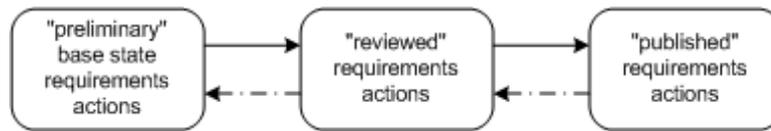
From this point, the object continues through the lifecycle. If the object was attached to a normal state, it can move to the next normal state, to the previous normal state, or to the exception state defined for the normal state. If the object was attached to an exception state, it can move to the normal state associated with the exception state or to the base state.

Each time the object is moved forward to a normal state or to an exception state, 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` property 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 properties) 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 properties. It doesn't evaluate the entry criteria.

Objects cannot skip normal steps as they progress through a lifecycle.

[Figure 22, page 308](#), 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 22. 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` property is set for the state.

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.

Programmatically, attaching an object is accomplished using an `IDfSysObject.attachPolicy` 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` property in the `docbase` config object. In those instances, that user must be either the owner or a superuser with Write permission or have Write and Change State permission on the object.

A promotion only succeeds if the object satisfies any entry criteria and actions on entry defined for the target state.

It is possible to bypass the entry criteria. If you choose to do that, the server does not enforce the entry criteria, but simply performs the actions associated with the destination state and, on their completion, moves the object to the destination state. You must own the lifecycle's policy object or be a superuser to bypass entry criteria.

Promotions are accomplished programmatically using one of the promote methods in the IDfSysObject interface. Bypassing the entry criteria is accomplished by setting the override argument in the method to true.

## Batch 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.

## 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 property for the state must be TRUE. Additionally, to demote an object back to the base state, the return\_to\_base property value must be TRUE for the current state.

Users who own an object or are superusers need only Write permission to demote the object. Other users must have Write permission and Change State permission to demote an object. If the user has only Change State permission, Content Server will attempt to demote the object as the user defined in the a\_bpaction\_run\_as property in the docbase config object. In those instances, that user must be either the owner or a superuser with Write permission or have Write and Change State permission on the object.

If the object'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 programmatically using one of the demote methods in the IDfSysObject interface.

## 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` property in the `docbase config` object. In those instances, that user must be either the owner or a superuser with Write permission or have Write and Change State permission on the object.

When an object is moved to an exception state, the server checks the state'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 programmatically using one of the suspend methods in the `IDfSysObject` interface. Bypassing the entry criteria is accomplished by setting the `override` argument in the method set to `true`.

## 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` property in the `docbase config` object. In those instances, that user must be either the owner or a superuser with Write permission or have Write and Change State permission on the object.

Additionally, to resume an object back to the base state, the exception state must have the `return_to_base` property set to `TRUE`.

When an object is resumed to either the normal state or the base state, the object must satisfy the target state'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.

Programmatically, resuming an object is accomplished using one of the resume methods in the `IDfSysObject` interface. Bypassing the entry criteria is accomplished by setting the `override` argument in the method set to `true`.

## 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 a variation that allows you to schedule a transition for a particular date and time. For example, for a scheduled promotion, you would use the `IDfSysobject.schedulePromote` method. If you issue a method that schedules the movement between states, 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 properties are set to the specified date and time. The job runs as the user who issued the initial method that created the job, unless the `a_bpaction_run_as` property is set in the docbase config. If that is set, the job runs as the user defined in that property.

The destination state for a scheduled change can be an exception state or any normal state except the base state. You cannot schedule the same object for multiple state transitions at the same time.

You can unschedule a scheduled transition. Each of methods governing movement also has a variation that allows you to cancel a schedule change. For example, to cancel a scheduled promotion, you would use `cancelSchedulePromote`.

## 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 18, page 312](#), lists the methods.

**Table 18. 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.
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 properties are set to the date and time identified in the scheduling argument of the state-change method. When the job is executed, it invokes dm\_bp\_transition\_java or dm\_bp\_transition.

**Note:** The dm\_bp\_transition\_java and dm\_bp\_transition methods are also invoked by an attach method.

The dm\_bp\_transition\_java and dm\_bp\_transition methods perform the following actions:

1. Use the supplied login ticket to connect to 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 return.
8. Set the `r_current_state` and `r_resume_state` properties. For an `attachPolicy` method, also sets the `r_policy_id` and `r_alias_set_id` properties.
9. Save the `SysObject`.
10. If no errors occurred, commit the transaction.
11. If errors occurred, abort the transaction.
12. Execute any post-entry actions.

By default, the transition methods run as the user who issued the state-change method. To change the default, you must set the `a_bpaction_run_as` property in the `docbase` config object. If the `a_bpaction_run_as` property is set in the `docbase` config object, the actions associated with state changes are run as the user indicated in the property. Setting `a_bpaction_run_as` 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 property is not set, the actions are run as the user who changed the state.

If an error occurs during execution of the `dm_bp_transition_java` or `dm_bp_transition` method, a log file is created. It is named `bp_transition_session_.out` in `%DOCUMENTUM%\dba\log\repository_id\bp` (`$DOCUMENTUM/dba/log/repository_id/bp`). If an error occurs during execution of the `dm_bp_schedule_java` or `dm_bp_schedule` methods, a log file named `bp_schedule_session_.out` is created in the same directory.

**Note:** If you set the `timeout_default` value for the `bp_transition` method to a value greater than five minutes, 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.

## For more information

- [Attachability, page 318](#), describes attachability in more detail.
- The *Content Server DQL Reference Manual* provides reference information for the `BATCH_PROMOTE` administration method.

- [Scheduled transitions, page 311](#), contains more information about the jobs that process transitions.

## 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 to the SOP folder requires the Change Location permission and the appropriate base object-level permission to unlink the document from its current folder and link it to the SOP folder.

The actions associated with a state can be used to reset permissions as needed.

## 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 properties. (Sysobjects can use aliases in the `owner_name`, `acl_name`, and `acl_domain` properties.) Which alias set is chosen is determined by how the client application is designed. The application may display a list of the alias sets to the user and allow the user to pick one. Or, the application may use the default resolution algorithm for choosing the alias set.

Additionally, you can use template ACLs, which contain aliases, and aliases in folder paths in actions defined for states to make the actions usable in a variety of contexts.

If you define one or more alias sets for a lifecycle definition, those choices are recorded in the policy object's `alias_set_ids` property.

## 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 properties that store the information required by the application for the states. For example, suppose you have an application called `EngrApp` that will handle documents attached to `LifecycleA`. This lifecycle has two states, `Review` and `Approval`, that require a list of users and a deadline date. The state extension subtype for this lifecycle will have two defined properties: `user_list` and `deadline_date`. Or perhaps the application needs a list of users for one state and a list of possible formats for another. In that case, the properties defined for the state extension subtypes will be `user_list` and `format_list`.

State extension objects are associated with particular states through the `state_no` property, inherited from the `dm_state_extension` supertype.

State extensions must be created manually. The Lifecycle Editor does not support creating state extensions.

## 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` property to determine the type of the object's current state and then use the type name to determine the application behavior.

In addition to the repeating property that defines the state types in the policy object, state types may also be recorded in the repository using `dm_state_type` objects. State type objects have two properties: `state_type_name` and `application_code`. The `state_type_name` identifies the state type and `application_code` identifies the application that recognizes and uses that state type. You can create these objects for use by custom applications. For example, installing 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.

## For more information

- [Determining the lifecycle scope for SysObjects, page 351](#), describes the default resolution algorithm for choosing the alias set to be used with a lifecycle.
- [Using aliases in actions, page 329](#), contains more information about using aliases in templates and lifecycle actions.
- [Using state extensions, page 331](#), contains information about creating a state extension subtype and adding a state extension to a lifecycle state.

## Designing a Lifecycle

This section describes the basic decisions required before starting to create a lifecycle.

### Required design decisions

When you design a lifecycle, you must make the following decisions:

- What objects will use the lifecycle

- What normal and exception states the lifecycle will contain, and for each state, what is the definition of that state

A state's definition includes a number of items, such as whether it is attachable, what its entry criteria, actions on entry, and post-entry actions are, and whether it allows scheduled transitions.

- Whether to include an alias set in the definition
- Whether you want to assign state types

If objects attached to the lifecycle will be handled by the 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.

- What states, if any, will have state extensions
- Whether you want to use a custom validation program

## For more information

- [Types of objects that may be attached to lifecycles, page 303](#), describes how the object types whose instances may be attached to a lifecycle are specified.
- [Lifecycle state definitions, page 317](#), contains guidelines for defining lifecycle states.
- [Lifecycles, alias sets, and aliases, page 314](#), describes how alias sets are used with lifecycles.
- [State types, page 315](#), describes the purpose and use of state types.
- [State extensions, page 315](#), describes the purpose and use of state extensions.
- [Lifecycle definition validation, page 306](#), describes validation and the default validation methods.
- [Attachability, page 318](#), describes the attachability characteristic of states.

## Lifecycle state definitions

Each state in the lifecycle has a state definition. All of the information about states is stored in properties in the `dm_policy` object that stores the lifecycle definition. For example, whether a state is a normal or exception state is recorded in the `state_class` property. The properties that record a state definition are repeating properties, and the values at a particular index position across the properties represent the definition of one state in the lifecycle. If you are using the Lifecycle Editor to create a lifecycle, these properties are set automatically when you create the definition. If you are creating a lifecycle outside the Editor, you must set the properties yourself.

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` property of the `dm_policy` object holds the names of the states.
- For normal states, whether users can attach an object to the state
- Whether objects can be returned to the base state from the state after Save, Checkin, Saveasnew, or Branch operations
- Whether objects can be demoted from the state
- Whether users can schedule transitions.
- The state's type, if any

## 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` property. This is a Boolean property.

## 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 properties in the policy object: `return_to_base` and `return_condition`. These are repeating properties. Each index position corresponds to a lifecycle state. `return_to_base` is a Boolean property that controls whether an object in a particular state can be returned to the base state. `return_condition` is an integer property 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` property 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` property. This property is set to `TRUE` if you decide that transitions out of the state may be scheduled. It is set to `FALSE` if you do not allow scheduled transitions for the state.

The setting of this property only affects whether objects can be moved out of a particular state at scheduled times. It has no effect on whether objects can be moved into a state at a scheduled time. For example, suppose `StateA` allows scheduled transitions and `StateB` does not. Those settings mean that you can promote an object from `StateA` to `StateB` on a scheduled date. But, you cannot demote an object from `StateB` to `StateA` on a scheduled date.

## Entry criteria 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 SBO modules and a jar file. For information about SBO 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

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 properties referenced in the program must be properties 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 properties 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 328.

## 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 properties referenced in the function must be properties 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 properties 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 property, `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` property is applied to objects entering the state identified in the corresponding index position in the `state_name` property.

Because procedure objects can be versioned, policy objects have the repeating property `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` properties.

## 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. Properties referenced in the expressions must be properties defined for the lifecycle's primary object type (the object type identified in `included_type[0]`). The expressions cannot reference properties inherited by that object type. For example, the expression `title=MyBook` references the `title` property, 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 property. To define criteria for a particular state, set the computed property at the index position corresponding to the state's index position in the `state_name` property. 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` property. The `entry_criteria_id` property is a repeating property, 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` property 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.

