

CA Identity Manager

Programming Guide for Java Connector Server

r12



This documentation and any related computer software help programs (hereinafter referred to as the "Documentation") is for the end user's informational purposes only and is subject to change or withdrawal by CA at any time.

This Documentation may not be copied, transferred, reproduced, disclosed, modified or duplicated, in whole or in part, without the prior written consent of CA. This Documentation is confidential and proprietary information of CA and protected by the copyright laws of the United States and international treaties.

Notwithstanding the foregoing, licensed users may print a reasonable number of copies of the Documentation for their own internal use, and may make one copy of the related software as reasonably required for back-up and disaster recovery purposes, provided that all CA copyright notices and legends are affixed to each reproduced copy. Only authorized employees, consultants, or agents of the user who are bound by the provisions of the license for the Product are permitted to have access to such copies.

The right to print copies of the Documentation and to make a copy of the related software is limited to the period during which the applicable license for the Product remains in full force and effect. Should the license terminate for any reason, it shall be the user's responsibility to certify in writing to CA that all copies and partial copies of the Documentation have been returned to CA or destroyed.

EXCEPT AS OTHERWISE STATED IN THE APPLICABLE LICENSE AGREEMENT, TO THE EXTENT PERMITTED BY APPLICABLE LAW, CA PROVIDES THIS DOCUMENTATION "AS IS" WITHOUT WARRANTY OF ANY KIND, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IN NO EVENT WILL CA BE LIABLE TO THE END USER OR ANY THIRD PARTY FOR ANY LOSS OR DAMAGE, DIRECT OR INDIRECT, FROM THE USE OF THIS DOCUMENTATION, INCLUDING WITHOUT LIMITATION, LOST PROFITS, BUSINESS INTERRUPTION, GOODWILL, OR LOST DATA, EVEN IF CA IS EXPRESSLY ADVISED OF SUCH LOSS OR DAMAGE.

The use of any product referenced in the Documentation is governed by the end user's applicable license agreement.

The manufacturer of this Documentation is CA.

Provided with "Restricted Rights." Use, duplication or disclosure by the United States Government is subject to the restrictions set forth in FAR Sections 12.212, 52.227-14, and 52.227-19(c)(1) - (2) and DFARS Section 252.227-7014(b)(3), as applicable, or their successors.

All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies.

Copyright © 2008 CA. All rights reserved.

CA Product References

This document references the following CA products:

- CA Identity Manager

Contents

Chapter 1: Java CS	11
Knowledge Requirements	11
Java Connector Server	11
Endpoint Systems	12
Java CS Architecture	13
Java CS Framework	14
What Java CS Does	14
Java CS Services	14
Connectors	16
Connector Interfaces	16
Connector Implementation	16
LDAP Overview	17
LDAP Operations	17
LDAP Request Processing	17
Default Installation Directories	18
 Chapter 2: Java CS SDK	 19
SDK Overview	19
Example Connectors	20
SDK Packages	21
 Chapter 3: High Level Connector Design Considerations	 23
Scope	23
Containment Model	24
API	25
Special Character Considerations	27
Searching	28
IM--How Your Containment Model Impacts Your Search Strategy	28
Search Strategy Considerations	29
Porting an Existing C++ Connector to Java	29
Implementation Recommendations	30
Recommended Implementation Steps	31
 Chapter 4: Custom Connector Concepts	 35
Connector Configuration	35

Connection Management	36
Plug-In Classes	37
Validators	37
Converters	38
Exceptions	39
LDAP Exception Considerations	40
Custom Connector Code Upgrade Considerations	42

Chapter 5: SDK Sample Connector **43**

Sample Connector Overview	43
Possible Clients	44
Compiling the Sample Connectors	44
Sample Connector Upgrading	45

Chapter 6: Configuration Files **47**

How the Java CS Handles Configuration	47
Connector Jar Files	48
Converter and Validator Plug-Ins Registration	48

Chapter 7: The Object Model **51**

Metadata	51
Metadata Syntaxes	51
Data Model	52
Operation Bindings	53
Metadata Definition	54
Defining Metadata for an Existing C++ Connector Server Based Connector	54
How You Define Metadata for a New Connector	55
Create New Metadata	57
Metadata Descriptions	58
Association Metadata	77
Direct Associations	77
Indirect Associations	80
How Metadata Is Used	82
Association Related Code	83

Chapter 8: Implementing Connectors **85**

How to Implement a Connector	85
Implementation Guidelines	86
Connector Base Classes	86

Style Processors	88
Method Style Processor	88
Scripting Style Processor	91
Attribute Style Processor	91
Style Processor Methods	92
How Connectors Work	92
Connection Pooling Considerations	93
How To Test a Connection	94

Chapter 9: Writing Scripts 95

Implementing in Java or JavaScript Considerations	96
How you Pass Data to and from Scripts	98
LOOKUP and SEARCH Query Operations through Script Opbindings Considerations	100
Pure Scripted Connectors	101
Scripted Logic Update Considerations	102

Chapter 10: Creating Endpoint Objects 103

How you Create an Object	103
Add Operation Testing	105

Chapter 11: Removing Endpoint Objects 107

How you Delete an Object	107
Example: Implementing doDeleteAssocs	109
Example: Calling doDeleteAssocs() Methods Inside the doDelete and doModifyRn	109
Delete Operation Testing	110

Chapter 12: Querying Connector Objects 111

How You Search for an Object	111
How you Implement doLookup	112
How you Implement doSearch	112
How you Test the Search Operation	115

Chapter 13: Updating Endpoint Objects 117

Endpoint Object Update	117
Updating an Object	117
Update Operation Testing	119

Chapter 14: Implementing Associations	121
Associations	121
AssocAttributeProcessor Methods	121
Defining Associations	122
Reverse Associations	122
Handling DN Conversion	123
 Chapter 15: Renaming Endpoint Objects	 125
How You Rename the Object	125
Example: Implementing doModifyRnAssocs	126
Example: Calling doModifyRnAssocs() inside doModifyRn()	127
Rename Operation Testing	127
 Chapter 16: Moving Endpoint Objects	 129
How You Move the Object	130
Example: Implementing doMoveAssocs	131
Example: Calling doMoveAssocs() inside doMove()	131
Move Operation Testing	132
 Chapter 17: Custom Connector Deployment	 133
Java CS Installer Connector Deployment	133
Example Deployment	134
Deploying with Ant	135
 Appendix A: Testing with JMeter	 137
JMeter	137
Execute JMeter Test Cases Interactively	137
Test Case Contents	138
Run a JMeter Test Case	140
Editing Test Files	141
 Appendix B: Frequently Asked Questions	 143
Design Questions	143
Implementation Questions	146

Appendix C: Debugging Tips	151
Appendix D: Connector Review Checklist	153
Holistic Design Considerations	154
Java Development Standards Considerations	156
Metadata Use Considerations	157
Connector Coding Considerations	158
Component Test Considerations	160
Index	161

Chapter 1: Java CS

This section contains the following topics:

[Knowledge Requirements](#) (see page 11)

[Java Connector Server](#) (see page 11)

[Connectors](#) (see page 16)

[LDAP Overview](#) (see page 17)

[Default Installation Directories](#) (see page 18)

Knowledge Requirements

This guide is intended for developers who have the following technical background:

- Experience with the Java programming language and development environments
- Familiarity with the Java Naming and Directory Interface (JNDI) API
- Familiarity with the Lightweight Directory Access Protocol (LDAP) API
- Familiarity with XML and the Spring XML Open Source library that can convert XML documents into corresponding POJO (Plain Old Java Object) representations

Java Connector Server

The *Java Connector Server (Java CS)* is a server component which handles hosting, routing to, and management of Java connectors. The Java CS provides a Java alternative to the C++ Connector Server. It is architecturally and functionally similar to the C++ Connector Server, except that it has a Java API instead of a C++ API, which allows your connectors to be implemented in Java. In addition, the Java CS is data-driven rather than code-driven, which allows more functionality to be addressed by the container (or Java CS) instead of by connectors themselves.

The *Provisioning Server* handles provisioning of users, and then delegates to connectors (using the C++ Connector or Java CS) to manage endpoint accounts, groups, and so on.

Endpoint Systems

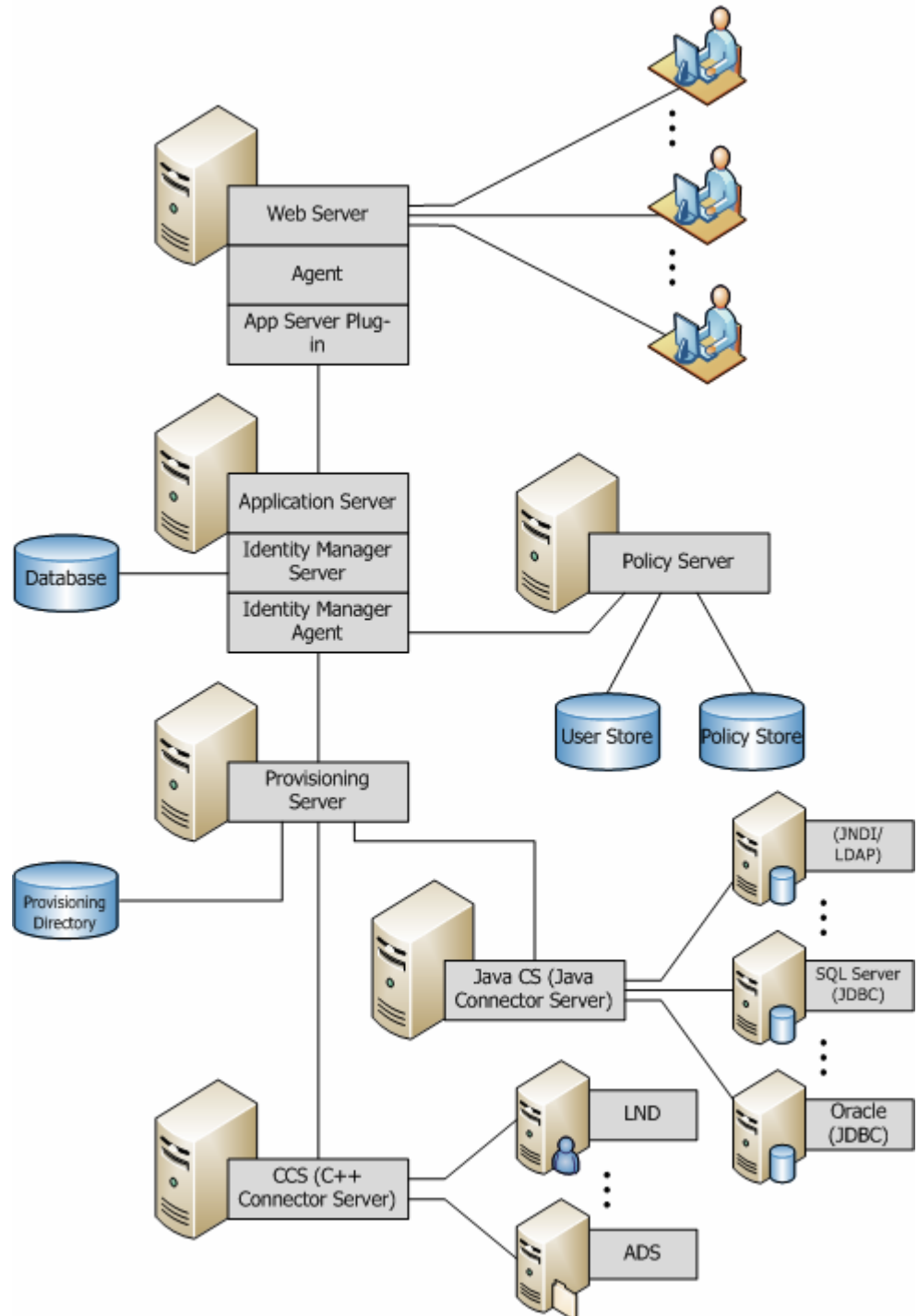
An *endpoint* is a specific target system or application, such as Active Directory or Microsoft Exchange, that is managed by the Provisioning Server.

A *connector* is the software that enables communication between a Provisioning Server and an endpoint system. For each endpoint that you want to manage, you must have a connector. Connectors are responsible for representing each of the object classes in your endpoint, and translating add, modify, delete, rename, and search LDAP operations on those objects into corresponding actions against the endpoint system.

A dynamic connector has XML metadata that is generated and maintained by end-users using Connector Xpress, whereas a static connector has metadata that is generated and maintained by developers rather than end-users.

Java CS Architecture

This following illustration shows the Java CS in a typical installation.



Java CS Framework

The Java CS makes extensive use of third party open source Java software libraries. It is built on top of Apache Directory Service (ApacheDS) and relies on the Spring Framework for XML configuration support.

You can find documentation at the following websites:

- <http://directory.apache.org>
- <http://www.springframework.org>

The Java CS framework supports:

- Run-time configurable data model and operation bindings using XML metadata
- Mapping of object and attribute names to / from LDAP to native equivalents
- Built-in connection resiliency
- Consistent data representation, validation, and conversion
- Extension and reuse of existing connector components
- (Optional) Reverse association handling, where associative relationships need only be expressed in one direction

What Java CS Does

Like the C++ Connector Server, the Java CS provides services to the connector implementation deployed to it while imposing minimal restrictions, in the same way that a J2EE application server provides for its hosted applications.

Java CS provides a framework and a standard set of service to ease connector development. Java CS:

- Uses XML files to simplify development and allow uniform configuration of all attributes.
- Enforces standardized representation of data types.
- Enables reuse among connector implementations, thus reducing the time, effort, and risk required to develop new connectors.

Java CS Services

The Java CS Framework provides the following high-level services:

- **Name Mapping**—The Java CS handles mapping between LDAP objectclass and attribute names with the matching names expected by the endpoint system (as specified by XML metadata).
- **Validations**—Through Java CS components called Validators that validate LDAP data before it is passed to, or optionally received from, connectors
- **Conversion**—Java CS components called Convertors that reformat LDAP data before it is passed to or received from connectors.
- **Resiliency**—The Java CS provides connection resiliency by attempting to reconnect to an endpoint system in the event of transient failures and connections, or both, becoming stale.

The Java CS Framework assists in the following developer tasks:

- **Associations**—The Java CS uses information in metadata about associations between objects, such as which accounts belong to which groups. Assistance from the framework takes the form of base classes which can be extended, and an optional Java proxy which can automatically handle implementation of reverse associations (for example, calculating `account.memberOf` from `group.member`) in many cases.
- **Activation and Deactivation**—The Java CS automatically handles registration and activation and deactivation of connectors when connection-sensitive values are modified.
- **Connection Management**—The Java CS has framework classes which assist in the configuration of implementation of pooling of connections to endpoint systems.
- **Exception Mapping**—The Java CS uses a Java proxy to support exception mapping from an endpoint system to a standard JNDI exception hierarchy, including LDAP error codes.
- **Coding Java proxy wrappers**—The Java CS framework includes base classes and utility methods to assist in coding and interposing Java Proxy objects wrapping any of the processing styles supported by a connector.

Connectors

A *connector* is the software that enables communication between a Provisioning Server and an endpoint system. The connector virtual directory maps LDAP operations such as add, modify, delete, search and rename to equivalent APIs calls and protocols specific to endpoint systems.

Note: Traditionally, Provisioning Manager connector implementations (previously known as Options) are written in C++ and are deployed to the C++ Connector Server. These C++ connectors perform the same job as Java connectors, and the C++ Connector Server performs a similar role to the Java CS although it provides less assistance to the connector developer. For more information, see the *Programming Guide for Provisioning*.

Connector Interfaces

Connectors are responsible for translating incoming LDAP operations to the equivalent operations on endpoint systems. Each connector that is hosted by Java CS must implement at least one (and possibly all, like the JDBC connector) of the following processing styles defined by the Java CS framework:

- **Attribute Style Processor**—Maps LDAP attributes to endpoint attributes, usually through Java CS framework support driven by metadata.
- **Method Style Processor**— Maps LDAP operations to PRE/OP/POST methods invoked on the endpoint system (for example, stored procedure support in JDBC connector).
- **Scripting Style Processor**—Maps LDAP operations and attribute into scripted output which is then submitted to the endpoint system for processing.

More Information:

[Style Processors](#) (see page 88)

Connector Implementation

Connectors are implemented using a combination of the following:

- CA-maintained implementation. The actual code is determined by the connector technology (for example, JDBC, JNDI).
- CA-maintained XML configuration (like registering plug-in validators and converters).
- End-user maintained XML metadata, managed with Connector Xpress, drives configured plug-ins.

LDAP Overview

Java CS sends and receives LDAP (Lightweight Directory Access Protocol) requests, and is based on the JNDI (Java Naming and Directory Interface) API. The LDAP protocol is designed to manipulate hierarchical, object-oriented data.

The primary unit of data for LDAP is an object. An object represents items such as an account, a directory, or an intermediate container. All these objects are organized in a Directory Information Tree (DIT). A DIT defines the structure or schema of your endpoint to Java CS.

LDAP Operations

The LDAP protocol provides the following simple set of operations that LDAP clients can perform on objects:

- **ADD**—Adds a new object to the DIT
- **MODIFY**—Modifies an object in the DIT
- **SEARCH**—Locates or enumerates one or more objects in the DIT
- **DELETE**—Deletes an object from the DIT
- **MODRDN**—Renames an object by modifying its Relative Distinguished Name (RDN)
- **COMPARE**—Compares an object in the DIT to a certain criteria

LDAP Request Processing

An LDAP request from a client interface always includes two items of information:

- The type of the request
- The DN of an object that the request targets

The Partition Loader Service detects incoming DN's, and creates and registers endpoint types and connectors when required, and as such is referred to as the *front end* of the Java CS architecture. It then passes LDAP requests back to the core ApacheDS to be serviced. As connectors are registered as ApacheDS partitions, they receive requests that fall into their sub-tree of the ApacheDS DIT.

Java CS can host any number of endpoint types (each of which is associated with a single metadata definition) and each of these can contain any number of connector instances (each of which is associated with a single set of connection details).

Default Installation Directories

The Java CS and Java CS SDK installation programs create directories for the Java CS executables and associated files. The default Windows and Solaris directories are listed in the following table. Your actual installation directories depend on your operating system and selections during the installation process.

The following notation is used throughout this guide to refer to various CA product installation directories:

Path Notation	Default Directory (Windows)	Default Directory (Solaris)
<i>jcs-sdk-home</i>	C:\Program Files\CA\Identity Manager\Connector Server	opt/CA/Identity Manager/Connector Server
<i>jcs-home</i>	jcs-sdk-home\build\dist	jcs-sdk-home\build\dist
Note: Under production JCS, this has the same structure as JCS home		
<i>sample-home</i>	C:\Program Files\CA\Identity Manager\Connector Server SDK\connectors\sdk	opt/CA/Identity Manager/Connector Server SDK/connectors/sdk
<i>im-home</i>	C:\Program Files\CA\Identity Manager	opt/CA/Identity Manager
<i>imps-home</i>	C:\Program Files\CA\Identity Manager\Provisioning Server	opt/CA/Identity Manager/Provisioning Server
<i>conxp-home</i>	C:\Program Files\CA\Identity Manager\Connector Xpress	opt/CA/Identity Manager/Connector Xpress

Chapter 2: Java CS SDK

This section contains the following topics:

[SDK Overview](#) (see page 19)

[Example Connectors](#) (see page 20)

[SDK Packages](#) (see page 21)

SDK Overview

The Java CS SDK installer includes sample code that covers the following areas of development:

- Writing custom connectors
- Handling connection management to endpoint systems
- Handling object associations
- Writing new converter and validator plug-ins, and arranging to trigger core plug-ins using metadata
- Implementing or enhancing connector logic using JavaScript (new in r12).

Note: Connectors have their own methods (such as `add()` and `modify()`) which they inherit from the `MetaConnector` base class, the primary engine of Java CS connectors. Connector developers should rarely (if ever) need to override these methods themselves. However, if this is done it is imperative that the `super.add()` method be invoked.

Example Connectors

The following example connectors are included in this SDK. Each connector has fully functioning build support and JMeter components tests validating their correct behavior:

- **SDK**—Contains examples of all core elements involved in implementation of a JCS hosted connector, such as:
 - Metadata and configuration files
 - Connector logic coding
 - Search filter conversion
 - Connection manager implementation
 - Search result streaming
- **SDKDYN**—Takes all its Java implementation from the SDK connector, and differs only in that it makes use of the DYN schema instead of the SDK schema (and consequently makes use of the DYN User Interface plug-in built in to the Provisioning Manager). This makes it possible to demonstrate some capabilities like automatic reverse associations which are not catered for in the SDK schema and Provisioning Server User Interface plug-in. This connector defines virtual containers in metadata (the new preferred approach) instead of in connector.xml. Look for the `<class name="eTDYNAccountContainer">` in *jcs-sdk-home/connectors/sdkdyn/conf/sdkdyn_metadata.xml*.
- **SDKSCRIPT**—This connector uses a copy of the metadata used by the SDKDYN connector, but all of its logic is implemented in JavaScript instead of Java. This is achieved using the script operation bindings that can be found in */conf/sdkscript_opbindings.xml*, which wrap a stub connector as configured via */conf/connector.xml*.

Note: This same technology also allows wrapping custom JavaScript processing around methods of existing connectors.

Note: As well as these connectors, some carefully chosen samples from the Java CS framework code are also included in the SDK (under *jcs-sdk-home/src/*), to aid the connector developer's understanding of core classes and best practices.

Note: For more information about the SDK's contents or building, see *index.html* under *jcs-sdk-home* or *README.txt*.

SDK Packages

The following are some key packages in using the Java CS SDK:

- `com.ca.datamodel`—Contains a library supporting loading and representation of data model metadata (XML documents conforming to the syntax spelled out in the `datamodel.xsd` schema) as Java objects.
- `com.ca.jcs`—Contains all classes that comprise the Java CS implementation. The Java CS supports connectors which accept LDAP input at the top level and convert it into the native language of the endpoint system with which they communicate.
- `com.ca.jcs.assoc`—A collection of classes and interfaces which support the representation and processing of associations between objects.
- `com.ca.jcs.cfg`—A collection of classes used to configure the Java CS and its contained components.
- `com.ca.jcs.enumeration`—A collection of classes used to handle returning the results of SEARCH operations back to client applications.
- `com.ca.jcs.filter`—Contains components for representing, analyzing, and converting LDAP filters passed to search operations.
- `com.ca.jcs.meta`—Contains components which are metadata-driven or assist in the condensing of information derived from metadata to allow efficient processing.
- `com.ca.jcs.processor`—Contains components for the processing of LDAP operations like add, modify, search, etc. Three styles are supported: attribute-style, method-style, and script-style.
- `com.ca.jcs.validator`—Contains all support for writing and configuring pluggable validators.
- `com.ca.jcs.converter`—Contains all support for writing and configuring pluggable converters.

Note: For more information on the packages in the SDK, see the JCS Javadoc.

Chapter 3: High Level Connector Design Considerations

When designing and implementing a new connector, there are a number of major questions that need to be considered. These considerations are discussed in this chapter, and are referred to throughout this document as a way of grouping logically related material together.