If you define your own actions on entry, the program must be a Java program if the lifecycle is Java-based. Java-based actions on entry are stored in the repository as SBO modules and a jar file. If the lifecycle is Docbasic-based, the actions on entry program must be a Docbasic program.

If both system-defined and user-defined actions on entry are specified for a state, the server performs the system-defined actions first and then the user-defined actions. An object can only enter the state when all actions on entry complete successfully.

## 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` property. If you are using a Docbasic-based lifecycle, the chosen actions are recorded in the `action_object_id` property.

**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 properties 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.

## 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` property. These properties 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 properties referenced in the function must be properties 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 properties 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` property is a repeating property. 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 property `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 SBO modules and a jar file.

If the lifecycle is Docbasic-based, the post-entry action programs must be Docbasic programs.

## 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.

## 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,
    ByVal 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 properties referenced in the function must be properties defined for the primary object type of the lifecycle. The function cannot reference properties 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 property.

The user\_postproc\_id property is a repeating property. 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 property.

Because procedure objects can be versioned, policy objects have the repeating property 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.

## Using aliases in actions

Aliases provide a way to make the actions you define for a state flexible and usable in multiple contexts. Many documents may have the same life stages, but have differing business requirements. For example, most documents go through a writing draft stage, a review stage, and a published or approved stage. However, some of those documents may be marketing documents, some may be engineering documents, and some may be human resource documents. Each kind of document requires different users to write, review, and approve them.

Using aliases in actions can make it possible to design one lifecycle that can be attached to all these kinds of documents. You can substitute an alias for a user or group name in

an ACL and in certain properties of a SysObject. You can use an alias in place of path name in the Link and Unlink methods.

In template ACLs, aliases can take the place of the accessor name in one or more access control entries. When the ACL is applied to an object, the server copies the template, resolves the aliases in the copy to real names, and assigns the copy to the object.

In the Link and Unlink methods, aliases can replace the folder path argument. When the method is executed, the alias is resolved to a folder path and the object is linked to or unlinked from the proper folder.

When the actions you define for a state assign a new ACL to an object or use the Link or Unlink methods, using template ACLs and aliases in the folder path arguments ensures that the ACL for an object or its linked locations are always appropriate.

## For more information

- [Including electronic signature requirements, page 329](#), contains information about adding a signature requirement to a program.
- [Scheduled transitions, page 311](#), describes how scheduled transitions are implemented internally.
- The Documentum Composer documentation contains a listing of the pre-defined actions.
- The *DFC Development Guide* describes simple modules.
- *Content Server Administration Guide* contains complete information about using digital and electronic signatures or simple sign-offs.
- [Appendix A, Aliases](#), describes how aliases are implemented in detail.

## Lifecycle creation

Lifecycles are typically created using the Lifecycle Editor, which is accessed through Documentum Composer. It is possible to create a lifecycle definition by directly issuing the appropriate methods or DQL statements to create, validate, and install the definition. However, using the Lifecycle Editor is the recommended and easiest way to create a lifecycle.

## 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.
2. Create a draft lifecycle definition.
3. If using state extensions, create the state extension subtype and object instances.
4. If using a custom validation program, create and install the program.
5. Validate the draft lifecycle definition.
6. Install the validated lifecycle definition.

## Using state extensions

If you want to use state extensions for a lifecycle, you must add them 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.

The basic procedure is:

1. Create a subtype of the dm\_state\_extension type for the lifecycle.  
Use the Documentum Composer to define a subtype of dm\_state\_extension.
2. Create the extension objects.  
The extension objects are instances of the subtype you created for the lifecycle. You must create objects of that subtype for each lifecycle state that requires an extension.
3. Set the extension\_type property in the lifecycle definition to the name of the state extension subtype created for the lifecycle.  
Use the API or DQL to set the extension\_type property. You cannot use the Lifecycle Editor.

## Extension objects

The properties of an extension object identify the lifecycle and state extended by the extension object:

- `state_no`  
This identifies the state by the state's number in the lifecycle.
- `parent_id`  
This identifies the lifecycle by the object ID of the `dm_policy` object representing the lifecycle's definition.
- `child_id`  
This identifies the extension object by its object ID.
- `relation_name`  
This is set to `BP_STATE_EXTENSION` to define the relationship between the extension object and the lifecycle state.

State extensions, like other types of relationships, may be maintained across versions of the policy if you wish. The `permanent_link` property setting controls that behavior. If the property is `TRUE`, the relationship is retained when the lifecycle definition is versioned.

## 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. These methods have an argument that, if set to `TRUE`, directs the method to perform the operation 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` property of the method object associated with the `dm_bp_transition_java` method:

```
-debug T
```

This argument can also be added to the `method_verb` property 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`).

## For more information

- [Designing a Lifecycle, page 316](#), has guidelines for designing a lifecycle.
- The *Documentum Composer* documentation describes how to use the Lifecycle Editor to create a lifecycle.
- [Using state extensions, page 331](#), has information about state extensions.

- [Custom validation programs, page 334](#), contains information about custom validation.
- [Lifecycle definition validation, page 306](#) has information about lifecycle validation.

## Custom validation programs

This section outlines the basic procedure for creating and installing a user-defined lifecycle validation method. Documentum provides standard technical support only 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.

### Overview

In some situations, you may want some custom validation for a particular lifecycle. You can write and use a custom validation program for a lifecycle. If you want to use a custom validation program, the program must be written in the same language as that used for any entry criteria, actions on entry, or post-entry actions written for the lifecycle. This means that if those programs are written in Java, the custom validation program must be in Java also. If the programs are written in Docbasic, the validation program must be in Docbasic also.

**Note:** Docbasic is a deprecated language.

After you write the program, use Documentum Composer to add the custom validation program to the lifecycle definition. You must own the lifecycle definition (the policy object) or have at least Version permission on it to add a custom validation program to the lifecycle.

### For more information

- The *Documentum Composer* documentation contains instructions for creating a custom validation program and adding to a lifecycle definition.

## 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.

## Obtaining 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 properties 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 property.

## Object information

Computed properties contain information about an object's current state in a lifecycle, and about the object's policy-related permissions.

Three computed properties 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 properties describe an object's current state in a lifecycle:

- The `r_policy_id` property contains the object ID of the policy object representing the lifecycle attached to the object
- The `r_current_state` property contains the name of the object's current state in the lifecycle.
- The `r_resume_state` property contains the name of the previous normal state if the current state is an exception state.

In addition, there are computed properties that record the user's basic and extended permissions. These properties may be useful when checking permissions for a lifecycle operation.

## For more information

- The *Documentum Object Reference Manual* lists the computed properties that record a user's basic and extended permissions.

## 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` properties.

The objects identified in the `user_criteria_id` and `user_action_id` properties 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 339](#)
- [Introducing Inboxes, page 341](#)
- [dmi\\_queue\\_item objects, page 341](#)
- [Obtaining Inbox content, page 342](#)
- [Manual queuing and dequeuing , page 343](#)
- [Registering and unregistering for event notifications, page 344](#)
- [Querying for registration information, page 346](#)

### Introducing tasks and events

Tasks and events are occurrences within an application or repository that are of interest to users. This section describes how these occurrences are supported by Content Server.

### What a task is

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.

## What an event is

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.

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.

## Repository implementation

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.

## Accessing tasks and events

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.

Tasks are sent to the inbox automatically, when the task is generated. Users must register to receive events. Users can register to receive notifications of system-defined events. When a system-defined event occurs, Content Server sends an event notification automatically to any user who is registered to receive the event.

Users cannot register for application-defined events. Generating application-defined events and triggering notifications of the events are managed completely by the application.

## For more information

- The *Content Server Administration Guide* has tables listing all system-defined events.
- [dmi\\_queue\\_item objects, page 341](#), describes queue items.

- [Work item objects, page 267](#), describes work items.
- [Introducing Inboxes, page 341](#), describes inboxes.

## 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.

### What an inbox is

An inbox is a virtual container that holds tasks, event notifications, and other items sent to users manually (using a Queue method). For example, one of your employees might place a vacation request in your inbox, or a co-worker might ask you to review a presentation. Each user in a repository has an inbox.

### 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 querying and referencing `dmi_queue_item` objects.

## dmi\_queue\_item objects

The `dmi_queue_item` object type supports inboxes.

## Purpose

All items that appear in an inbox are managed by the server as objects of type `dmi_queue_item`. The properties of a queue item object contain information about the queued item. For example, the `sent_by` property contains the name of the user who sent the item and the `date_sent` property tells when it was sent.

## Historical record

`dmi_queue_item` objects are persistent. They remain in the repository even after the items they represent have been removed from an inbox, providing a persistent record of completed tasks. Two properties that are set when an item is removed from an inbox are particularly helpful when examining the history of a project with which tasks are associated. These properties 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.

## For more information

- The *Documentum Object Reference Manual* contains the reference information for the `dmi_queue_item` object type.

## Obtaining Inbox content

There are a variety of ways to obtain the content of a particular inbox programmatically. You can use a `GET_INBOX` administration method or a `IDfSession.getEvents` method or you can query the `dm_queue` view.

To determine whether to refresh an inbox, you can use an `IDfSession.hasEvents` method to check for new items. A new item is defined as any item queued to the inbox after the previous execution of `getEvents` for the user. The method returns `TRUE` if there are new items in the inbox or `FALSE` if there are no new items.

## GET\_INBOX administration method

GET\_INBOX returns a collection containing the inbox items in query result objects. Using GET\_INBOX is the simplest way to retrieve all items in a user's inbox.

## getEvents method

An IDfSession.getEvents method returns all new (unread) items in the current user's queue. Unread items are all queue item objects placed on the queue after the last getEvents execution against that queue.

The queue item objects are returned as a collection. Use the collection identifier to process the returned items.

## The dm\_queue view

The dm\_queue view is a view on the dmi\_queue\_item object type. To obtain information about a queue using DQL, query against this view. Querying against this view is the simplest way to view all the contents of a queue. For example, the following DQL statement retrieves all the items in Haskell's inbox. For each item, the statement retrieves the name of the queued item, when it was sent, and its priority:

```
SELECT "item_name","date_sent","priority" FROM "dm_queue"  
WHERE "name" = 'Haskell'
```

## For more information

- The *Content Server DQL Reference Manual* has instructions on using GET\_INBOX.
- The *Documentum Object Reference Manual* contains the reference information about the properties of a queue item object.

## Manual queuing and dequeuing

Most inbox items are generated automatically by workflows or an event registration. However, you can manually or programmatically queue a SysObject or a workflow-related event notification using a queue method. You can also manually or programmatically take an item out of an inbox. This is called dequeuing the item.

## Queuing items

Use a queue method to place an item in an inbox. Executing a queue method creates a queue item object. You can queue a SysObject or a user- or application-defined event.

When you queue an object, including an event name is optional. You may want to include one, however, to be manipulated by the application. 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 property for one of the workflow's activities.

Although you must assign a priority value to queued items and events, your application can ignore the value or use it. For example, perhaps the application reads the priorities and presents the items to the user in priority order. The priority is ignored by Content Server.