This section contains the following topics:

[Scope](#) (see page 23)

[Containment Model](#) (see page 24)

[API](#) (see page 25)

[Special Character Considerations](#) (see page 27)

[Searching](#) (see page 28)

[Porting an Existing C++ Connector to Java](#) (see page 29)

[Implementation Recommendations](#) (see page 30)

[Recommended Implementation Steps](#) (see page 31)

Scope

When designing a connector you need to consider the following:

- How many object classes your connector must support

We recommend that you use the built-in DYN parser table support bundled with the Identity Manager r12 Provisioning Server, as this greatly simplifies the connector development process (that is, you don't have to understand parser tables and you get a basic Provisioning Manager user interface for free). However for this release, if you require more than a single account, group and container object classes, then you will need to define your own custom parser tables.

Note: For more information see, the C++ *Provisioning Server SDK Guide*.

- An important distinction here is between objects that are fully managed versus objects that exist solely so that they can be referenced by managed objects. For example, there may be a need to display a list of all native roles so that the customer can choose which apply to a newly created account, even though the native roles are not themselves managed.
- What are the minimum attributes you need to implement on each object class to provide a useful *shallow* connector?

We recommend that you target this as an initial phase one delivery, and then build on this functioning base, rather than attempt to implement a *deep* connector immediately.

Containment Model

When designing a connector you need to consider the containment mode you will use, that is, is your Connector Flat, Hierarchical or Hybrid?

- You use flat connectors when the endpoint does not natively have a concept of containers for its objects, so all accounts and groups exist at the top-level of the connector. For this kind of connector, you should define *Virtual Containers* in the connector's metadata (or for legacy support, in the connector's `/conf/connector.xml` file) so that objects of each type appear to the user to be grouped into top-level logical containers. These containers are called *virtual* because they are not actually stored on the endpoint system, but are an artificial abstraction to group large collections of objects by object class for the benefit of the user. Most connectors are of this type, for example, the SDK sample, JDBC and AS400.
- You use hierarchical connectors when the endpoint system natively supports the notion of containers, as is the case for the Directory Servers with which the DYN JNDI connector communicates. These connectors do not need to define any Virtual Containers, but instead map to native containers to managed object classes in their metadata.
- Some connectors mix both the flat and hierarchical models, with each applying to a distinct set of object classes.
- Should your connector support the MODIFYRN (that is, rename) operation?
- If your connector is hierarchical, should it support the MOVE operation?

Note: For more information about the way these design choices impact they way you implement searches, see the topic Searching.

API

When designing a connector, you need to consider the following when deciding which technology or API the Connector will use connect to the endpoint:

- You need to be aware that the ApacheDS framework on top of which the Java CS is built allocates threads from a configured pool to all incoming requests, regardless of which connectors they target. Therefore your connector must be written in a threadsafe manner, as a single connector instance may be asked to process any number of requests concurrently. This makes use of a connection pool especially attractive, as connections to the endpoint system are often intended to be used by only one thread at a time. However, there may be other objects that you need to guard via synchronized methods or blocks in your connector code too.

Note: For more information, see the `configuration.maxThreads` setting in `conf/SAMPLE.server_jcs.properties`.

- Is the chosen technology API compatible with the notion of connection pooling?

If so, is it best to use the Java CS framework support for writing a pool or are there particular advantages to using native connection pooling support, if it is available?

- Connection pools have two main advantages:
 - Improving scalability / throughput when creating a new connections is expensive, as the pool allows existing connections to be reused rather than created and destroyed for each use.
 - Resource throttling, the pool imposes a limit so that the number of connections doesn't grow in an unbounded way, even under severe loading.
- If your connector deals with any multi-valued attributes, then the Java CS may be able to help deal with LDAP MODIFY requests, where each modification has a mode chosen from REPLACE (full list of new values is provided), ADD (list of values to be added to existing list is provided) or REMOVE (list of values to be removed from existing list is provided). A null list means the attribute should be removed). This means that for modifies of multi-valued attributes, you need to determine whether the chosen technology API best suits updating using `forceReplaceMode=REPLACE` metadata setting for each attribute (your connector will be provided with the complete list of new values, regardless of the mode chosen by the client application), or `forceReplaceMode=DELTA` (separate lists of items to be added or removed will be provided in all cases). There is also a value `forceReplaceMode=DELTA_WITH_REMOVES_FIRST` which behaves the same as DELTA except that the REMOVE delta is sent to your connector before the ADD delta (which suits some APIs better).
- Are you allowed to bundle the necessary libraries with your connector?

If not, you will need to document where the user can find them and where they need to be copied so the Java CS can find them, along with any additional configuration burden on the Java CS.

- Your connector can make use of its resource or directory to include any configuration files and utilities, or both, which are not directly part of the connector code. We recommend that you put migration scripts here if your connector is a port of an existing connector, and migration is required.
- Does the chosen technology or API use JNI and therefore require the Java CS to have runtime access to non-Java libraries through the Java Native Interface (JNI) API?

Note: Given the risks of using JNI, unless you have a very high level of confidence in the library concerned, it may be worth hosting the connector in a separate Java CS instance to ensure that if any JNI problems cause the host Java CS to crash, that no other connectors are impacted.

- Does the connector need to use multiple APIs to communicate with the endpoint? If so, this greatly complicates its design and implementation, particularly in the areas of connection pooling, resiliency and search result streaming.

Special Character Considerations

Careful attention needs to be paid to the testing the handling of special characters early in the design and prototyping stage, as problems relating to the handling of special characters in RDN values have been known to force which API is used. It is also important to ensure any characters that are special to the connector's chosen API are going to be quoted correctly on the way into and out of the native endpoint system, as well as considering quoting of characters special to LDAP itself as documented below.

The following is a list of special characters which need to be quoted with a preceding \ (backslash) character when they appear in Relative Distinguished Name (RDN) values (which appear at each level of a DN).

- A space or # (number sign or pound sign) character occurring at the beginning of the string
- A space character occurring at the end of the string
- One of the following characters:
 - , (comma)
 - + (plus sign)
 - " (quotation mark)
 - \ (backslash)
 - < > (angle brackets or chevrons)
 - ; (semi-colon)

Note: For more information, see <http://www.ietf.org/rfc/rfc2253.txt>

Multi-byte characters can be represented as \HH, where each H is a hex digit.

The following table lists the special characters that need to be quoted with a preceding \ (backslash) when they appear in LDAP search filters (used internally by the Java CS in reverse association handling):

Character	ASCII Value
*	0x2a
(	0x28
)	0x29
\	0x5c

Searching

Searching is one of the most difficult operations to implement for a connector. LDAP searches are powerful because they are formulated using a number of independent choices:

1. What is the base object for the search, that is, under what object should the search start?
2. What scope should apply to objects under the searches base object?
 - Object—only the base object
 - One-level—immediate children of the base object only
 - Subtree—any objects contained anywhere under the base object under any depth of containment
3. What filter condition should be applied to the objects falling within the chosen scope? Filter conditions further restrict the objects which are of interest for the search, based on the values of their attributes

Note: The *objectclass* attribute can be used to restrict only certain object classes

4. What attributes should be returned for each object falling within the specified scope and matching the specified filter?

How Your Containment Model Impacts Your Search Strategy

The way you implement your connectors search strategy and the type of filter conditions you apply is impacted by your decision to implement a flat or hierarchical containment model. For example:

- Flat connectors can ask the Java CS framework to split searches potentially encompassing multiple object classes into a separate sub-search over each object class, which greatly simplifies the implementation of the search logic.
- The Java CS framework handles Virtual Container logic, so that your connector does not need to be aware they exist. For example, a search specifying a Virtual Container as its base object will automatically have its search filter constrained to the single object class permitted for its children.
- The Java CS framework provides a notion of FilterVisitors that can assist in mapping LDAP filters to another syntax (for example to an SQL *where clause*) along with some concrete implementations that provide useful simplifications of the filters themselves (such as a Map of the attribute assertions contained).

Search Strategy Considerations

In implementing a search strategy you need to consider the following:

- How well do LDAP search filters map to filtering concepts in the native endpoint API?

Where possible, connectors should attempt to make use of filtering restrictions at the lowest possible level in the implementation, to prevent extra search results from being returned which then need to be filtered out at higher levels at greater performance cost. Connectors must at minimum support filtering based on object class, and against naming attribute values (both exact and wildcard matching). The Java CS can optionally (as configured via metadata) deal with performing post-filtering on search results, to account for cases where the connector can't natively respect all of the clauses in a search filter.

- Can your connector stream large sets of search results back to the client incrementally?

This is a very important behavioral characteristic for a connector, and while it is possible to achieve streaming with most APIs, a performance cost may be incurred. Where streaming can not be achieved (for example, with DYN or JDBC), you need to:

- Configure the Java CS needs with sufficient resources (for example, virtual memory) to support the largest sets of search results expected for the connector
- Place constraints on the amount of concurrent searching a Java CS instance can perform

Porting an Existing C++ Connector to Java

If you are porting an existing C++ connector to Java you need to consider the following:

- You need to decide if the Java connector is completely deprecating the C++ connector (in which case the Provisioning Manager plug-in does not need to concern itself with backward compatibility), or alternatively, whether its Provisioning Manager plug-in needs to detect whether it is communicating with a C++ or Java connector at the back-end by testing for the presence of the eTMetaData attribute on the connector's associated endpoint type.
- If there are any non-backwardly compatible changes (that is, changes rather than just additions) to an existing schema, parser table or format of any attribute values, then you will need to consider data migration and possible updates to the C++ connector if it is not being deprecated.

Implementation Recommendations

The following is a list of general recommendations to keep in mind while implementing connectors:

- Do not reference LDAP objectclass or attribute names in your code.
- Drive connector logic using metadata settings where possible, adding custom metadata properties if required.
- Use validator / converter plug-ins where possible, rather than repetitive code in your connector itself. Remember they can easily be registered against custom metadata properties through your connector's connector.xml file.
- Stream search results where large numbers of results are possible.
- Pay careful attention to result codes and error messages (which must be prefixed with `LdapExceptionPrefix`) for `LdapNamingExceptions` thrown by your connector.
- Be aware that the Provisioning Server now supports the `eTAgentOnly` operational attribute, which when included in the requested return attribute ids for a search, stipulates it should be directly routed to a connector. This means the Provisioning Manager GUI plug-in for hierarchical endpoint types can easily distinguish between all containers that exist on the endpoint system and possibly the smaller set that the customer has chosen to manage. This removes the need for work-arounds, such as the use of `containerList` attributes on container listing their immediate children. This attribute is also supported when the Provisioning Server is accessed through the JIAM (refer to the `com.ca.iam.eta.restrictTo` property which can be set to the values "agent" and "db") and SPML APIs.

Recommended Implementation Steps

The following are the recommended steps for implementing a new connector:

1. Choose a short indicative prefix for your connector and create a source directory for it under *jcs-sdk-home/connectors/* using the SDK connector's structure as a template.
 - a. Determine whether any useful base classes exist for you to derive your connector and attribute-style processor classes from (this may mean extending an existing connector implementation).
 - b. Create new derived classes as required, and ensure they are referenced properly in *connector.xml*.
 - c. Derive basic metadata for the object classes to be managed by your connector, initially paying particular attention to the top-level namespace and directory level properties and associated metadata settings. In particular the choice of the *implementationBundle=value*, which must match the value for `<property name="name">` your *ImplBundle* *JavaBean* in *connector.xml*.
2. Implement and test connection to the endpoint, which requires connector-level metadata settings are complete and correct. You should start a new JMeter file for your connector at this stage, and add test steps to it for each additional step. Such a test suite is invaluable, and very easy to write if you add to it incrementally during the implementation process.
3. Implement and test ADD operation (no associations yet).
4. Implement and test LOOKUP operation (no associations yet). This must be done very early (and very carefully) as the ApacheDS framework on which the JCS is built uses lookup operations internally to ensure sanity of the other operations. Hence, any bugs at this stage are a road-block for the whole connector's implementation.

When the array of requested attribute ids is null, all attributes (including expensive ones) should be returned. This differs from the default semantics for search operations.

5. Implement and test DELETE operation (no associations yet).
6. Implement and test MODIFY operation (no associations yet). If any multi-valued attributes are to be supported, then carefully consider whether use of the *forceModificationMode=REPLACE* or *forceModificationMode=DELTA* metadata settings on them might aid your implementation.
7. Implement and test SEARCH operation (no associations yet). Consider the following:
 - When the provided array of requested attribute ids is null, all attributes (excluding expensive ones) should be returned. This differs from the default semantics for lookup operations.

- Can your connector make use of the Java CS framework's *search one class at a time* support, to simplify implementation? If so, then *isBehaviourSearchSingleClass()* should return true.
 - NamingEnumeration (returned from search operations) base classes can be found in the `com.ca.jcs.enumeration` package. In particular *RawNamingEnumeration* takes care of handling size and time limits for its derived classes.
 - If the number of managed objects on the endpoint system could potentially be large, then a streaming solution is highly desirable. If your connector is making use of *search one class at a time* support, it may make sense to selectively implement streaming searches on only the object types which can have large numbers of instances.
 - It is possible to implement non-streaming search logic first and then later move to streaming logic as required? However, once you've written streaming logic such a phased approach is unlikely to be required.
 - After getting searches on managed object classes working, you should proceed to implement searches targeting unmanaged object classes as these are often required when using the Provisioning Manager, as described in step 9.
8. This is best point at which to start writing any custom validator and converter plug-ins required by any of your connector's attributes. As the set of attributes supported by your connector grows, you can add more as required.
9. Test the connector using the Provisioning Manager or Provisioning Server.

Important! You could also do this incrementally if desired, but we recommend you always use JMeter first. You could also delay this integration until the connector is fully implemented, and instead validate entirely using your JMeter test. However, as this integration point may produce problems we recommend you do this at this point in the implementation.

- Populate the record of the endpoint type in the Provisioning Server. For DYN based endpoint types, this can be achieved by placing a `_uninst/*pop.ldif` file in the connector's jar, and then running the JCS installer.

Note: For more information on other endpoint types and details about writing parser tables and POP scripts, see the Programming Guide for Provisioning.

- For all endpoint types (whether DYN based or not), the JCS installer can setup routing rules to your development JCS for your connector's endpoint type, if you stipulate that the endpoint type is to be registered. This can also be done manually using Connector Xpress at any stage.

10. Implement association handling logic in the same order as steps 2 - 7. If your connector makes use of direct associations, you need to consider the following:

- Can its attribute-style processor derive from `com.ca.jcs.assoc.DefaultAssocDirectAttributeOpProcessor` to do the heavy lifting? In any case, ensure that DNs stored in membership attributes are being converted to and from connector terminology properly by `com.ca.jcs.converter.connector.DNPropertyConverter`.

Note: This source code is bundled with the SDK.

- Should your connector return true from `isAutoDirectAssocRequired()` so that `com.ca.jcs.assoc.AssocAttributeOpProcessorProxy` will automatically take care of reverse association handling?
- If your connector returns true, then do any operations need to be excluded from this automatic handling via `getAutoDirectAssocExclusions()`?

11. Implement and test MODIFYRN operation (including handling of associations), if implemented by your connector.

12. Implement and test MOVE operations (including handling of associations), if implemented by your connector.

13. Configure and test resiliency (that is, investigate exceptions and such). This is a good place in the implementation to test your connector behaves properly when any of its connection related attributes are modified. Consider the following:

- Activation is treated specially, that is, the ADD request for a connector instance is not retried, if connectivity can not be established at this stage then the ADD will fail, and the customer will have to retry the ADD themselves manually. Only after the first successful ADD for the connector instance (after a JCS restart) will the resiliency support come into effect.
- Testing for resiliency involves trying to determine all the failure conditions your connector may encounter when communicating with the endpoint system, and categorizing them by their exception messages.
- Results are captured in `exceptionRetryMap` entries in `connector.xml`, tying each exception message to appropriate retry settings. Common groupings are non-retriable exceptions (the default), transient failures, stale connections and server too busy. Where configured, retrying is actually carried out by `com.ca.jcs.processor.RetryOpProcessorProxy`.
- Connectors can also force retrying in cases where they have access to important context available only in their code.

- Once resiliency has been correctly configured your connector should be able to reestablish connectivity with the endpoint system after transient failures. Retries should be configured to run for minutes rather than hours, at some point the connector needs to give up and defer the job of retrying to a higher level of the provisioning architecture.

Note: The corresponding chapters of this documentation sometimes group multiple steps around their associated operation (for example, all aspects of implementing each operation are discussed in the same chapter).

Chapter 4: Custom Connector Concepts

This section contains the following topics:

[Connector Configuration](#) (see page 35)

[Connection Management](#) (see page 36)

[Plug-In Classes](#) (see page 37)

[Exceptions](#) (see page 39)

[Custom Connector Code Upgrade Considerations](#) (see page 42)

Connector Configuration

The Spring Framework converts the `connector.xml` file, the major descriptor for the connector, into an initialized JavaBean instance of the `com.ca.jcs.ImplBundle` class. JavaBean settings contain all the information needed by the Java CS framework about a connector implementation.

In addition, connectors are encouraged to provide an override file installed into `jcs-home/conf/override/<connector-name>/SAMPLE.connector.xml` that contains the potentially more dynamic subset of the JavaBean properties that you may want to configure (for example, connection pool sizes).

The Java CS framework ignores these files unless they are renamed (or copied) so that the `SAMPLE.` prefix is removed. Future upgrades via the installer will not overwrite any custom modifications made by the user.

Note: The Spring framework does not merge settings from an active override file and the `connector.xml` file included in the connector for any JavaBean property. Settings in the `connector.xml` file are completely ignored for any JavaBeans mentioned in the override file.

Note: For more information, see `com.ca.jcs.ImplBundle`, the SDK sample connector, and <http://www.springframework.org> for usage examples.

Connection Management

Every Java CS connector uses the `getConnectionManager()` method to provide an instance of the `com.ca.jcs.ConnectionManager` class to its constituent processors and the Java CS framework.

We encourage connector developers to use these abstractions to interface with a connection pool to achieve the following benefits:

- Improve throughput by keeping heavy-weight connections to the endpoint available, rather than having to establish and close a connection every time one is needed.
- Ensure that the pool enforces a limit on the number of connections to the endpoint, limiting both memory usage inside the Java CS and the amount of work each connector can queue up with the endpoint.

The Java CS framework automatically deactivates and activates a connector when the value of any of attribute stored at the connector level of the DIT, and which has the `isConnection` Boolean metadata property set to true, changes. For example, when the credentials, host or port used to connect to the endpoint are modified, the Java CS will deactivate and reactivate the connector causing it to close the current connection manager and then reopen a new connection configured with the new settings.

Some connectors may need to make use of multiple forms of connectivity to the endpoint, in which case you should code the built-in connection manager as the primary form of connectivity and deal with the other forms in custom code. For example, with custom accessors such as `getConnectionManagerOtherAPI()`. Note however, that all connection managers can make use of the connection manager support classes allowing for minimal coding and customizing through the `connector.xml` file.

Plug-In Classes

Validators and converters are collectively referred to as plug-in classes because they can be written in isolation and then plugged into the Java CS using the `connector.xml` file (for single connector visibility) or the top-level `server_jcs.xml` file (for server-wide visibility). The Spring Framework processes the contents of these files, and establishes the links between an attribute and the plug-ins to be triggered for it based on its type as defined in metadata (for example, `intValue`) and its metadata property settings (for example, `maxLength=30`).

Note: For more information, see <http://www.springframework.org>

Validators are triggered before converters. Converters therefore know that their input is valid, and can focus on changing the form or syntax of attribute values as required.

Note: For more information, see *Converter and Validator Plug-Ins Registration* (see page 48).

Validators

A *validator* is a Java class that checks the validity of either individual values, attributes, or an entire object class. Validators are used by the Java CS to ensure that data values meet specific requirements before being passed to or from the endpoint system for processing. Validators also support localized error messages (using Sun internationalization support built into the JDK), which converters do not.

For example, if the legitimate values for a field are *X*, *Y*, and *Z*, then a validator could check modification requests to ensure that values are always in the accepted set. (The built-in `com.ca.jcs.validator.meta.EnumValueValidator` already does this where the permitted values are specified in an *enum* definition the data model metadata).

Where possible, using plug-in validators or converters increases opportunities for code reuse across connectors.

You can find several bundled validators to handle common scenarios in the packages `com.ca.jcs.validator.attr` and `com.ca.jcs.validator.meta`. These packages contain plug-ins with global scope and are therefore registered using the file `conf/server_jcs.xml`.

The SDK connector registers an example `com.ca.jcs.sdk.validator.NoCommaAttributeValidator`, including support for localizing its messages using the `messageResourceBundle` field in `conf/connector.xml`. This enables file access for the connector's files `validator.properties` (English) and `validator_fr.properties` (French), depending on the locale of your Java Virtual Machine.

Note: When testing, you can temporarily change `jcs.bat` and `jcs.sh` and try different locales using command lines like the following to run the Java CS:

```
java -Duser.language=fr ...
```

Note: For more information about Locale objects, see Understanding Locale in the Java Platform at <http://java.sun.com/>.

Typically, you run validators only on information received from the client. However, if you also want to enable validation on LDAP query responses, you can set the `validateFromConnector` attribute on your `ConnectorConfig` JavaBean in the `connector.xml` file.

Note: The package `com.ca.jcs.validator` contains validators that handle several common scenarios such as checking that attribute values conform to a maximum length or fall within a prescribed set of *enum* alternatives.

Converters

A *converter* is a Java class that converts data to and from specific formats. Converters are used by the Java CS infrastructure to convert data between the format of the Java CS LDAP type model and your endpoint's type model.

Converters are responsible for adjusting attribute values to and from the format and types expected by the endpoint system to which the connector communicates. For example, the Java CS represents a Date type in the XML format `2006-12-25` whereas a given endpoint may use the American date style `12/25/06`.

Configuring a converter allows a connector implementation to be concerned with the endpoint format, while an LDAP client has the format transparently transformed into its native format. Although format and type conversions can be performed in a connector implementation, a converter lets you separate this formatting code from the connector implementation and reuse it in other connectors.

Fields that require converters usually need matching validators. Validators are evaluated before converters, and support detailed localized error messages, which converters do not. This means that converters can assume they receive valid input data.

Converters must be able to map both from the values for LDAP attributes to their equivalent connector values (`convertToConnector()`), and the reverse direction (`convertFromConnector()`).

Several provided converters to handle common scenarios can be found in the packages `com.ca.jcs.converter.attr` and `com.ca.jcs.converter.meta`. These packages have global scope and therefore are registered in the file `conf/server_jcs.xml`.

Note: See *Converter and Validator Plug-Ins Registration* (see page 48) for instructions about how to register converter plug-ins.

Exceptions

You need to translate local exceptions from your endpoint system Java API into those provided by the LDAP protocol with the closest possible `ResultCodeEnum` values.

It is not always possible to find a perfect match between errors encountered in implementing a custom connector and appropriate classes extending `LdapException` and `ResultCodeEnum` values, but it is worth giving it (and the messages chosen for exceptions) careful thought. Keep in mind that these choices directly impact the content of the Provisioning Server's log messages, and are therefore very important when troubleshooting.