You may also include a message to the user receiving the item.

## Dequeuing an inbox item

Use an IDfSession.dequeue method to remove an item placed in an inbox using a queue method. Executing a dequeue method sets two queue item properties:

- dequeued\_by

This property contains the name of the user who dequeued the item.

- dequeued\_date

This property contains the date and time that the item was dequeued.

## 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 for a system event, you must register for the event.

The event can be a specific action on a particular object or a specific action on objects of a particular type. You can also register to receive notification for all actions on a particular object.

For instance, you 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 an `IDfSysObject.registerEvent` method.

Although you must assign a priority value to an event when you use the `registerEvent` method, your application can ignore the value or use it. This argument is provided as an easy way for your application to manipulate the event when the event appears in your inbox. For example, the application might sort out events that have a higher priority and present them first. The priority is ignored by Content Server.

You cannot register another user for an event. Executing a `registerEvent` method registers the current user for the specified event.

## Removing a registration

To remove an event registration, use Documentum Administrator or an `IDfSysObject.unregister` method.

Only a user with Sysadmin or Superuser privileges can remove another user'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.

## For more information

- The *Content Server Administration Guide* lists the system events for which you may register for notification.

## 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 347](#)
- [Internal implementation, page 348](#)
- [Defining aliases, page 348](#)
- [Alias scopes, page 349](#)
- [Resolving aliases in SysObjects, page 351](#)
- [Resolving aliases in template ACLs, page 352](#)
- [Resolving aliases in Link and Unlink methods, page 352](#)
- [Resolving aliases in workflows, page 353](#)

## 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` properties
- In ACL template objects, in the `r_accessor_name` property

**Note:** Aliases are not allowed as the `r_accessor_name` for ACL entries of type `RequiredGroup` or `RequiredGroupSet`.
- In workflow activity definitions (`dm_activity` objects), in the `performer_name` property
- In a Link or Unlink method, in the folder path argument

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 properties 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 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 property of the alias set object.

To put an alias in a SysObject or activity definition, use a set method. To put an alias in a template ACL, use a grant method. To include an alias in a link or unlink method, substitute the alias specification for the folder path argument.

For example, suppose you have an alias set named engr\_aliases that contains an alias\_name called engr\_vp, which is mapped to the user name henryp. If you set the

owner\_name property to %engr\_alias.engr\_vp, when the document is saved to the repository, the server finds the alias set object named engr\_aliases and resolves the alias to the user name henryp.

It is also valid to specify an alias name without including the alias set name. In such cases, the server uses a 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 property. This property is set when the workflow is instantiated. The server copies the alias set specified in the perf\_alias\_set\_id property of the workflow's definition (process object) and sets the r\_alias\_set\_id property in the workflow object to the object ID of the copy.

Within the session scope, the server searches the alias set object defined in the session config's alias\_set property.

In the user performer scope, the server searches the alias set defined for the user who performed the work item that started the activity containing the alias. A user's alias set is defined in the `alias_set_id` property 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` property.

Within the server configuration scope, the search is conducted in the alias set defined in the `alias_set_id` property of the server config object.

## 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` property. This property is set when the object is attached to a lifecycle.

Within the session scope, the server searches the alias set object defined in the session config's `alias_set` property.

Within the user's scope, the search is in the alias set object defined in the `alias_set_id` property 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` property 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` property, 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` property.
- If the session scope's alias set isn't found, the server uses the alias set defined for the user's scope if it is in the `alias_set_ids` list.
- If the user scope'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` property.

If the policy object has no defined alias set objects in the `alias_set_ids` property, the SysObject's `r_alias_set_id` property 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` property. This is the lifecycle scope.
- If the alias is not found in the lifecycle scope or if `r_alias_set_id` is undefined, the server looks next at the alias set object defined for the session scope.
- If the alias is not found in the session'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` property 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` property 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` property of the session config object. If a session scope alias set is defined, but no match is found, the server returns an error.
- If the object has no attached lifecycle and there is no alias defined for the session scope, the server resolves the alias using the alias set defined for the user scope. This is the alias set identified in the `alias_set_id` property of the `dm_user` object for the current user. If a user scope alias set is defined but no match is found, the server returns an error.
- If the object has no attached lifecycle and there is no alias defined for the session or user scope, the server resolves the alias using the alias set defined for the user'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.

---

## 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.

Resolving an alias when an activity starts is done automatically by the server.

## 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` property to the copy's object ID.

If the workflow is started through a Documentum client application, the application prompts the starter for alias values for the missing alias names. The server adds the alias values to the alias set copy attached to the workflow object. If the workflow is started through a custom application, the application must prompt the workflow'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` property is set to 0. The server searches the following scopes, in the order listed:

- Workflow
- Session
- User performer of the previous work item
- The default group of the previous work item'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` property 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` property.

If the `resolve_pkg_name` property 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` property is not set, the search begins with the package defined in `r_package_name[0]`. The components of that package are searched. If a match is not found, the search continues with the components in the package identified in `r_package_name[1]`. The search continues through the listed packages until a match is found.

## The user resolution algorithm

The server uses the user resolution algorithm if the activity's `resolve_type` property is set to 2. In such cases, the search is conducted in the following scopes:

- The alias set defined for the user performer of the previous work item

- The alias set defined for the default group of the user performer of the previous work item.

The server first searches the alias set defined for the user. If a match isn't found, the server searches the alias set defined for the user'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` property. For example, if `performer_type` indicates that the designated performer is a user, the server will validate that the alias value represents a user, not a group. If the alias value matches the specified `performer_type`, the work item is created for the activity.

## 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



## Internationalization Summary

This appendix describes EMC 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 summarizes the internationalization requirements of various features of Content Server.

**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 4, The Data Model](#).

The chapter contains the following topics:

- [What Content Server internationalization is, page 357](#)
- [Content files and metadata , page 358](#)
- [Configuration requirements for internationalization, page 359](#)
- [Where ASCII must be used , page 362](#)
- [User names, email addresses, and group names, page 362](#)
- [Lifecycles, page 363](#)
- [Docbasic, page 363](#)
- [Federations, page 363](#)
- [Object replication, page 364](#)
- [Other cross-repository operations, page 364](#)

## What Content Server internationalization is

Internationalization refers to Content Server's ability to handle communications and data transfer between itself and client applications in a variety of code pages. This ability 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.)

## Content files and metadata

This section summarizes how internationalization affects content file and metadata storage in a repository.

### Content files

You can store content files created in any code page and in any language in a repository. The files are transferred to and from a repository as binary files.

**Note:** It is recommended that all XML content use one code page.

### Metadata

What metadata values you can store depends on the code page of the underlying database. The code page may be a national character set or it may be Unicode.

If the database was configured using a national character set as the code page, you can store only characters allowed by that code page. For example, if the database uses EUC-KR, you can store only characters that are in the EUC-KR code page as metadata.

All code pages supported by 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.

### Client communications with Content Server

All communications between DFC and Content Server are performed using the UTF-8 (Unicode) code page.

## Constraints

A UTF-8 Unicode repository can store metadata from any language. However, if your client applications are using incompatible code pages in national character sets, they may not be able to handle metadata values set in different code page. For example, if an application using Shift-JIS or EUC-JP (the Japanese code pages) stores objects in the repository and another application using ISO-8859-1 (Latin-1 code page) retrieves that metadata, the values returned to the ISO-8859-1 application will be corrupted because there are characters in the Japanese code page that are not found in the Latin-1 code page.

## For more information

- *XML Application Development Guide* has more information and recommendations for creating XML content.

## 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.

## 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.

- 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.

## For more information

- The *Content Server Administration Guide* contains a table of default values for code pages by locale.

## Values set during sessions

Properties defining code pages are set when DFC is initialized and when a session is started.

## The client config object

A client config object records global information for client sessions. It is created when DFC is initialized. The values reflect the information found in the `dfc.properties` file used by the DFC instance. Some of the values are then used in the session config object when a client opens a repository session.

The following properties for internationalization are present in a client config object:

- `dfc.codepage`

The `dfc.codepage` property controls conversion of characters between the native code page and UTF-8. The value is taken from the `dfc.codepage` key in the `dfc.properties` file on the client host. This code page is the preferred code page for repository sessions started using the DFC instance. The value of `dfc.codepage` overrides the value of the `default_client_codepage` property in the server config object.

The default value for this key is UTF-8.
- `dfc.locale`

This is the client's preferred locale for repository sessions started by the DFC instance.

## 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 property values are taken from values in the client config object, the server config object, and the connection config object.

The following properties for internationalization are set in the session config object:

- `session_codepage`

This property is obtained from the client config object's `dfc.codepage` property. It is the code page used by a client application connecting to the server from the client host.

If needed, set the `session_codepage` property in the session config object early in the session and do not reset it.
- `session_locale`

This is the locale of the repository session. The value is obtained from the `dfc.locale` property of the client config object. If `dfc.locale` is not, the default value is determined programmatically from the locale of the client's host machine.

## How values are set

The values of the `dfc.codepage` and `dfc.locale` properties in the client config object determine the values of `session_codepage` and `session_locale` in the session config object. These values are determined in the following manner:

1. Use the values supplied programmatically by an explicit set on the client config object or session config object.

2. If the values are not explicitly set, examine the settings of `dfc.codepage` and `dfc.locale` keys in the `dfc.properties` file.

If not explicitly set, the `dfc.codepage` key and `dfc.locale` keys are assigned default values. DFC derives the default values from the JVM, which gets the defaults from the operating system.

## Where ASCII must be used

Some objects, names, directories, and property 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` property of a location object
- Mount point object names
- Format names
- Format DOS extensions
- All file store names
- Object type names and property names
- Federation names
- Content stored in turbo storage
- String literals included in check constraint definitions
- String literals included in the expression string referenced in the conditional clauses of value assistance definitions
- Text specified in an AS clause in a SELECT statement

## User names, email addresses, and group names

There are code page-based requirements for the following property values:

- `dm_user.user_name`

- dm\_user.user\_os\_name
- dm\_user.user\_db\_name
- dm\_user.user\_address
- dm\_group.group\_name

The requirements for these differ depending on the site's configuration. If the repository is a standalone repository, the values in the properties must be compatible with the code page defined in the server's server\_os\_codepage property. (A standalone repository does not participate in object replication or a federation and its users never access objects from remote repositories.)

If the repository is in an installation with multiple repositories but all repositories have the same code page defined in server\_os\_codepage, the values in the user property must be compatible with the server\_os\_codepage. However, if the repositories have different code pages identified in server\_os\_codepage, then the values in the properties 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 property 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 UTF-8 if the source and target repositories are version 4.2 or later. 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.