When raising exceptions in your connector code (possibly by translating native endpoint exceptions), make sure to use your connectors *IdapExceptionPrefix* as the start of all exception messages, to distinguish them from exceptions originating from the ApacheDS/Java CS framework levels of the software stack.

Note: For more information, see the SDK sample connector for examples.

LDAP Exception Considerations

Consider the following LDAP exceptions when writing a custom connector. Most exceptions are from the `org.apache.directory.shared.ldap.exception` package of ApacheDS, but a few exceptions are defined in the Java CS code. All the exceptions extend `javax.naming.NamingException`, but implement `org.apache.directory.shared.ldap.exception.LdapException` so a detailed LDAP code can be passed through. All exceptions can be found in the JCS Javadoc for either the ApacheDS (included in the SDK installer) or the Java CS, which you should use as your definitive reference.

Note: See the `org.apache.directory.shared.ldap.message.ResultCodeEnum` and `org.apache.directory.shared.ldap.exception.LdapException` in the Java CS Javadoc to see other implementing classes that are not listed.

LdapNameAlreadyBoundException

Thrown when an object with the same name as the one you are trying to create on the endpoint system already exists.

Result code: `ResultCodeEnum.ENTRY_ALREADY_EXISTS`.

LdapNameNotFoundException

Thrown when a DN is received which references an object found not to exist on the endpoint system.

Result code: `ResultCodeEnum.NOSUCHOBJECT`.

LdapServiceUnavailableException

Takes one of the return codes defined in `ResultCodeEnum.SERVICEUNAVAILABLE_CODES`. Call this when you are having communication exceptions with the endpoint system. This exception is extremely important for the retry code at higher layers of the system.

You can use an instance of this exception to flag transient failures to the Java CS framework, by setting its result code to `ResultCodeEnum.UNAVAILABLE`, in which case the resiliency support will cause the operation which caused the failure to be retried.

LdapConfigurationException

Thrown when an error in the configuration of a connector or the Java CS is encountered. Try to use more specific exceptions. You should avoid using this error code if possible. Details of the error should be provided in the error message.

Result code: `ResultCodeEnum.OTHER`

LdapNoPermissionException

Specifies that the requester does not have the right to carry out the requested operation.

Result code: ResultCodeEnum.INSUFFICIENTACCESSRIGHTS

LdapSizeLimitExceededException

Thrown when the number of results generated by a search exceeds the maximum number of results specified by either the client or the server, after results up to this limit have already been returned. If you make use of `sdk.com.ca.jcs.enumeration.RawNamingEnumeration` or one of its derived classes, handling size limits should not be an issue.

Result code: ResultCodeEnum.SIZELIMITEXCEEDED

LdapTimeLimitExceededException

See *LdapSizeLimitExceededException*.

Result code: ResultCodeEnum.TIMELIMITEXCEEDED

LdapInvalidAttributesException

Takes one of the six result codes defined in ResultCodeEnum.ATTRIBUTE_CODES.

LdapInvalidAttributeValueException

Thrown when an invalid value is encountered for an attribute, but in many cases correct use of validators and converters will remove the need to throw it.

Takes one of the following result codes:

- **Result code:** ResultCodeEnum.CONSTRAINTVIOLATION
- **Result code:** ResultCodeEnum.INVALIDATTRIBUTESYNTAX

LdapSchemaViolationException

Thrown when a request is received which attempts to bypass structural rules dictated by the endpoint system, such as creating an object under an inappropriate container.

Takes one of the following result codes:

- **Result code:** ResultCodeEnum.OBJECTCLASSVIOLATION
- **Result code:** ResultCodeEnum.NOTALLOWEDONRDN
- **Result code:** ResultCodeEnum.OBJECTCLASSMODSPROHIBITED.

LdapNamingException

Specifies a generic exception, to be avoided if at all possible.

LdapInvalidNameException

Result code: Not required

Custom Connector Code Upgrade Considerations

This release of the JCS is compatible with all previous releases of the JCS SDK Connector Code (8.1SP2 SDK CR10 and above). Therefore, unless a custom connector makes use of additional JCS API calls not present in the SDK connector code, you do not need to recompile it. If however, you have used extra API calls, note that some classes in the 8.1SP2 JCS API have been moved into sub-packages in this release, so you may need to make minor updates to import statements.

When recompiling your custom connector, you should deal with any deprecation warnings by moving to the alternative APIs as described in the relevant Javadoc.

Chapter 5: SDK Sample Connector

This section contains the following topics:

[Sample Connector Overview](#) (see page 43)

[Possible Clients](#) (see page 44)

[Compiling the Sample Connectors](#) (see page 44)

[Sample Connector Upgrading](#) (see page 45)

Sample Connector Overview

The SDK sample connector is an example of how to implement a custom connector. The SDK sample connector uses the `java.util.Properties` API for persisting account and group objects to a series of Java properties flat-files beneath a specified file system directory, local to the machine running the Java CS. Like all connectors, it implements the following mandatory LDAP operations:

- `doAdd()`
- `doModify()`
- `doDelete()`
- `doLookup()`
- `doSearch()`

It also implements the optional operation `doModifyRn()`, and supports direct associations between object types. It does not support `doMove()` and its variants.

It typifies a static connector (with developer-maintained metadata) in that it is a flat connector with no hierarchy except for the virtual containers *SDK Accounts* and *SDK Groups*. However, its use of flat files makes its connection manager a somewhat contrived example.

As recommended, it makes extensive use of metadata as well as sample validator and converter plug-ins. It also comes bundled with fully functional JMeter component tests.

The *sample-home*\sdk\build.xml ant file is included for compiling the SDK sample connector and associated code:

Possible Clients

To drive the SDK sample connector using the Provisioning Manager, you need to install the Admin SDK Option.

Note: For more information about installing the Admin SDK Option, see the *Java Connector Server Implementation Guide*.

Some JMeter tests are included for communicating directly with the Java CS, in which case the Provisioning Manager need not be running, or even installed. Although we do not provide support for the JMeter library other than some basic pointers, we strongly recommend that you write component tests concurrently with connector development (whether using JMeter or some other LDAP enabled testing framework). Informal manual testing may be useful in the early stages of connector development, but ultimately automated regression tests are required.

Unlike the C++ Connector Server, you can start the Java CS independently of the Provisioning Server. Connectors hosted by Java CS act much more like autonomous subtrees of a larger DIT, in particular supporting richer LDAP filter semantics. The components testing sequence is as follows:

- The SDK build.xml starts the Java CS when *ant jmeter.core* is invoked.
- The set of tests for each connector implementation runs in turn. Each test must create an endpoint type, a connector within the endpoint type, and then put the connector through its paces by submitting various LDAP requests on target objects of various kinds
- When all tests are completed, the Java CS spawned for the test run is automatically shutdown.

Note: You need to ensure you do not have a development Java CS (listening on port 20412) already running before executing them.

The JXplorer utility, included as part of the Provisioning Manager installation, can also be useful during connector development, particularly for running manual queries to check the state of a connector during the early stages of development.

Compiling the Sample Connectors

To compile the SDK Sample Connector, you need to run a top-level *ant* (which runs the default target *ant dist*) in *sample-home* from either your Java IDE or the command line using the bundled Ant build.xml. You can then run the Java CS hosting the connector by running build/dist/bin/jcs.bat (Windows) or jcs.sh (UNIX) or by using the bundled Eclipse or IDEA IDE runtime configurations.

The default development Java CS is configured to start on port 20412 (non-secure) or 20413 (secure TLS) to avoid conflicts if it is started on the same machine as a non-development Java CS service.

You can change this port by:

- Editing the ports specified in *jcs-sdk-home/conf/server_jcs.xml*. A better approach is to:
 - Rename */conf/override/SAMPLE.server_jcs.properties* to */conf/override/server_jcs.properties*
 - Edit */conf/override/server_jcs.properties* to add settings *configuration.ldapPort=<new ldap port>* and *configuration.ldapsPort=<new ldaps port>*
 - Restart JCS
- Changing the *jcs.test.port* setting in *jcs-sdk-home/build.xml*

Sample Connector Upgrading

The Java CS SDK uses Java deprecation to simplify upgrading connectors between different versions of Identity Manager. When you upgrade to a new version of the Java CS, pay attention to the JDK compiler deprecation warnings when compiling your custom connector.

Note: For more information on recommended alternatives to deprecated interfaces, see the Java CS Javadoc.

Chapter 6: Configuration Files

This section contains the following topics:

[How the Java CS Handles Configuration](#) (see page 47)

[Connector Jar Files](#) (see page 48)

[Converter and Validator Plug-Ins Registration](#) (see page 48)

How the Java CS Handles Configuration

Java CS handles configuration with XML support offered by the Spring Framework open source project in the following ways:

- XML content is used to describe JavaBeans and their property settings. This eliminates unnecessary code and offers a high level of configurability for virtually all of the Java CS and connectors.
- The Spring Framework reads global configuration for Java CS from the `server_jcs.xml` file. This is where you can change server-side configuration settings, and add new global-scoped validators and converters.
- The Spring Framework scans the *jcs-home/lib* directory on the local file system for connector jars with names like *jcs-connector-*.jar* and automatically loads their */conf/connector.xml* files.

Where present, the Spring framework will override any configuration settings read from these files with settings from *jcs-home/conf/override/<connector>/connector.xml*.

- The most dynamic component of a connector configuration is the metadata stored in its parent endpoint type, which dictates which validators and converters of those registered are actually activated for each objectclass or attribute. This can be changed using normal LDAP modify operations targeting the endpoint type.

Configuration of connector specific settings is achieved by JavaBeans deriving from `ConnectorConfig` that introduce any connector-specific properties that are required, instances of which are created based on the setting of the “defaultConnectorConfig” field in `connector.xml`. These instances are then passed in to the connector’s constructor by the Java CS framework.

Note: For more information, see www.springframework.org.

Connector Jar Files

A connector is packaged by the `-dist` target in its `build.xml` file as a single file named `jcs-connector-*.jar` which may depend on separate external third-party libraries `.jar` files

When `ant inst` is invoked from `jcs-sdk-home`, a connectors' `build.xml` file is asked to execute its `-inst` task which must create a single consolidated `jcs-connector-*.zip` file in `jcs-home/build/inst/` which includes every file it needs to be deployed to a production Java CS. This includes its `jcs-connector-*.jar` file, any third party libraries, and all sources of configurations settings. This `.zip` file is directly extracted under a `jcs-home` directory, so folder names must exactly match the standard names used by the Java CS (for example, `lib/`, `extlib/`, `conf/override/<connector>/` etc) on which it depends.

All connector jar files are collected in the Java CS lib directory: `jcs-home/lib`

Note: Currently you need to restart the Java CS for it to notice new connector `.jar` files added to its `lib/` directory.

The contents of a `jcs-connector-*.jar` file are:

- **/conf/connector.xml**—This file is converted into a `com.ca.jcs.ImplBundle` JavaBean using the Spring Framework XML support. This file is the major descriptor for the connector in the same way that the `/WEB-INF/web.xml` is for a `.war` file used to deploy a J2EE web application.
- **/conf/*_openldap.schema**—This optional file can be used by connectors that need to make their own connector-specific schemas known to the ApacheDS server hosting the Java CS. This file needs to be in the format dictated by the OpenLDAP standard.
- **/conf/*_metadata.xml**—This optional file can be used by connectors that have static metadata, or metadata maintained by developers rather than end-users. Such connectors are referred to as static connectors, as opposed to dynamic connectors where metadata is generated on the fly by end users using Connector Xpress.

Converter and Validator Plug-Ins Registration

You can use the `connector.xml` file to register validator and converter plug-ins, in addition to the system-wide library made available by the `server_jcs.xml` file. Both types of plug-ins are configured with the `com.ca.jcs.cfg.MetaPluginConfigSuite` JavaBean which ties attributes and classes to plug-ins using the following three collections:

- **typeToPluginMap**—A map of attributes values (such as *DATE*, *FLEXI_STR:DN*) to the AttributeValidator/AttributeConverter plug-ins you want to be triggered for them.

Note: For more information on value strings, see `com.ca.commons.datamodel.DataModelValue`.

- **propertyPluginConfigs**—This is a list of plug-ins driven by metadata settings rather than by the simple type. Each plug-in is invoked on every attribute which mentions any of the metadata settings configured in the plug-in's `metadataPropNames` trigger list, but is free to veto its application to each target attribute after performing further checks in its constructor (for example, by throwing a *PluginNotRequiredException* exception).
- **classPluginConfigs**—Also a list of plug-ins driven by metadata settings, but where the plug-ins work over all of the attribute settings for an object at once (rather than one attribute at a time).

Note: For examples matching your intended plug-in use case, see the *server_jcs.xml* file.

Note: Plug-ins can arrange to have configuration JavaBeans passed to their constructors, where they require context to configure their behavior. For an example of the required syntax matching your intended use case, see the *pluginConfig* setting for `com.ca.jcs.converter.meta.EncryptPropertyConverter` in the *server_jcs.xml* file.

Chapter 7: The Object Model

This section contains the following topics:

[Metadata](#) (see page 51)

[Metadata Definition](#) (see page 54)

[Association Metadata](#) (see page 77)

Metadata

Every connector has associated XML metadata that describes the classes of objects that the connector manages. The metadata includes the names of classes, their attribute names and attribute types. Additional descriptive information can be attached to a class or attribute, such as the maximum number of characters an attribute accepts.

Metadata Syntaxes

There are two different syntaxes for metadata documents:

- [Data Model](#) (see page 52)
- [Operation Bindings](#) (see page 53)

Data Model

The data model is of universal interest to all layers of the architecture and is stored as the value of the *eTMetadata* attribute on the *eTNamespace* object for an endpoint type. It defines object classes and their properties, and adheres to the following XML schema:

jcs-sdk-home/conf/xsd/datamodel.xsd

This schema defines both an object model (the classes and their properties) and the metadata properties which specify particulars of their processing. It includes a *SimpleValueGroup* group (which in turn references *PrimitiveValueGroup*) that defines the choices available for a property's value. Also note the *SimpleValue* complex type allows default values to be specified using the `default=` XML attribute, which will be used for a property when no explicit value is provided in an LDAP ADD request.

An example data model element might define an objectclass called *eTDYNAccount* which has a single-valued string property *eTDYNAccountName*.

Note: For an example of fully functional example documents see, *jcs-sdk-home/connectors/sdkdyn/conf/sdkdyn_metadata.xml* and *jcs-sdk-home/connectors/sdkdyn/conf/sdk_metadata.xml*.

Data model metadata settings describe extra data used by the software layers that need to interact with the data model. For example, the data model outlined above would require a Boolean *isNaming* metadata property with the value *true* specified for the *eTDYNAccountName* property, specifying that this property describes the naming attribute for its parent object class.

Operation Bindings

Operation Bindings metadata is stored as the value of the *eTOpBindingsMetaData* attribute on the *eTNamespace* object for an endpoint type, and can be used to register custom logic to wrap around the processing of LDAP operations (for example, in JDBC stored procedure support). As such, it is less frequently used than datamodel metadata. OpBindings documents adhere to the following schema:

jcs-home/conf/xsd/opbindings.xsd.

Each XML fragment can consist of multiple *OpBindingType* elements specifying:

- A guard element. The guard for an opbinding must match a particular operation on a particular object class for the Java CS framework to execute its payload. It therefore has elements specifying:
 - Which Operation matching an LDAP request (ADD / MODIFY etc) the binding targets.
 - Which target objectClass(es) (account / group) the binding targets

Note: When no objectClass elements are specified, this signifies this guard is intended to match all object classes defined in the datamodel metadata.
- An LDAP Boolean, which when true, denotes that the object class or classes are specified in LDAP terminology (for example, eT...Account, eT...Group) instead of in connector terminology (for example, account or group).
- A payload, which can be either:
 - A native method called on the target system (used for JDBC stored procedure support) using specified parameter definitions.
 - A complete script (or script function in a global script) to be called. Script and script functions derive from a *ScriptType* base type which contains:
 - A *scriptLanguage* string field specifying the language the script is written in (currently only javascript is supported).
 - An *executedDirectly* Boolean field which specifies whether the script does its work directly, or instead generates a string to be executed by the connector's script-style processor.
- A timing which can be:
 - **PRE**—Execute the opbinding before the target Operation. For example, it may be used for an ADD request to retrieve an additional attribute value from an external source of data and add it to the attribute values to be stored persistently.

- **OP**—Execute the opbinding instead of the target Operation on the connector.
- **POST**—Execute the opbindings after the target Operation on the connector, for example, to write an audit records.
- A strictCompletion Boolean which, when true, signifies that a failure encountered while executing the payload (either an exception, or a non-null textual error status return) should cause the framework to treat the whole LDAP operation as having failed. Where the endpoint system supports transactional behaviour (such as DYN JDBC) a failure when strictCompletion is true should be treated as a cause to rollback the transaction.
- An order integer used to specify the order opbindings should be executed in, when there is more than one guard condition matching a particular LDAP request and their order of execution is important.

Examples of opbindings metadata configuration can be found at *jcs-sdk-home/connectors/sdkscript/conf/sdkscript_opbindings.xml*.

Note: For more information on coding of script payloads, see [Writing Scripts](#) (see page 95).

Note: In this case, an entire connector is implemented using JavaScript but it is also possible to provide opbindings for only a few guard conditions, to customize some behavior of an existing connector. This is the recommended way to implement what were previously known as connector programs in Identity Manager 8.1 SP2. For example, the AS400 and OS400 connector which is part of Identity Manager r12, includes a sample opbinding.

Metadata Definition

The process of writing the metadata for a connector depends on the kind of connector being developed.

When using Connector Xpress, Connector Xpress creates the metadata document based on the selections you make. You should not need to edit metadata at all.

Defining Metadata for an Existing C++ Connector Server Based Connector

If you are reimplementing an existing connector that runs in the C++ Connector Server, you can create the basic skeleton of the metadata document by converting the parser table of the existing connector to metadata. This conversion process means that the LDAP attributes for the Java CS version of the connector will be identical to the original connector.

You convert the parser table to metadata by running the command-line tool `ptconvert`, which is part of the Connexor Xpress installation, located in *conxp-home/bin*.

Note: You must have the Admin SDK installed in order to gain access to the .PTY files needed to run `ptconvert`.

An example of a command to convert a PTY file to metadata is:

```
ptconvert -e "c:\program files\CA\eTrust Admin SDK\eTrust\Admin" myconnector.pty  
myconnector.xml
```

The conversion produces a metadata XML file that matches the object model defined in the parser table. Some additional metadata items that are needed by the Java CS must be manually added to the XML document.

In particular, metadata properties like `connectorMapTo` and those used to trigger validator and converter plug-ins will need to be added.

Note: For a complete list of all metadata properties, see `com.ca.jcs.meta.MetadataUtil` in the Java CS Javadoc.

You need to re-implement (in Java) any logic that is not automatically handled by the Java CS framework or the base connector you choose to derive from, based on the metadata.

Connector code (excluding metadata) which has been ported to Java CS has been found to be 70% to 90% smaller than the original C++ Connector Server C++ code.

How You Define Metadata for a New Connector

If you are creating a new connector for which there is no pre-existing specialized schema, you should consider the following two approaches:

- [Create a specialized data model based on the generic DYN schema and metadata](#) (see page 56)
- [Write your own specialized LDAP schema and register as you would for a pre-existing one.](#) (see page 57)

Regardless which options you chose, you need to write metadata from scratch that annotates your LDAP schema with all the information required by Java CS.

The most critical metadata setting is *connectorMapTo*, which specifies the mappings for objectclasses and attributes to connectors. For example, in a JDBC-based connector, the account objectclass (using the *class name*= 'XML syntax'), is mapped to a database table and its properties are mapped to columns within its parent table.

Some connectors may instead need to make use of the similar *connectorMapToAmbiguous* metadata property. For example, JNDI-based connectors might have an account which can have either of the naming attributes *cn=* or *uid=* on the endpoint.

Note: For an example of a hand-written metadata document, see the *sample-home/conf/sdk_metadata.xml* file in the SDK sample connector (ignoring metadata properties starting with *pt.*). For a list of supported metadata properties and values, see `com.ca.jcs.meta.MetadataUtil` in the Java CS Javadoc.

Use a Specialized Data Model

You can write connectors using a specialized data model based on the generic DYN schema (`eta_dyn_openldap.schema`) and metadata. For example, like the JDBC and JNDI DYN metadata output from Connector Xpress. This approach requires a metadata-aware client to interact with your connector, for example, the GUI client DYN Provisioning Manager plug-in. This means that, rather than displaying the LDAP attribute `id` `eTDYN-str-01` to the user, its mapped name `Description` is displayed instead.

We recommend this approach as long as your connector only needs to manage two object classes (`eTDYNAccount` and, optionally `eTDYNGroup`) and if its requirements are compatible with the set of attributes defined for them in the `r12` DYN schema. Where possible, making use of the DYN schema greatly simplifies development (as you don't need to write a custom parser table) and also simplifies enhancing released connectors as only metadata mappings need to change (that is, there is no impact on the Provisioning Server).

In this release, is it now possible to include POP scripts for DYN based connectors, as demonstrated for the SDKDYN connector by:

```
jcs-sdk-home/connectors/sdkdyn/conf/_uninst/ sdkdynpop.ldif.
```

POP scripts are required for DYN endpoint types because the custom mapping chosen are important when defining default account templates.

Note: Connector Xpress does the work of a POP script for endpoint types created within it.

It is also possible to write your own custom Provisioning Manager plug-in for the DYN schema, if the DYN plug-in does not meet your requirements.

Write Your Own Specialized LDAP Schema

You can write connectors by writing your own specialized LDAP Schema and registering the schema as you would for a pre-existing one.

Note: If you cannot use the DYN schema and want to use this approach, see the *Programming Guide for Provisioning* in the Identity Manager Bookshelf.

Metadata Used By the JIAM API

In addition to special metadata settings that are critical to the Java CS, there are also settings which are important to JIAM (Java Identity and Access Management) in that they establish a standardized facade over all connectors.

Note: For more information, see `com.ca.commons.datamodel.DataModelUtil` for relevant constants.

Note: JIAM is a Java front-end to the Provisioning Server.

Create New Metadata

To create new connector metadata

1. Add the following setting to the objectclass for the connector to distinguish it from all the other objectclasses listed in your metadata file:

```
<metadata name="connectorMapTo">
  <value>
    <strValue>connector</strValue>
  </value>
</metadata>
```

Note: Expressions like this are written in short-hand as `connectorMapTo=connector` for the remainder of this section.

2. Ensure that the connector's naming attribute has both a `connectorMapTo='mapping` (usually `=name`') and has `isNaming=true`'
3. Add `connectorMapTo=` values for all of the connector's properties which are connection-related or are otherwise singled out for special handling that derived connectors may want to make use of.

```
<metadata name="isConnection">
  <value><boolValue>true</boolValue></value>
</metadata>
```

This is because the Java CS framework informs your connector that it needs to deactivate and reactivate again after they are changed.

4. Look carefully at *connectorMapTo* values chosen for connection related attributes, and use the names defined as constants of form `com.ca.jcs.ConnectorConfig.CONN_*_ATTR` and in `ConnectorConfig`'s derived classes.