% (percent sign) in alias specification, 348

## A

a\_bpaction\_run\_as property, 313

a\_contain\_desc property, 203

a\_contain\_type property, 203

a\_content\_type attribute, 174

a\_controlling\_app property, 127

a\_full\_text attribute, 172

a\_is\_signed property, 146

a\_special\_app property

    use in workflows, 238

a\_storage\_type attribute, 176

a\_wq\_name property

    automatic activities, use by, 283

AAC tokens, *see* application access control

    tokens

Abort method, 294

ACLs

    custom, creating, 186

    default ACL, assigning, 185

    default, described, 184

    described, 134, 184

    entries, described, 135

    grantPermit method, 187

    kinds of, 135

    non-default, assigning, 186

    object-level permissions, 130

    revokePermit method, 187

    room defaults, 187

    template ACLs, 185

    templates

        alias use, 347

        aliases, resolving, 352

        described, 136

    Trusted Content Services and, 188

    useACL method, 186

ACLs

    replacing, 189

acquired state, for work items, 275

actions on entry (lifecycle)

    aliases in, 329

    defining, 323

    described, 304

    Docbasic programs, 325

    execution order, 324

    Java programs, 324

active state (activity instance), 274

activities

    automatic, 238

    Begin, 235

    described, 235

    End, 235

    links between, 237

    manual, 238

    manual transitions

        output choice behavior, 257

        output choices, limiting, 257

    number of work items to complete,

        defining, 255

    performer categories, 241

    repeatable, 241

    starting conditions, 251

    Step, 235

    suspend timers, 261

    task subjects, 250

    timer implementations, 289

    transition behavior, 255

    transition types, 256

    trigger condition, 251

    trigger events, 251

    validation checks, 264

    warning timers, 260

activity definitions

    automatic transitions, 256

    control\_flag property, 240

    delegation, 240

    described, 237

    destroying, 298

    extension characteristic, 240

    installing, 265

- manual transitions, 256
  - modifying, 297
  - multiple use, 236, 241
  - names of, 236
  - package definitions
    - adding, 297
    - compatibility, 254
    - removing, 297
    - scope, 253
    - validation, 254
  - performers
    - alias as performer, 248
    - defining, 241, 247
  - ports
    - adding, 297
    - described, 252
    - input, 252
    - output, 252
    - removing, 297
    - requirements for, 252
    - revert, 252
  - prescribed transitions, 256
  - priority values, 238
  - route cases, described, 258
  - saving, 298
  - starting condition, defining, 251
  - states, 263
  - task subjects, defining, 250
  - transition\_eval\_cnt property, 256
  - transition\_flag property, 257
  - transition\_max\_output\_cnt property, 257
  - trigger conditions, 251
  - trigger events, 251
  - uninstalling, effects of, 297
  - validation, 263 to 264
  - XML files as packages, 253
  - XPath expressions and, 258
- activity instances
- alias resolution
    - alias\_category property, 355
    - default, 354
    - described, 282
    - errors, 355
    - package resolution algorithm, 354
    - user resolution algorithm, 354
  - completion, evaluating, 284
  - defined, 266
  - dm\_bpm\_transition method, 287
  - execution, 279, 282
  - halted workflows and, 293
  - halting, 293
  - packages
    - acceptance protocol, 288
    - consolidation of, 280
  - restarting failed, 294
  - resuming
    - automatically, 275
    - paused activities, 294
  - sequence number, obtaining, 294
  - starting condition
    - described, 251
    - evaluating, 280
  - states of, 274
  - timer instantiation, 290
  - transition conditions, evaluating, 287
  - user time and cost reporting, 292
  - work items
    - completing, 268
    - pausing and resuming, 295
- addAttachment method, 271
- addESignature method
- audit trail entries, 143
  - description of actions, 141
  - permissions needed, 145
- addNoteEx method, 194
- addPackage method, 279
- Addpackageinfo method, 297
- Addport method, 297
- addRendition method, 224
- AEK (Administration Encryption Key), 126
- alias set object type, 348
- alias sets
- alias set object type, 348
  - group alias sets, 350
  - lifecycle usage, 314
  - server configuration scope, 350
  - session alias set, 349
  - user alias sets, 350
- alias\_category property, 355
- alias\_name property, 348
- alias\_set property, 349 to 350
- alias\_set\_id property, 350
- alias\_set\_ids property, 314
- \_alias\_sets (computed property), 336
- alias\_value property, 348
- aliases
- alias set object type, 348
  - lifecycle scope, defining, 351

- lifecycle state actions, use in, 314, 329
  - methods, resolving in, 352
  - object types and, 347
  - purpose, 347
  - resolution
    - errors, 355
    - in activities, 282, 354
    - in lifecycles, 350
    - in workflows, 249, 349
    - non-workflow scopes, 350
  - scope, defined, 349
  - specification format, 248, 348
  - workflows, use in, 248
  - allow\_attach property, 308, 318
  - allow\_demote property, 320
  - allow\_schedule property, 320
  - ALTER TYPE (statement), 102
  - annotations
    - addNoteEx method, 194
    - creating, 194
    - deleting from repository, 195
    - described, 193
    - detaching, 195
    - effect of operations on, 195
    - keeping across versions, 194
    - workflow package notes, 270
  - application access control tokens
    - described, 52
    - dmtkgen utility, 54
    - expiration, 55
    - format, 53
    - generating, 54
    - getApplicationTokenDiagnostics method, 53
    - login ticket key use, 53
    - methods and, 55
    - scope, 54
    - superuser privileges and, 52
    - use of, 52
  - application access tokens
    - trusted repositories and, 57
  - application component classifiers, 97
  - application events, auditing, 137
  - application\_code property, 127
  - applications, *see* client applications
    - events, 340
  - approved\_clients\_only property, 150
  - ASCII use requirements, 362 to 363
  - aspect properties
    - described, 112
    - full-text indexing, 112
    - object types for, 112
    - query optimization, 112
  - aspects, 111
    - See also* aspect properties
    - described, 111
    - property bags and, 82
  - assemblies, *see* snapshots
  - assembling virtual documents, 211
  - assembly behavior, defining, 205, 210
  - assembly objects, 201
    - modifying, 215
  - assume method, 48
  - asynchronous write and retention, 171
  - attachability, lifecycle states, 318
  - attachments, for workflows, 270
  - attachPolicy method, 308
  - attr\_identifier property, 80
  - attr\_restriction property, 79
  - attributes
    - domains, 81
    - immutability and, 164
    - single-valued, modifying, 180
  - Audit method
    - described, 137
  - auditing
    - Audit method, 137
    - default auditing, 137
    - described, 136
    - registry objects and, 137
    - repository storage, 137
  - automatic activities
    - a\_wq\_name property, use of, 283
    - described, 237
    - effects of Complete method, 269
    - executing, 282
    - executing in dm\_bpm servlet, 238
    - execution queue size, 283
    - information passed to method, 283
    - mode parameter values, 284
    - performer categories for, 247
    - priority values, 238
    - resolving performers, 281
    - workflow agent, 239, 271
  - automatic activity transitions, 256
- ## B
- base permissions, *see* object-level permissions

- base state (lifecycle), returning to, 318
  - batch promotion, 309
  - BATCH\_PROMOTE (administration method), 309
  - Begin activities, 235
  - bindFile method, 183
  - binding
    - in lifecycles
      - actions on entry, 326
      - post-entry actions, 327
      - user criteria, 322
  - blob storage areas
    - digital shredding and, 152
  - BOF, *see* business object framework
  - BOF modules
    - caching, 110
    - client-side requirements, 109
    - components of, 108
    - dfc.bof.cache.currency\_check\_interval, 110
    - jar objects, 109
    - java library objects, 109
    - module objects, 108
    - overview, 107
    - packaging and deployment, 108
    - sandboxing, 109
    - simple, described, 113
    - testing in development mode, 110
  - Boolean expressions as entry criteria, 323
  - bp\_actionproc.ebs, 324
  - BP\_STATE\_EXTENSION relationship
    - name, 332
  - Branch method, 161
  - branching versions
    - defined, 160
    - numeric version labels and, 161
  - business object framework, 101
    - See also* aspects; service-based objects;
    - simple modules; type-based objects
    - described, 107
    - module overview, 107
- C**
- cabinets
    - destroying, 104
    - linking documents, 174
    - privileges to create, 103
  - cache config objects, 70
  - cache.map file, 67
  - caches
    - data dictionary cache, 65
    - object, 66
    - persistent client, 65
    - query, 65
    - query cache location, 66
  - cancelCheckOut method, 166
  - Centera profiles, 171
  - CHANGE...OBJECT (statement), 105
  - check constraints, 96
  - CHECK\_CACHE\_CONFIG
    - administration method, 70
  - Checkin method, 183
  - CheckinEx method, 183
  - Checkout method, 180
  - child, in foreign key, 95
  - classifiers, for application components, 97
  - client applications
    - a\_controlling\_app property, 127
    - aliases, use of, 347
    - application events, auditing, 137
    - application\_code property, 127
    - CheckinEx method, 183
    - component classifiers, 97
    - connection pooling, 48
    - control of SysObjects, 127
    - digital signatures, using, 146
    - lifecycle state types, using, 315
    - lifecycles and, 314
    - lifecycles, testing, 332
    - locking strategies, 165
    - persistent client caches, 65, 67
    - repository sessions, 39
    - roles
      - domain groups, 123
      - supporting groups, 121
    - virtual document components,
      - obtaining path to, 218
    - XML support, 202
  - client config object, 43, 360
  - client registration objects, 149
  - client rights objects, 149
  - client sessions, *see* repository sessions
  - client\_check\_interval property, 70
  - client\_pcaching\_change property, 71
  - close method, for IDfSessionManager, 43
  - code pages
    - ASCII support, 358
    - default\_client\_codepage property, 360

- group name requirements, 362
  - server, 359
  - user name requirements, 362
- Collaborative Edition, 35
- Collaborative Services, 35
- collection objects, 74
- compatibility, workflow package, 254
- complete method, 268
- component specifications (data dictionary), 97
- components (virtual document)
  - adding to snapshots, 214
  - adding to virtual documents, 209
  - assembly behavior, defining, 205
  - changing order, 210
  - deleting from snapshots, 215
  - removing, 210
- composite predicate objects, 259
- compound\_integrity property, 201
- computed properties
  - described, 78
  - \_is\_restricted\_session, 44
  - lifecycle specific, 323, 336
- concurrent sessions
  - defined, 58
- concurrent users, 58
- cond expr objects, 258
- conditional assembly, 201
- Config Audit user privileges, 129
- connection brokers
  - described, 45
  - purpose in installation, 45
- connection config object, 43
- connection pooling, 47
- connections
  - application access control tokens, 52
  - connection config object, 43
  - login tickets, 48
- consistency checking
  - cache config object use, 70
  - CHECK\_CACHE\_CONFIG
    - administration method, 70
  - client\_check\_interval property, 70
  - consistency check rules
    - default rule, 71
    - defined, 67
  - described, 68
  - DMCL behavior, 71
  - query results, 71
  - r\_last\_changed\_date property, 70
- constraints (data dictionary)
  - check, 96
  - defined, 93
  - foreign key, 95
  - not null, 96
  - primary key, 95
  - unique key, 94
- constraints on explicit transactions, 59
- containment objects
  - copy\_child property, 207
  - follow\_assembly property, 207
  - object type for, 199
  - updatePart method, 210
  - use\_node\_ver\_label property, 206
- content assignment policies, 175
  - overriding, 176
- content files
  - adding, 174
  - assigning to storage area, 175
  - association of primary and rendition, 156
  - bindFile method, 183
  - content assignment policies, 175
  - default storage algorithm, 175
  - digital shredding, 152
  - internationalization, 358
  - object types accepting, 77
  - page numbers, 156, 182
  - removing from documents, 104, 182
  - renditions and, 221
  - replacing, 182
  - sharing, 183
  - storage options, 89
  - virtual documents and, 202
- content objects
  - described, 155
- Content Server
  - communicating with, 37
  - compound\_integrity property, 201
  - concurrent sessions, 58
  - data dictionary, use of, 91
  - introduction, 25, 38
  - privileged DFC, recognition of, 150
  - supported format converters, 225
  - transactions, 58
  - user authentication, 124
- Content Services for EMC Centera
  - described, 32
- Content Storage Services
  - described, 33

Content Storage Services license, 175  
 Content Transformation Services, 222  
 content-addressed storage areas  
   metadata fields, setting, 176  
   retention periods, 168  
   retention policies, effect of, 169  
 control\_flag property, 240  
 convert.tbl file, 224  
 copy\_child property, 207  
 CREATE...TYPE (statement), 101  
 crypto\_key property, 151  
 CSEC, *see* Content Services for EMC  
   Centera  
 CSS, *see* Content Storage Services  
 CURRENT version label, 159  
 \_current\_state property (computed  
   property), 336  
 custom ACLs  
   creating, 186  
   naming convention, 186

## D

data dictionary  
   cache, 65  
   components for object types, 97  
   Content Server use of, 91  
   dd attr info objects, 93  
   dd common info objects, 93  
   dd type info objects, 93  
   default lifecycles for types, 97  
   described, 91  
   ignore\_immutable attribute, 165  
   lifecycle state information,  
     defining, 98  
   locales, supported, 92  
   localized text, 98  
   mapping information, 98  
   modifying, 92  
   object type constraints, 93, 96  
   property default values, 98  
   publishing, 92  
   retrieving information, 99  
   value assistance, 98  
 data validation, *see* check constraints  
 database-level locking, 60, 166  
 dd attr info objects, 93  
 dd common info objects, 93  
 dd type info objects, 93  
 deadlocks, managing, 60

default aspects  
   described, 113  
 default storage algorithm for content, 175  
 default values, for properties, 98  
 default\_acl property  
   default value, 185  
   use, 184  
 default\_app\_permit property, 128  
 default\_client\_codepage property, 360  
 default\_folder attribute, 173  
 delegation  
   control\_flag property, effects of, 240  
   defined, 240  
   enable\_workitem\_mgmt (server.ini  
     key), 240  
 deleting, *see* destroying; removing  
 deletions, forced, 171  
 demotion in lifecycles, 320  
 dequeue method, 344  
 destroying  
   activity definitions, 298  
   lifecycles, 337  
   objects, 104  
   process definitions, 298  
   versions, 161  
 development registry for BOF  
   modules, 110  
 DFC (Documentum Foundation  
   Classes), 148  
   *See also* privileged DFC  
   connection pooling, 47  
   data dictionary, querying, 100  
   dfc.properties file, 42  
   persistent caches, 65  
   sessions, implementation, 40  
 dfc.bof.cache.currency\_check\_  
   interval, 110  
 dfc.codepage property, 361  
 dfc.config.check\_interval key, 42  
 dfc.data.cache\_dir, 110  
 dfc.data.dir, 110  
 dfc.keystore file, 149  
 dfc.locale property, 361  
 dfc.properties file  
   described, 42  
   dfc.config.check\_interval key, 42  
   max\_session\_count, 40  
   verify\_registration key, 149  
 digital shredding  
   definition, 152

- implementation overview, 152
- digital signatures
  - a\_is\_signed property, 146
  - definition, 146
  - implementation overview, 146
  - lifecycle states, 329
- disassemble method, 215
- Disconnect method, 43
- discussions, described, 35
- distributed notification, 299
- distributed repositories
  - mirror objects, 190
- distributed storage areas
  - digital shredding and, 152
- distributed workflows, 298
- dm name prefixes, 76
- dm\_acl type, 134
- dm\_addesignature events, audit trail
  - entries, 143
- dm\_alias\_set type, 249, 348
- dm\_audittrail objects, 137
- dm\_audittrail\_acl objects, 137
- dm\_audittrail\_group objects, 137
- dm\_bp\_batch\_java method
  - log files, generating, 333
- dm\_bp\_schedule method, 312
- dm\_bp\_schedule\_java method, 312
- dm\_bp\_transition method
  - described, 312
  - log file, generating, 333
- dm\_bp\_transition\_java method
  - described, 312
  - log files, generating, 333
- dm\_bp\_validate method
  - described, 306
- dm\_bp\_validate\_java method
  - described, 306
  - log files, generating, 333
- dm\_bpactionproc (procedure), 324
- dm\_bpm servlet, 238
- dm\_bpm\_timer method, 291
- dm\_bpm\_transition method, 287
- dm\_changepriorityworkitem event, 268
- dm\_completedworkitem event, 292
- dm\_lightweight object type, 75
- dm\_lightweight type
  - i\_property\_bag property, 81
- dm\_owner
  - default object-level permissions, 132
- dm\_policy type, 305
- dm\_queue (view), 343
- dm\_relation\_type objects, 191
- dm\_retention\_managers group, 169 to 170
- dm\_retention\_users group, 169
- DM\_SESSION\_DD\_LOCALE
  - (keyword), 99
- dm\_sig\_template page modifier, 142
- DM\_TRANSLATION\_OF
  - relationship, 192
- dm\_type type
  - attr\_restriction property, 79
- dm\_user objects, 120
- dm\_WfmsTimer job, 261, 291
- dm\_WFReporting job
  - described, 292
- dm\_WFSuspendTimer job, 262, 291
- dm\_workflow objects, 266
- dmc\_completed\_workflow objects, 292
- dmc\_completed\_workitem objects, 292
- dmc\_jar objects, 109
- dmc\_java\_library objects, 109
- dmc\_module objects, 108
- dmc\_wf\_package\_schema type, 253
- dmc\_wf\_package\_skill type, 269
- DMCL (client library)
  - application access control token
    - retrieval, 55
- dmi\_package objects, 269
- dmi\_package type, 253
- dmi\_queue\_item objects, 342
- dmi\_wf\_attachment type, 270
- dmi\_wf\_timer objects, 261
- dmi\_workitem objects, 267
- dmSendToList2 workflow template, 234
- dmtkgen utility, 54
- docbase configuration
  - client\_pcaching\_change property, 71
- Docbasic
  - actions on entry (lifecycles), 325
  - entry criteria (lifecycles), 322
  - internationalization, 363
  - post-entry actions (lifecycles), 327
- documents, 154
  - See also* virtual documents
  - ACL, replacing, 189
  - ACLs, assigning, 178, 186
  - annotations, 193 to 195
  - attributes, setting, 180
  - bindFile method, 183

- Branch method, 161
  - branching, 160
  - checking out, 180
  - content files
    - adding, 174, 181
    - format characteristics, 223
    - Macintosh-generated, 174
    - page numbers, 156, 182
    - removing, 182
    - sharing, 183
    - specifying storage, 175
  - content objects and, 155
  - deleting with unexpired retention, 170
  - described, 154
  - dm\_document type, 154
  - fetching, 180
  - forced deletions, 171
  - immutability, 163
  - lifecycles and, 172
  - linking to cabinets/folders, 174
  - locking strategies, 165
  - modifying, 179
  - permissions, revoking, 188
  - primary location default, 174
  - privileged delete, 170
  - properties, modifying, 180
  - prune method, 161
  - renditions, 174
    - defined, 156, 221
    - removing, 171, 224
    - user-generated, 224
  - retention control
    - deletion and, 167
    - effect on modification ability, 179
  - saving, 183
  - translation relationships, 192
  - translation support, 157
  - versions
    - described, 157
    - removing, 161, 171
    - version tree, 160
  - Documentum Administrator, 37
  - domain groups, 123
  - domains, 81
  - dormant state
    - activity instance, 274
    - work items, 275
    - workflows, 273
  - DQL (Document Query Language)
    - data dictionary, querying, 99
    - query result objects, 74
  - draft state (workflow definitions), 263
  - DROP TYPE (statement), 102
  - DROP\_INDEX administration method, 89
  - dynamic groups
    - described, 123
    - non-dynamic group as member, 124
- ## E
- early binding
    - defined, 204
    - virtual document components, 204
  - electronic signatures, 139
    - See also* digital signatures; Signoff method
    - addESignature behavior, 141
    - content handling, default, 143
    - customizations allowed, 144
    - definition, 140
    - implementation overview, 140
    - lifecycle states, 329
    - PDF Fusion library and license, 142
    - signature creation method,
      - default, 142
    - signature page template, default, 142
    - verifying, 145
    - work items, 269
  - enable\_persistence dfc.properties key, 67
  - enable\_workitem\_mgmt (server.ini key)
    - delegation and, 240
    - halting activities and, 275, 293
    - priority values and, 268
    - purpose, 272
  - encryption
    - login ticket key, 56
    - password, 126
  - encryptPassword method, 126
  - End activities, 235
  - entry criteria (lifecycles)
    - Boolean expressions, 323
    - described, 304, 320
    - Docbasic programs, 322
    - Java programs, 321
    - \_entry\_criteria (computed property), 323
    - entry\_criteria\_id property, 323
  - esign\_pdf method object, 142
  - events
    - accessing, 340

- auditing, 136
- defined, 340
- getEvents method, 343
- notifications , 344
- queue items and, 340
- registrations for
  - establishing, 345
  - obtaining information about, 346
  - removing, 345
- trigger, for workflow activities, 251
- exception states (lifecycles)
  - described, 302
  - resuming from exception state, 310
- exceptional route case, 258
- execute method, 276, 279
- explicit sessions, 41
- explicit transactions
  - constraints, 59
  - database-level locking, 60, 166
  - deadlock management, 61
  - described, 59
- EXPORT\_TICKET\_KEY administration
  - method, 56
- extended permissions, *see* object-level permissions
- extension (workflow activities), 240
- extensions, *see* state extensions
- extents, for object type tables, 88
- external ACLs, 135
- external storage areas
  - digital shredding and, 152

## F

- failed state (activity instance), 274
- federated repositories
  - internationalization, 363
  - mirror objects, 190
- fetch method
  - locking and, 167
- Fetch method
  - described, 180
- file formats
  - conversions on Windows, 225
  - converter support, 222
  - PBM Image converters, 226
  - renditions, 222
    - described, 156, 221
  - supported converters, 225
  - transforming with Unix utilities, 227

- file store storage areas
  - crypto\_key property, 151
  - encrypted, 151
- files, *see* content files
- finished state
  - activity instance, 275
  - work items, 276
  - workflows, 273
- folder security, 133
  - default setting, 133
- folders
  - default ACL, 184
  - linking documents, 174
  - privileges to create, 103
- follow\_assembly property, 207
- forced deletions, 171
- foreign key constraints, 95
- format
  - alias specification, 248
  - application access control token, 53
  - login ticket, 49
- freeze method, 216
- Freeze method, 163
- full-text indexing
  - aspect properties, 112
  - SysObjects, 172