This greatly increases possibilities for reuse in the code that establishes connections, for all classes in an inheritance tree. As a result, the endpoint connector names can be independent of LDAP attribute names that tend to have different prefixes for each connector. For example, JNDI and a derived connector refer to the LDAP URL using the same connector name instead of two different LDAP attributes `eTDYN*` and `eT???` which both map to it.

5. For all other objectclasses, do the following:
 - a. Choose a *connectorMapTo*= value
 - b. Verify that its naming attribute has a *connectorMapTo*= value and has *isNaming=true*
 - c. Verify all properties which are supported by the connector implementation (presumably all) have *connectorMapTo* = values
 - d. Repeat from step a for all other objectclasses.
6. Check that the connector is functioning at a basic level.
7. Add and test association-related metadata properties like *refObjectType*= which describe relationships between objectclasses.
8. Review all objectclasses and their properties to ensure that all other relevant metadata properties documented in `com.ca.jcs.meta.MetadataUtil` have been correctly applied.

Metadata Descriptions

The following topics describe the metadata used in all layers of the architecture including Connector Xpress, JIAM, and the Java CS.

More information:

[Metadata for Endpoint Types](#) (see page 58)

[Metadata for Connectors](#) (see page 60)

[Metadata for Classes](#) (see page 61)

[Metadata for Classes or Attributes](#) (see page 62)

Metadata for Endpoint Types

abbreviation

Specifies the short abbreviated name of the endpoint type.

Type: String

accountclass

Specifies the LDAP object class of the account objects.

Type: String

endPointClass

Specifies the LDAP objectClass of the end-point objects.

Type: String

eta.legacyPolicyContainer

If true, indicates that the policy container is not under the endpoint type, but under the CommonObjects endpoint type.

Specifies the location of the policy container. If true, indicates that the policy container is not under the endpoint type, but under the CommonObjects endpoint type.

Type: Boolean

Default: False

groupclass

Specifies the LDAP objectClass of the group objects.

Type: String

implementationBundle

Should match exactly one of the getName() values of one of the com.ca.jcs.ImplBundle beans created from the conf/connector.xml files read from all the jcs-connector-*.jar files (each of which is a connector implementation) on the classpath. This value is of critical importance when the same LDAP schema is used for multiple connector implementations (for example, eTDYNNamespace which is used for both the JDBC and JNDI connector implementations).

Java Constant: MD_IMPL_BUNDLE

Type: String

Note: For more information, see com.ca.jcs.ImplBundle in the *Java CS Javadoc*.

isHierarchical

Specifies whether the connector supports arbitrary levels of nesting objects within containers, instead of being flat and having only fixed Accounts and Groups containers.

Java Constant: MD_IS_HIERARCHICAL

Type: Boolean

Default: False

policyContainerClass

Specifies the LDAP objectClass of the policy container.

Type: String

policyClass

Specifies the LDAP objectClass of the policy objects.

Type: String

quoteString

Specifies the quote string (for connector) to be replaced around objectClasses/attribute IDs.

Java Constant: MD_QUOTE_STRING

Type: String

Metadata for Connectors

connector

Connector (that is, Provisioning Directory) has connectorMapTo= this value, so it can be identified.

Java Constant: MAP_TO_CONNECTOR_VALUE

Type: String

isConnection

Distinguishes connection-related attributes on the connector, which when changed require the connector to be deactivated and then reactivated by the Java CS framework.

Java Constant: MD_IS_CONNECTION_ATTR

Type: Boolean

Default: False

isLogging

Distinguishes logging-related attributes on the Connector, to assist UI clients like the Provisioning Manager plug-in that need to be able to distinguish them.

Java Constant: MD_IS_LOGGING_ATTR

Type: Boolean

Default: False

Metadata for Classes

connectorMapToLax

Specifies that MD_CONN_MAP_TO mappings are *lax*. This Boolean can appear on any objectclass which has MD_CONN_MAP_TO specified.

This means any attributes on classes that aren't explicitly mapped will be treated as strings and keep their original names.

When connectorMapToLax is false, unmapped attributes passed in by clients for mapped object classes are discarded. When connectorMapToLax is true, unmapped attributes are passed through with their original names. Setting connectorMapToLax to true saves having to exhaustively map every property, which is particularly useful for the connector object class itself.

Java Constant: MD_CONN_MAP_TO_LAX

Type: Boolean

Default: False

isCreatable

Specifies whether the object class may be created. Otherwise the object class is not fully managed by the connector, but simply read from it.

Type: Boolean

Default: **True****isContainer**

Distinguishes containers from other object classes.

Java Constant: MD_IS_CONTAINER

Type: Boolean

Default: False

isDeletable

Specifies whether the object class may be deleted.

Type: Boolean

Default: True

isMovable

Specifies whether the object class may be moved.

Type: Boolean

Default: True

isRenameable

Specifies whether the object class may be renamed.

Type: Boolean

Default: True

isSingleNamingValue

Handle cases like multiple *cn=* values for JNDI-based connectors. This is acceptable as far as the Java CS is concerned, but may cause problems for the Provisioning Server during explore and correlations (setting this metadata property to true causes all values except the one mentioned in the object's RDN to be deleted).

Java Constant: MD_IS_SINGLE_NAMING_VALUE

Type: Boolean

Default: False

rdnAttribute

Specifies the name of the attribute that forms the RDN in LDAP.

Type: String

Metadata for Classes or Attributes

connectorMapTo

Specifies which name to map an object class (including the connector itself) or attribute to in connector-speak. For a dynamic connector, this specifies the name of the native system item to map the attribute to. For example, a JDBC based connector would probably map an object class to the name of a database table, and each property within it to the name of one of its columns.

Java Constant: MD_CONN_MAP_TO

Type: String

connectorMapToAlias

Specifies which name to map an objectclass (including the connector itself) or attribute to in connector-speak, in circumstances where MD_CONN_MAP_TO needs to be mapped to a complicated expression and therefore doesn't act as a useful reference name in the connector's code.

For more information, see LDA which has ambiguous mappings for many objectclasses/attributes or ORA and SQL JDBC-based connectors which set:

```
<code>MD_CONN_MAP_TO=[some complicated SQL SELECT statement]</code>
```

and

```
<code>MD_CONN_MAP_ALIAS=account</code>
```

Java Constant: MD_CONN_MAP_ALIAS

Type: String

connectorMapToAmbiguous

Specifies the rare cases where a single LDAP attribute ambiguously maps to more the one connector objectclass / attribute. The order of the choices is important as the first one will be defaulted to when no explicit choice is made and a new object is created. For instance the LDA connector has a number of such ambiguities:

- `eTLDAAccount` is mapped to any one of the objectclasses `person`, `residentialPerson`, `organizationalPerson`, `inetOrgPerson`.
- `eTLDAContainer` is mapped to any one of the objectclasses, `organizationalUnit`, or `organization`, which each have different naming attributes. Consequently, `eTLDAContainerName` makes use of a special syntax which ties its ambiguous choice to the type of the parent object:

```
<metadata name="connectorMapToAmbiguous">
  <value>
    <sequenceValue>
      <val><strValue>organizationalUnit:ou</strValue></val>
      <val><strValue>organization:o</strValue></val>
    </sequenceValue>
  </value>
</metadata>
```

Java Constant: MD_CONN_MAP_TO_AMBIGUOUS

Type: Sequence

connectorMapToAmbiguousChoiceAttr

Specifies that an attribute with ambiguous mappings can optionally name another LDAP attribute which will be used to pass in an explicit choice when new instances of the objectclass are add()ed. For instance, in the LDA connector, `eTLDAAccountName` can be either `cn=` or `uid=`, therefore its metadata specifies:

```
<metadata name="connectorMapToAmbiguousChoiceAttr">
  <value>
    <strValue>eTLDANamingAttribute</strValue>
  </value>
</metadata>
```

This means that an account creation would specify `eTLDANamingAttribute=uid` to explicitly state the desired choice of connector-speak naming attribute.

Java Constant: MD_CONN_MAP_TO_AMBIGUOUS_CHOICE_ATTR

Type: String

connectorMapToScriptLang

Where MD_CONN_MAP_TO is a script, this field specifies its language (for example, Javascript).

Specifies the MD_CONN_MAP_TO language (for example, Javascript), where MD_CONN_MAP_TO is a script.

Java Constant: MD_CONN_MAP_TO_LANG

Type: String

Note: Not currently supported.

displayName

Displays a friendly name for entity.

Java Constant: MD_DISPLAY_NAME

Type: String

description

Describes the class or attribute, for programmer use.

Language: Always in English

Type: String

Metadata for Attributes

assocRefObjectClass

Defines a relationship from an obj type to a referenced obj type, where direct association through an attribute and indirect association through a table external to both object types are supported.

For direct association, an attribute in the *referencing* objectclass, (for example, Account.groups), defines its association with another *referenced* objectclass, (for example, Group), by specifying the multi-valued attribute in the referenced objectclass that is used to store the names of contained items (for example, Group.accounts). A concrete connector then needs to perform the logical reverse lookup required to populate the virtual associative attribute. In this example, the virtual Account.groups attribute is populated by looking for Groups which contain an account's name in its Group.accounts attribute. Two metadata settings are required:

- MD_ASSOC_REF_OBJECT_CLASS (=Group in the example above)
- MD_ASSOC_REVERSE_ATTR (=accounts in the example above)

For the indirect case, you need to specify the following:

- The referenced objectclass must be specified through MD_ASSOC_REF_OBJECT_CLASS), as for the direct case.
- An association table (specified by MD_ASSOC_TABLE) with two columns:
 - The column named in MD_ASSOC_TABLE_OBJ_COLUMN references the key for the containing objectclass
 - The column named by MD_ASSOC_TABLE_REF_COLUMN) references the key for the referenced objectclass.
- An optional MD_ASSOC_OBJ_KEY_ATTR can be used to flag that an alternative key for the containing objectclass is to be stored in the MD_ASSOC_TABLE_OBJ_COLUMN column (if it is not specified then the containing object's naming attribute is assumed).
- An optional MD_ASSOC_REF_KEY_ATTR can be used to flag that an alternative key for the referenced objectclass is to be stored in the MD_ASSOC_TABLE_REF_COLUMN column (if it is not specified then the containing object's naming attribute is assumed).

Except for the last two, all of these metadata settings are mandatory to specify an indirect association.

Java Constant: MD_ASSOC_REF_OBJECT_CLASS

Type: String

beanPropertyName

Specifies the name of the Java Bean property corresponding to this attribute.

Type: String

boolValues

Specifies, in order, the alternate symbolic values that you can use to represent false (0) and true (1) for target <boolVal> properties stored on the endpoint.

For example:

```
<metadata name="boolValues">
  <value>
    <sequenceValue>
      <!-- "false" value -->
      <val><strValue>XX</strValue></val>
      <!-- "true" value -->
      <val><strValue>YY</strValue></val>
    </sequenceValue>
  </value>
</metadata>
```

Java Constant: MD_BOOL_VALUES

Type: Sequence

connectorConverter

Allows explicit converter plug-in configuration to be chosen for a property by reference to its name, where contextual information is not sufficient to allow the choice to be made automatically (for example, if there are multiple flattening converters for a multi-valued field which needs to be flattened).

Java Constant: MD_CONN_CONV_NAME

Type: String

connectorMapToStrict

Causes the framework to augment objectclass filters to include an explicit check for the existence of a particular connector-speak naming attribute. This is used in cases where the naming attribute is potentially ambiguous but you have chosen to map only one choice.

For instance, in the JNDI case map account to *person* and map account naming attribute to *uid*. If this metadata setting is *true* on the naming attribute, then the LDAP filter:

```
<code>(objectclass=eTDYNAccount)</code>
```

will become

```
<code>(&(objectclass=person)(uid=*))</code>
```

Without this metadata setting the filter will be

```
<code>(objectclass=person)</code>
```

, and the framework will deduce the eTDYNAccountNames of pre-existing native accounts that only define

```
<code>cn=</code>
```

 and have no value for

```
<code>uid=</code>
```

.