## G

- GET\_INBOX (administration
  - method), 343
- getApplicationTokenDiagnostics, 53
- getEvents method, 343
- getLoginTicketDiagnostics method, 49
- getSession method, 41
- getTypeDescription method, 100
- global properties, 79
- grantPermit method, 187
- group\_class property, 122
- groups, 121
  - See also* dynamic groups
  - alias sets for, 350
  - code page requirements, 362
  - described, 121
  - local and global, 124
  - membership constraint for
    - non-dynamic groups, 124
  - mixing non-dynamic and dynamic
    - groups in membership, 124
  - module role, 122

- ownership of objects, 173
- privileged group, 122
- role, 122
- standard, 122

## H

- Halt method
  - automatic resumption and, 261
  - use of, 293
- halted state
  - activity instances, 275
  - workflows, 273
- haltEx method, 275
- hasEvents method, 342
- hierarchy, object type, 74
- home repository, 341

## I

- i\_chronicle\_id attribute, 160
- i\_performer\_flag property, 285
- i\_property\_bag property, 81
- i\_property\_bat property
  - overflow, 82
- i\_retain\_until property, 169
- i\_vstamp attribute, locking and, 167
- i\_vstamp property, 71
- identifiers
  - object type, 106
  - property, 80
- identifiers, session, 41
- IDfWorkflowAttachment interface, 271
- ignore\_immutable attribute, 165
- images
  - PMB Image converters, 226
  - transforming, 226
- immutability
  - attributes, effect on, 164
  - described, 163
  - ignore\_immutable attribute, 165
  - retention policies and, 164
- implicit sessions, 41
- IMPORT\_TICKET\_KEY administration
  - method, 56
- inboxes
  - dequeue method, 344
  - description, 341
  - dm\_queue view, 343
  - dmi\_queue\_item objects, 342

- GET\_INBOX (administration
  - method), 343
- getEvents method, 343
- hasEvents method, 342
- home repository and, 341
- obtaining event registrations, 346
- queue method, 343
- viewing queue, 342
- \_included\_types (computed
  - property), 336
- indexes for object types, 89
- inheritance, 74
- input ports
  - Begin activities and, 237
  - defined, 252
  - packages, 280, 288
- installed state (workflow definitions), 263
- internal ACLs, 135
- internal transactions
  - deadlock management, 61
  - described, 59
- internationalization
  - ASCII support, 358
  - ASCII use requirements, 362
  - client config object, 360
  - code pages, 359
  - content files, 358
  - data dictionary support for, 98
  - databases, 359
  - described, 357
  - Docbasic, 363
  - federations, ASCII requirements, 363
  - lifecycle constraint, 363
  - locales, 357
  - metadata, 358
  - National Character Sets, 36
  - object replication, 364
  - repository sessions, values set, 360
  - required parameters, 359
  - server config object, 360
  - session config object, 361
  - Unicode, 36
  - UTF-8, 36
- \_is\_restricted\_session (computed
  - property), 44

## J

- JAR files
  - jar objects, 109

jar objects, 109  
 Java  
   action on entry programs (lifecycles), 324  
   entry criteria (lifecycles), 321  
   post-entry actions (lifecycles), 326  
 java library objects, 109  
 jobs  
   dm\_WfmsTimer, 261, 291  
   dm\_WFReporting, 292  
   dm\_WFSuspendTimer, 262, 291  
   lifecycle state transition jobs, 311

## K

keep flag setting, 224

## L

language\_code attribute, 192  
 late binding  
   post-entry actions (lifecycles), 327  
   user actions (lifecycles), 326  
   user criteria (lifecycles), 322  
 lifecycle states  
   actions on entry  
     defining, 323  
     Docbasic, 325  
     execution order, 324  
     Java, 324  
     system-defined, 324  
   aliases in, 314  
   attachability, 318  
   attaching objects, 308  
   base state, returning to, 318  
   batch promotion, 309  
   computed properties for, 336  
   data dictionary and, 98  
   definitions  
     described, 317  
     modifying, 335  
   demoting from, 309, 320  
   entry criteria  
     described, 320  
     Docbasic, 322  
     entry\_criteria\_id property, 323  
     Java, 321  
   movement between, 308  
   naming rules, 318  
   normal states, 302

post-entry actions  
   described, 326  
   Docbasic, 327  
   Java, 326  
 promoting to, 308  
 sign-offs, enforcing, 329  
 state extensions  
   adding, 331  
   described, 315  
 state type definitions, 315  
 state\_class property, 317  
 suspending from, 309  
 transitions, scheduled, 311, 320  
 user\_action\_ver property, 326  
 user\_criteria\_id property, 322  
 user\_criteria\_ver property, 322  
 user\_postproc\_ver property, 327  
 lifecycles  
   a\_bpaction\_run\_as property, 313  
   action\_object\_id property, 324  
   actions, described, 304  
   alias scope, defining, 351  
   alias use in actions, 329  
   aliases and alias sets, 314  
   allow\_demote property, 320  
   attaching objects, 308  
   batch promotion, 309  
   bp\_actionproc.ebs, 324  
   changing, 335  
   code page requirements, 363  
   computed properties for, 336  
   defaults for object types, 97, 305  
   defined, 172, 301  
     *See also* lifecycle states  
   definition states, 305  
   demotion, 309  
   destroying, 337  
   entry criteria, described, 304  
   exception states, 302  
   installation, 306  
   log files, 313  
   methods supporting, 311  
   notifications of uninstall/reinstall, 335  
   object types for, 303  
   object-level permissions and, 314  
   primary object type for, 304  
   programming languages,  
     supported, 304  
   progression through, overview, 307

- promotion, 308
  - repository storage, 305
  - resumption from exception state, 310
  - state change behavior, 312
  - state definitions, 317
  - state extension relationship name, 332
  - state extensions, 315, 331
  - state types, 315
  - state-change methods, 308, 312
  - suspension from state, 309
  - system\_action\_state property, 324
  - testing and debugging, 332
  - validation
    - custom programs, 306, 334
    - overview, 306
  - lightweight object types
    - database storage, 85
    - defined, 75
    - materialization, 86
  - lightweight objects
    - i\_property\_bag property, 81
  - Link method
    - alias use in, 347
    - resolving aliases, 352
  - linking, folder security and, 133
  - links
    - described, 237
  - links (workflow)
    - compatibility, 264
    - port compatibility, 255
  - local properties, 79
  - locale\_name property, 360
  - locales
    - described, 92
    - DM\_SESSION\_DD\_LOCALE
      - keyword, 99
    - session\_locale property, 361
    - supported, 92, 357
  - location objects
    - SigManifest, 143
  - locking
    - database level, 60, 166
    - optimistic, 167
    - repository level, 166
    - strategies, 165, 167
  - log files, lifecycle, 313, 333
  - login ticket
    - definition, 48
  - login ticket key
    - application access control tokens
      - and, 53
      - defined, 56
      - resetting, 56
      - ticket\_crypto\_key property, 56
      - use of, 56
    - login tickets
      - described, 48
      - expiration, configuring, 50
      - format, 49
      - generation, 49
      - getLoginTicketDiagnostics
        - method, 49
      - global, defined, 49
      - login\_ticket\_cutoff property, 51
      - login\_ticket\_timeout property, 50
      - max\_login\_ticket\_timeout
        - property, 50
      - revoking, 51
      - scope, 49
      - single use, 48
      - superuser use, restricting, 51
      - time difference tolerances, 50
      - timed-out sessions, reconnecting, 44
      - trusted repositories and, 57
    - login\_ticket\_cutoff property, 51
    - login\_ticket\_timeout property, 50
    - LTK, *see* login ticket key
- ## M
- Macintosh files, adding as content, 174
  - MAKE\_INDEX administration
    - method, 89
  - manual activities
    - delegation, 240
    - described, 237
    - effects of complete method, 268
    - extension characteristic, 240
    - manual transitions
      - described, 256
      - output choice behavior, 257
      - output choices, limiting, 257
    - performers
      - alias use, 248
      - categories of, 246
      - resolving, 281
      - priority values, 238
  - mapping information (data dictionary), 98
  - mark method, 159

materialization, 86  
 max\_login\_ticket\_timeout property, 50  
 max\_session\_count, 40  
 Media Transformation Services, 222  
 metadata  
   described, 78  
   national character sets and, 358  
   setting in content-addressed  
     storage, 176  
 method objects  
   workflow, executing in dedicated  
     servlet, 238  
 methods  
   application access control tokens  
     and, 55  
   lifecycle, 311  
 mirror objects  
   described, 190  
 mode parameter values, 284  
 module objects, 108  
 module role groups, 122  
 modules, *see* BOF modules

## N

names  
   activity definitions, 236  
   custom ACLs, 186  
   lifecycle states, 318  
   object type name prefixes, 76  
 National Character Sets, 36  
 newSession method, 41  
 \_next\_state (computed property), 336  
 non-persistent objects, 74  
 non-qualifiable properties, 79  
 normal states (lifecycle), 302  
 not null constraints, 96  
 notes, 35  
   *See also* annotations  
 notification in distributed workflows, 299  
 notifications of events, 344  
 NULL values  
   foreign keys and, 95  
   not null constraints, 96  
   unique keys and, 94  
 numeric version labels, 158

## O

object caches

  consistency checking, 71  
   described, 65  
 object replication  
   internationalization, 364  
 object type tables  
   described, 83  
   extent size, defining, 88  
   lightweight object types, 85  
   subtype storage, 84  
   tablespace, defining, 88  
 object types  
   categories of, 74  
   component routines for, 97  
   content files and, 77  
   creating, 101  
   data dictionary information  
     constraints, 93  
     default lifecycle for type, 97, 305  
     mapping information, 98  
     value assistance, 98  
   (data dictionary information  
     constraints, 96  
   data dictionary information,  
     retrieving, 99  
   dd\_attr\_info, 93  
   dd\_common\_info, 93  
   dd\_type\_info, 93  
   default ACLs for, 185  
   defined, 73  
   dm name prefix, 76  
   identifiers, 106  
   indexes on, 89  
   lightweight, 75  
   non-qualifiable and qualifiable  
     properties, 79  
   owner, 101  
   persistence, 74  
   primary for lifecycles, 304  
   properties, defaults for, 98  
   RDBMS tables for, 82  
   removing, 102  
   shareable, 76  
   subtypes, described, 74  
   supertypes, 74  
   SysObjects, 153  
   valid types for lifecycles, 303  
 object-level permissions, 130  
   *See also* ACLs  
   assigning, 178  
   base permission levels, 130

- described, 129
  - extended permission levels, 131
  - lifecycles and, 314
  - revoking, 188
  - objects
    - alias use in, 347
    - attaching to lifecycles, 308
    - changing to another type, 105
    - creating, 103
    - default owner permissions, 132
    - default superuser permissions, 132
    - defined, 73
    - destroying, 104
    - freezing, 163
    - global and local properties, 79
    - ignore\_immutable attribute, 165
    - immutability, 163
    - information about, obtaining through
      - DFC, 100
    - ownership, assigning, 173
    - persistence, 74
    - privileges to create, 103
    - relationships
      - user-defined, 191
    - unfreezing, 164
  - optimistic locking, 167
  - output ports
    - defined, 252
    - End activities and, 237
    - selecting conditionally, 258
  - ownership of objects, 173
- P**
- package control (workflows), 262
  - package definitions
    - adding, 297
    - compatibility, 254
    - described, 253
    - empty, 253
    - removing, 297
    - scope, 253
    - validation of, 254
  - package object type, 253
  - packages
    - acceptance protocol, 288
    - adding to Begin activities, 279
    - consolidation of, 280
    - empty, 253
    - package objects, 269
      - package skill level, 269
      - skill level requirements, 254
      - visibility of, 254
  - page numbers, for content, 156, 182
  - parent, in foreign key, 95
  - password encryption, 126
  - path\_name property, 218
  - paused state, for work items, 276
  - PBM Image converters, 226
  - PDF Fusion library and license, 142
  - percent sign (%) in alias specification, 348
  - performance
    - optimizing for aspect properties, 112
  - performer aliases, format, 248
  - permanent\_link attribute, 194
  - persistence, 74
  - persistent attributes
    - domains, 81
  - persistent caching
    - described, 65
  - persistent client caches
    - cache.map file, 67
    - consistency checking, 68
    - enable\_persistence dfc.properties
      - key, 67
    - flushing, 71
  - object caches
    - consistency checking, 71
    - described, 66
    - query cache location, 66
    - query caches, 65, 67
    - subconnections and, 68
    - using, 67
  - PKI credentials, location, 149
  - policy object type, 305
  - \_policy\_name property (computed property), 336
  - ports, 252
    - See also* input ports; output ports
    - adding to activities, 297
    - compatibility, 255, 264
    - described, 252
    - package definitions
      - described, 253
      - validation, 254
    - removing from activities, 297
    - required, 252
  - post-entry actions (lifecycles)
    - aliases in, 329
    - defining, 326

- described, 304
  - Docbasic programs, 327
  - Java programs, 326
  - post-timers (workflows)
    - described, 260
    - instantiation, 290
  - pre-timers (workflows)
    - described, 260
    - instantiation, 290
  - predicate\_id property, 259
  - prescribed activity transitions, 256
  - \_previous\_state (computed property), 336
  - primary cabinet, *see* primary location
  - primary content files, *see* content files
  - primary folder, 184
  - primary key constraints, 95
  - primary location, 173
  - priority values
    - activity (workflow), 238
    - work items, setting for, 268
  - private ACLs
    - assigning to documents, 186
    - described, 136
  - private groups, 122
  - private sessions, 41
  - privileged delete, 170 to 171
    - See also* forced deletions
  - privileged DFC
    - approved\_clients\_only property, 150
    - client registration objects, 149
    - client rights objects, 149
    - Content Serve, recognition by, 150
    - described, 148
    - public key certificate objects, 149
    - registration described, 149
    - verify\_registration key (dfc.properties file), 149
  - privileged group, 122
  - Process Builder, 234
  - process definitions, 235
    - See also* workflow definitions
    - activity types, 235
    - described, 235
    - destroying, 298
    - installing, 265
    - links, 237
    - modifying, 295
    - re-installing, 296
    - uninstalling, 295
  - validating, 263
  - validation checks, 264
  - versioning, 296
  - promote method, testing, 332
  - properties, *see* properties
    - a\_full\_text, 172
    - characteristics of, 78
    - constraints (data dictionary), 93, 96
    - data dictionary information,
      - retrieving, 99
    - datatypes, 78
    - default values, defining, 98
    - defined, 77
    - global and local, 79
    - identifiers, 80
    - mapping information (data dictionary), 98
    - non-qualifiable, 79
    - qualifiable, 79
    - RDBMS tables for, 83
    - repeating, 78
    - repeating, modifying, 180
    - single-valued, 78
    - value assistance (data dictionary), 98
  - property bag
    - aspect attributes and, 82
    - described, 81
  - prune method, 161
  - public ACLs
    - assigning to documents, 186
    - described, 135
  - public groups, 122
  - public key certificate objects, 149
  - publishing data dictionary, 92
  - Purge Audit user privileges, 129
- ## Q
- qual comp objects, 97
  - qualifiable properties, 79
  - query caches, 65, 67
    - storage location, 66
  - query result objects, 74
  - queue items
    - placing in inbox, 343
    - work items and, 267
  - queue method, 343
  - queues, *see* inboxes

## R

- `_r` repository tables, 83
- `r_act_priority` property, 239
- `r_alias_set_id` property, 350
- `r_complete_witem` property, 284
- `r_condition_id` property, 258
- `r_condition_name` property, 260
- `r_condition_port` property, 260
- `r_current_state` property, 337
- `r_definition_state` property, 305
- `r_frozen_flag` attribute, 164
- `r_frozen_flag` property, 216 to 217
- `r_frzn_assembly_cnt` property, 216 to 217
- `r_has_frzn_assembly` property, 216 to 217
- `r_immutable_flag` attribute, 163
- `r_immutable_flag` property, 216 to 217
- `r_is_virtual_doc` property, 200
- `r_last_changed_date` property, 70
- `r_link_cnt` property, 200
- `r_policy_id` property, 337
- `r_predicate_id` property, 259
- `r_property_bag` property, 82
- `r_resume_state` property, 337
- `r_total_witem` property, 284
- `r_version_label` attribute, 158
- RDBMS
  - database-level locking, 166
  - Documentum tables in, 82
  - object type indexes, 89
  - `_r` repository tables, 83
  - registered tables, 90
  - `_s` repository tables, 83
- referential integrity, 201
- registered tables, 90
- `registerEvent` method, 345
- registry objects, 137
- relation objects
  - annotations and, 193
  - described, 191
- relationships
  - annotations, 193
  - defined, 191
  - relation object, 191
  - system-defined, 191
  - translation relationships, 192
  - user-defined, 191
- remote users, work items and, 299
- `removeAttachment` method, 271
- `removeContent` method, 182
- `removeNote` method, 195
- `Removepackageinfo` method, 297
- `removePart` method, 210
- `Removeport` method, 297
- `removeRendition` method, 224
- removing, 182
  - See also* destroying
  - aborted workflows, 294
  - content files, 182
  - event registrations, 345
  - queued inbox items, 344
  - renditions, 224
  - user-defined types, 102
  - versions, 161
  - virtual document components, 210
- rendition property, use of, 224
- renditions
  - adding user generated, 224
  - `addRendition` method, 224
  - connection to source document, 156
  - `convert.tbl` file, 224
  - converter support, 222
  - custom converters, adding, 228
  - defined, 156, 221
  - described, 174
  - file formats, 222
  - format characteristics, 223
  - keep flag setting, obtaining, 224
  - Media Transformation Services, 222
  - page numbers and, 156
  - PBM Image converters, 226
  - `removeRendition` method, 224
  - removing, 171
  - removing user-generated, 224
  - supported format converters, 225
  - system-generated, 223
- `Repeat` method, 240
- `repeatable_invoke` property, 241
- repeating properties, 78
  - performance tip, 181
  - replacing values, 180
  - storage, 83
- replicas
  - described, 190
  - retention policies and, 169
- repositories
  - `a_bpaction_run_as` property, 313
  - aborted workflows, removing, 294
  - annotations, deleting, 195
  - application access control tokens, 52

- application access control tokens,
  - generating, 54
- approved\_clients\_only property, 150
- architecture, 82
- auditing events, 136
- data dictionary
  - described, 91
  - retrieving information, 99
- default\_acl property, 184
- events, 340
- home repository, 341
- inboxes, 341
- localizing, 98
- login ticket use, 48
- object types
  - creating, 101
  - RDBMS indexes, 89
  - RDBMS tables, 82
  - removing, 102
- repository objects
  - changing type, 105
  - creating, 103
  - destroying, 104
- security, 119
- trusted mode, for connection
  - requests, 57
- user authentication, 124
- views, 91
- working with remote objects, 189
- repository sessions
  - alias sets for, 349
  - closing, 43
  - configuration, 42
  - configuration objects, 43
  - connection pooling, 47
  - deadlocks, managing, 60
  - default\_app\_permit property, 128
  - defined, 39
  - explicit sessions, 41
  - explicit transactions, 58
  - identifiers, 41
  - implicit sessions, 41
  - inactive, 44
  - internationalization, 360
  - \_is\_restricted\_session (computed property), 44
  - max\_session\_count, 40
  - maximum number, 58
  - object state, 43
  - private, 41
  - restricted, 44
  - secure connections, 46
  - session config objects, 361
  - session\_codepage property, 361
  - session\_locale property, 361
  - shared, 41
  - timed out sessions, reconnecting, 44
  - transactions, managing, 58
- repository-level locking, 166
- resetPassword (IDfSession), 44
- resolve\_pkg\_name property, 249, 354
- resolve\_type property, 249, 354
- respository sessions
  - default\_client\_codepage property, 360
- resume method, 294, 332
- \_resume\_state (computed property), 337
- retention and asynchronous write operations, 171
- retention policies
  - deleting documents under control, 170
  - described, 168
  - description, 168
  - dm\_retention\_managers group, 169
  - dm\_retention\_users group, 169
  - document versions and, 169
  - privileged delete, 170
  - r\_immutable\_flag and, 164
  - replicas and, 169
  - storage-based retention, interaction with, 169
  - SysObjects, modification constraints, 179
  - virtual documents, 203
- Retention Policy Services, *see* retention policies
  - described, 34
- return\_to\_base property, 318
- revert ports, 252
- revokePermit method, 187
- role groups, 122
- rooms
  - ACL assignments for governed objects, 187
- rooms, described, 35
- route cases
  - described, 258
  - evaluation, 287
  - non-XPath implementation, 258

- r\_condition\_id and r\_predicate\_id
  - compatibility, 259
  - selected ports, recording in repository, 260
  - XPath implementation, 259
- RPS, *see* Retention Policy Services
- S**
- \_s repository tables, 83
- sandboxing of Java libraries, 109
- Save method
  - described, 183
- SBO (service-based object)
  - described, 110
  - storage location, 111
- scope
  - alias, 349
  - application access control tokens, 54
  - login tickets, 49
- secure connections, described, 46
- security
  - cached query files, 67
  - digital shredding, 152
  - digital signatures, 146
  - folder security, 133
  - permissions, revoking, 188
  - secure connections, use of, 46
  - signature requirements support, 139
- security\_mode property, 119
- SELECT (statement)
  - processing algorithm, 212
- server config object, 43, 360
- server.ini file
  - enable\_workitem\_mgmt key, 272
- server\_os\_codepage property, 360
- service-based object, *see* SBO (service-based object)
- session code page
  - session\_codepage property, 361
- session config objects, 43, 361
- session configuration, *see* repository sessions
- session managers
  - terminating, 43
- session objects (DFC), 40
- session\_locale property, 361
- sessions, *see* repository sessions
  - DFC implementation, 40
- SET\_APIDEADLOCK administration
  - method, 61
- Setoutput method, 257
- setPriority (IDfWorkitem) method, 239
- Setpriority method, 268
- shareable object types
  - defined, 76
- shared sessions, 41
- shredding, digital, *see* digital shredding
- SigManifest location object, 143
- sign-offs, simple
  - definition, 147
  - implementation overview, 147
- signature creation method, default, 142
- signature page templates, default, 142
- signature requirements, support for, 139
  - See also* digital signatures; electronic signatures; sign-offs, simple
- signatures, digital, *see* digital signatures
- signatures, electronic, *see* electronic signatures
- signoff method
  - simple sign-off, use in, 147
- sigpage.doc, 142
- sigpage.pdf, 142
- simple modules, 113
- single-valued properties, 78
- snapshots
  - assembly object type, 201
  - components
    - adding, 214
    - deleting, 215
  - creating, 214
  - described, 201, 213
  - disassemble method, 215
  - freezing, 216
  - modifying, 214
  - path\_name property, 218
  - unfreezing, 217
- SSL (secure socket layer) protocol, 46
- standard groups
  - described, 122
- starting condition (activities)
  - described, 251
  - evaluating, 280
- state extensions
  - adding to lifecycles, 331
  - described, 315
  - instance properties, 332
  - relationship name, 332

- state types, for lifecycles, 315
  - state\_class property, 317
  - \_state\_type (computed property), 336
  - states
    - work items, 275
    - workflow, 273
    - workflow definition objects, 263
  - Step activities
    - described, 235
    - required ports, 252
  - storage areas
    - assigning content to, 175
    - digital shredding, 152
    - retention periods, 169
  - subconnections
    - persistent client caching and, 68
  - subtypes
    - database storage, 84
    - described, 74
    - owner, 101
    - removing, 102
  - supertypes, 74
  - Superuser user privilege
    - application access control tokens
      - and, 52
    - login tickets and, 51
    - workflow supervisor and, 271
  - superusers
    - default object-level permissions, 132
  - supervisor, workflow, 271
  - supported format converters, 225
  - suspend method, 310
  - suspend timers
    - described, 261
    - dm\_WFSuspendTimer job, 291
    - implementation, 289
    - instantiation, 290
  - symbolic version labels, 158
  - Sysadmin user privilege
    - workflow supervisor and, 271
  - SysObjects
    - ACL assignments in rooms, 187
    - ACLs and, 184
    - alias use in, 347
    - annotations, adding, 194
    - application-level control, 127
    - attachPolicy method, 308
    - content files
      - adding, 181
      - replacing, 182
    - default\_folder attribute, 173
    - described, 153
    - full-text indexing, 172
    - i\_chronicle\_id attribute, 160
    - i\_retain\_until property, 169
    - lifecycle scope, defining, 351
    - modifying, 179
    - ownership, assigning, 173
    - primary folder, defined, 184
    - property bags, 81
    - r\_alias\_set\_id property, 350
    - r\_is\_virtual\_doc property, 200
    - r\_link\_cnt property, 200
    - r\_property\_bag property, 82
    - r\_version\_label attribute, 158
    - removeContent method, 182
    - repeating properties, modifying, 180
    - resolving aliases, 351
    - retention control, effects of, 167, 179
    - retention policies, 168
    - state information, obtaining, 336
    - version labels, 157
    - version tree, 160
  - system ACLs
    - assigning to documents, 186
    - public ACLs and, 135
- ## T
- tablespaces for object type tables, 88
  - task subjects, 250
  - task\_subject property, 250
  - task\_subject property (activities), 250
  - task\_subject property (queue items), 251
  - tasks, workflow
    - accessing, 340
    - defined, 339
    - queue items and, 340
    - work items and, 340
  - TBO (type-based object)
    - described, 111
    - storage location, 111
  - TCS, *see* Trusted Content Services
  - template ACLs, 136
  - template signature pages, 142
  - template workflows
    - alias usage, 248
    - described, 233
    - dmSendToList2, 234
  - terminated state (workflows), 273

- ticket\_crypto\_key property, 56
  - timers, for workflow, 260
  - tokens, *see* application access control
    - tokens
  - tracing, overview, 138
  - transactions
    - database-level locking, 166
    - deadlocks, managing, 60
    - defined, 58
    - locking strategies, 165, 167
  - TRANSCODE\_CONTENT administration
    - method, 222
  - transformations
    - PBM Image converters, 226
    - renditions, 221
    - supported converters, 225
    - using UNIX utilities, 227
  - transition condition objects, 259
  - transition types, for activities, 256
  - transition\_eval\_cnt property, 256
  - transition\_flag property, 257
  - transition\_max\_output\_cnt property, 257
  - transitions, activity, 255
  - translation relationships, 192
  - translations of documents, 157
  - trigger condition (activities), 251
  - trigger events (activities), 251
  - trust\_by\_default property, 57
  - Trusted Content Services
    - ACLs and, 135, 188
    - described, 31
  - trusted\_docbases property, 57
  - turbo storage areas
    - digital shredding and, 152
  - type identifiers, 106
  - type-based object, *see* TBO (type-based object)
  - type\_category property, 75
- ## U
- Unfreeze method, 164, 217
  - Unicode, 36
  - unique key constraint, 94
  - Unlink method
    - alias use in, 347
    - resolving aliases, 352
  - unlinking, folder security and, 133
  - unmaterialize, 86
  - unRegister method, 345
  - updatePart method, 210
  - use\_node\_ver\_label property, 206
  - useACL method, 186
  - user authentication, 124
  - user privileges
    - basic, list of, 128
    - Config Audit, 129
    - extended, list of, 129
    - Purge Audit, 129
    - View Audit, 129
  - user\_action\_id property, 325
  - user\_action\_service property, 324
  - user\_action\_ver property, 326
  - user\_criteria\_id property, 322
  - user\_criteria\_ver property, 322
  - user-defined types, removing, 102
  - user\_delegation property, 240
  - user\_postproc\_id property, 327
  - user\_postproc\_ver property, 327
  - users
    - alias sets for, 350
    - alias\_set\_id property, 350
    - code page requirements, 362
    - default ACL, 185
    - described, 120
    - dm\_user objects, 120
    - local and global, 121
    - object-level permissions, 130
    - privileges to create, 103
    - user privileges, list of, 128
  - UTF-8, 36
  - utilities
    - dmtkgen, 54
- ## V
- validated state (workflow definitions), 263
  - validation, lifecycle definitions, 306, 334
  - validity periods
    - application access control tokens, 55
    - login tickets, 50
  - value assistance, 98
  - vdmPath method, 219
  - vdmPathDQL method, 219
  - verification of electronic signatures, 145
  - verify\_registration key, 149
  - verifyESignature method, permissions
    - needed, 145
  - version trees
    - changeable versions, 162

- versioning
    - defined, 157
  - versions
    - Branch method, 161
    - branching, 160
    - changeable, 162
    - Destroy method and, 161
    - i\_chronicle\_id attribute, 160
    - label uniqueness, 159
    - mark method and, 159
    - numeric version label, 158
    - prune method and, 161
    - r\_version\_label attribute, 158
    - removing, 104, 161, 171
    - retention policies and, 169
    - symbolic version label, 158
    - SysObjects and, 157
    - version tree, 160
  - View Audit user privileges, 129
  - view on inboxes, 343
  - views, on repository tables, 91
  - virtual documents
    - assembling, 211
    - components
      - adding, 209
      - appending, 208
      - assembly behavior, defining, 205, 210
      - determining paths to, 218
      - early binding, 204
      - ordering, 200, 210
      - removing, 210
    - compound\_integrity property, 201
    - conditional assembly, 201
    - containment objects
      - copy\_child property, 207
      - described, 199
      - updatePart method, 210
    - content files and, 202
    - copy behavior, defining, 207
    - creating, 208
    - described, 154, 198
    - freezing, 216
    - permissions to modify, 209
    - querying, 217 to 218
    - r\_is\_virtual\_doc property, 200
    - r\_link\_cnt property, 200
    - referential integrity, 201
    - retention policies, 203
    - snapshots
      - assembly objects, 201
      - creating, 214
      - described, 201, 213
      - disassembling, 215
    - unfreezing, 217
    - vdmPath method, 219
    - vdmPathDQL method, 219
    - versioning, 200
- ## W
- warning timers
    - described, 260
    - dm\_WfmsTimer job, 291
    - implementation, 289
  - Webtop Workflow Reporting tool, 292
  - wf attachment objects, 270
  - wf package schema object type, 253
  - wf package skill object type, 269
  - WfmsTimer job, 291
  - Windows platforms, format conversions
    - supported, 225
  - work items
    - completing, 268
    - delegating, 240
    - described, 267
    - overview of use, 267
    - package skill level, 254
    - pausing, 295
    - priority, setting, 268
    - queue items and, 267
    - remote users and, 299
    - resuming paused, 295
    - signing off, 269
    - states of, 275
    - user time and cost reporting, 292
  - work queues
    - priority values for work items, 239
  - workflow agent
    - activities, assigning, 282
    - activity priority value use, 239
    - batch size, 282
    - described, 271
  - workflow definitions
    - activities, naming, 236
    - activity definitions
      - modifying, 297
      - multiple use of, 236
    - alias use in, 233, 347

- architecture, 235
  - Begin activities, 235
  - described, 237
  - End activities, 235
  - modifying, 295
  - package compatibility, 254
  - package control, enabling, 262
  - process definitions
    - modifying, 295
    - validation checks, 264
  - states, 263
  - Step activities, 235
  - templates, 233
  - validating, 263
  - workflow instances
    - aborting, 294
    - activity instances, 266
      - completion, evaluating, 284
      - evaluating transition
        - conditions, 287
      - halting, 293
    - attachments, 270
    - halting
      - described, 293
      - effect on activities, 293
    - notes, for packages, 270
    - reinstalling definition, effects of, 296
    - reports about, 292
    - resolving performers
      - aliases, 282
      - automatic activities, 281
      - manual activities, 281
      - workflow\_disabled property, 281
    - restarting
      - failed activities, 294
      - workflow, 294
    - resuming
      - halted workflows, 294
      - paused activities, 294
    - starting, 276, 279
    - states of, 273
  - workflow objects, 266
  - workflow worker thread
    - activity execution, described, 283
  - workflow\_disabled property, 281
  - workflows, 232
    - See also* activities; workflow definitions; workflow instances
  - activities
    - definitions, 237
    - delegation, 240
    - execution, 279
    - extension, 240
    - instances of, 266
    - names of, 236
    - repeatable, 241
    - types of, 235
  - aliases
    - default resolution, 354
    - package resolution algorithm, 354
    - resolving at activity start, 353
    - resolving at startup, 353
    - scopes of, 349
    - user resolution algorithm, 354
  - attachments, 270
  - defined, 232
  - distributed, 298
  - dm\_bpm servlet, 238
  - implementation overview, 232
  - input ports, 252
  - links, 237
  - manual activities, 237
  - output ports, 252
  - package definitions, 253
  - revert ports, 252
  - route cases, 258
  - runtime architecture, 266
  - runtime execution, 276
  - starting, 276, 279
  - supervisor, 271
  - tasks, defined, 339
  - timers, for activities, 260
  - user time and cost reporting, 292
  - work item objects, 267
  - workflow agent, 271
- X**
- XML files
    - activity packages, as, 253
    - support for, 202
  - XPath expressions
    - route cases and, 259
    - validation, 258