Note: Naming property only.

Java Constant: MD_CONN_MAP_TO_STRICT

Type: Boolean

Default: False

converterRegex

Specifies regular expression (of style specified by MD_CONVERT_REGEX_STYLE) to be used to convert values stored in the target property. Where the same regular expression is used in a number of places, a better approach would be to define a

```
<code>FLEXI_STR:myType</code>
```

 type and tie this type to a `com.ca.jcs.converter.attr.RegexAttributeConverter` through your connector's `conf/connector.xml` file.

Java Constant: MD_CONVERT_REGEX

Type: String

Note: For more information, see `com.ca.jcs.converter.attr.RegexAttributeConverter` in the *Java CS Javadoc*.

converterRegexStyle

Specifies style of regular expression to be used to validate the target property.

Java Constant: MD_CONVERT_REGEX_STYLE

Type: String

Default: perl5

Note: For more information, see MD_CONVERT_REGEX and also com.ca.jcs.cfg.RegexPluginConfig#PERL5 in the *Java CS Javadoc* for possible styles.

connectorValidator

Allows explicit validator plug-in configuration to be chosen for a property by reference to its name, where contextual information is not sufficient to allow the choice to be made automatically (for example, if there are multiple flattening converters for a multi-valued field which needs to be flattened).

Java: Constant: MD_CONN_VALID_NAME

Type: String

defaultPolicyValue

Contains the value to use for this attribute in the default policy. Only present on account attributes.

Type: String

(Mandatory) DNLDAPObjectClass

Specifies name of objectclass referenced by the distinguished name value(s) for the target property.

Note: Any property with FLEXI_STR:DN setValue

Java Constant: MD_DN_LDAP_OBJECTCLASS

Type: Boolean

DNNameOnly

Specifies that the connector, (which must have `isHierarchical = false`), wants the DNs converted to a single name for storage.

For example, `<code>eTDYNGroupName=grp1</code>` will be converted to `<code>grp1</code>`, before passing the value to the connector. In addition, the reverse transformation from name to DN will also occur when the query results are passed back by the connector.

Note: Applies to Any property with *FLEXI_STR:DN* setValue

Java Constant: MD_DN_NAME_ONLY

Type: Boolean

Default: False

DNTestExists

Specifies that each contained DN should be tested for existence by doing a `doLookup()` on each DN requesting an empty array of attributes.

Note: Applies to any property with *FLEXI_STR:DN* setValue.

Java Constant: MD_DN_TEST_EXISTS

Type: Boolean

Default: False

Note: For more information, see `com.ca.jcs.LdapUtil#TEST_EXISTENCE_ATTRS` in the *Java CS Javadoc*.

encryptAlgorithm

Specifies algorithm to be used for encryption. For more information, see. `MD_IS_OBFUSCATED`.

Note: Any encrypted property.

Java Constant: MD_ENCRYPT_ALG

Type: String

flattenStyle

Specifies that the values for the multi-valued target property should be flattened into a single string literal using the named *style* (for example, XML, CSV, SQL).

Note: Multi-valued properties only.

Java Constant: MD_FLATTEN_STYLE

Type: String

Note: For more information, refer to classes deriving from `com.ca.jcs.converter.meta.MultiValuePropertyFlattener` in the *Java CS Javadoc*.

flattenStyle

Type: String

Specifies the way Java CS flattens a multi-valued attribute in the Provisioning Server to a single valued endpoint attribute. This metadata item can be set to either XML, CSV, or SQL. No flattening will be done if none is provided.

forceModificationMode

Specifies that there is only one mode of modification item for the target property, regardless of the modification operation originally passed in by the client.

This can be set to REPLACE or DELTA.

This is useful when the API used by the connector to interact with the managed system mandates a particular style of access for the property.

DELTA

Specifies that a REPLACE_ATTRIBUTE modification item is replaced by one or both equivalent ADD_ATTRIBUTE and REMOVE_ATTRIBUTE deltas.

A REPLACE_ATTRIBUTE with no values (that is, completely remove the attribute) is replaced by a REPLACE_ATTRIBUTE with all current values mentioned explicitly.

REPLACE

Specifies that an ADD_ATTRIBUTE or REMOVE_ATTRIBUTE modification item is replaced by equivalent item with operation REPLACE_ATTRIBUTE.

Java Constant: MD_FORCE_MOD_MODE

Type: String

isConnectorFilterable

Indicates that the connector is unable to filter this attribute. Filters supplied to the connector will have references to this attribute removed. The results returned for searches filtering on this attribute will have the complete filter reevaluated against each of them by the Java CS framework to determine if they really should be returned, at an additional runtime cost.

Java Constant: MD_IS_CONNECTOR_FILTERABLE

Type: Boolean

Default: False

isEncrypted

Triggers the Provisioning Manager plug-in to display the attribute in a "password" type control so that characters are echoed as *. The user must retype their choice in a second confirmation test field.

Java Constant: MD_IS_ENCRYPTED

Type: Boolean

Default: False

isExpensive

Designates that the target property to which it is attached is expensive to retrieve and is therefore only returned in SEARCH requests when explicitly listed amongst the attributes to be returned (and therefore not by the default null list). For example, a photograph stored on a user / account object.

Java Constant: MD_IS_EXPENSIVE

Type: Boolean

Default: False

isInternal

Specifies whether this attribute will be made available through public APIs such as JIAM. If false, attribute will be available.

Type: Boolean

Default: False

isJavaObject

Triggers `com.ca.jcs.converter.meta.JavaObjectPropertyConverter` converter from LDAP string values to the Java Object matching DataModel value defined in the metadata.

For example, the string `true` stored in a `boolValue` field will be replaced with `<code>Boolean.TRUE</code>`.

Java Constant: MD_IS_JAVA_OBJECT

Type: Boolean

Default: False

Note: For more information, see `DataModelValue#getValue(int, String)` and `com.ca.jcs.converter.meta.JavaObjectPropertyConverter` in the *Java CS Javadoc*.

isMultiValued

Specifies whether the attribute can hold multiple values.

Java Constant: MD_IS_MULTI_VALUED

Type: Boolean

Default: False

isNaming

Specifies whether the attribute is the LDAP naming attribute.

Java Constant: MD_IS_NAMING

Type: Boolean

Default: False

isObfuscated

Triggers encryption converter using the algorithm specified using MD_ENCRYPT_ALG. This means that values are persisted in obfuscated form on the managed system.

Note: Obfuscating fields means that they can only be read by the connector and can no longer be accessed by direct access to the managed system.

Java Constant: MD_IS_OBFUSCATED

Type: Boolean

Default: False

isReadOnly

Specifies whether the attribute is read only.

Java Constant: MD_IS_READ_ONLY

Type: Boolean

Default: False

isRequired

Designates that the target property to which it is attached is required, meaning that a value must be provided for it when objects are created and that it must have a least one value at all times.

Note: Using `<value default=true>...</value>` implies that `isRequired=true`, as the Java CS will insert the provided default value unless the target property has a value explicitly provided for it in an ADD request.

Java Constant: MD_IS_REQUIRED

Type: Boolean

Default: False

isReadOnly

Specifies whether the attribute is read only.

Java Constant: MD_IS_READ_ONLY

Type: Boolean

Default: False

isStoreNumeric

Specifies that target properties which have types that can either be stored as strings or numeric types, (for example, <boolVal> or <enumVal>), are stored on the endpoint in numeric form.

See also MD_IS_STORE_SYMBOLIC, which specifies that it is stored in string form.

Java Constant: MD_IS_STORE_NUMERIC

Type: Boolean

Default: False

isStoreSymbolic

Specifies that target properties which have types that can either be stored as strings or numeric types, (for example, <boolVal> or <enumVal>), are stored on the endpoint in string (symbolic form).

See also MD_IS_STORE_Numeric, which specifies that it is stored in numeric form.

Java Constant: MD_IS_STORE_SYMBOLIC

Type: Boolean

Default: True

isStringArray

Triggers com.ca.jcs.converter.meta.StringArrayPropertyConverter converter from multi-valued LDAP attribute to String[].

Note: Multi-valued properties only

Java Constant: MD_IS_STRING_ARRAY

Type: Boolean

Note: For more information, see com.ca.jcs.converter.meta.StringArrayPropertyConverter in the *Java CS Javadoc*.

isTrimRequired

Specifies that that target property needs a whitespace trim operation.

Java Constant: MD_IS_TRIM_REQUIRED

Type: Boolean

Default: False

isVirtual

Flags the target property as *computed* rather than *persisted*.

Note: This is currently used for associations but may see wider use in the future.

Java Constant: MD_IS_VIRTUAL

Type: Boolean

Default: False

isWriteOnly

Designates that the target property to which it is attached is write-only and therefore can be modified but not read. This is expected to be of use for security-sensitive fields like passwords.

Java Constant: MD_IS_WRITE_ONLY

Type: Boolean

Default: False

minLength

Specifies minimum length for value(s) of target property.

Java Constant: MD_MIN_LENGTH

Type: Integer

maxLength

Specifies maximum length for value(s) of target property.

Java Constant: MD_MAX_LENGTH

Type: Integer

minValue

Specifies minimum bound on value(s) of target property.

Java Constant: MD_MIN_VALUE

Type: Number (Integer, long, float or double)

maxValue

Specifies maximum bound on value(s) of target property.

Java Constant: MD_MAX_VALUE

Type: Number (integer, long, float or double)

regexDescription

Describes the regular expression used to validate, or validate and convert the target property, which appears in validation failure messages.

Java Constant: MD_REGEX_DESCR

Type: String

validatingRegex

Specifies regular expression (of style specified by MD_VALID_REGEX_STYLE) to be used to validate values stored in the target property. Where the same regular expression is used in a number of places, a better approach would be to define a `FLEXI_STR:myType` type and tie this type to a `com.ca.jcs.validator.attr.RegexAttributeValidator` using your connector's `conf/connector.xml` file.

Java Constant: MD_VALID_REGEX

Type: String

validatingRegexStyle

Specifies style of regular expression used to validate the target property. For more information, see MD_VALID_REGEX, and `com.ca.jcs.cfg.RegexPluginConfig#PERL5` for example, for possible styles.

Java Constant: MD_VALID_REGEX_STYLE

Type: String

Value: java

writableBy

Restricts where a property's value, or all of a class' property, can be set.

Note: Setting this value to ADD is similar to MD_IS_READ_ONLY, except that the latter throws an error whereas this setting simply throws away attempts to modify the property using the prohibited ADD or MODIFY. The default is null, where both ADD and MODIFY can set property's value.

Note: Applies to any writable property.

Java Constant: MD_WRITABLE_BY

Type: String

Value: null

Metadata for Associations

assocReverseAttr

See MD_ASSOC_REF_OBJECT_CLASS

Note: Direct association property only.

Type: String

assocRefKeyAttr

See MD_ASSOC_REF_OBJECT_CLASS

Java Constant: MD_ASSOC_REF_KEY_ATTR

Type: String

assocTable

See MD_ASSOC_REF_OBJECT_CLASS

Note: Indirect association property only.

Java Constant: MD_ASSOC_TABLE

Type: String

assocTableObjColumn

See MD_ASSOC_REF_OBJECT_CLASS

Note: Indirect association property only.

Java Constant: MD_ASSOC_TABLE_OBJ_COLUMN

Type: String

assocObjKeyAttr

See MD_ASSOC_REF_OBJECT_CLASS

Note: Indirect association property only.

Java Constant: MD_ASSOC_OBJ_KEY_ATTR

Type: String

assocTableRefColumn

See MD_ASSOC_REF_OBJECT_CLASS

Note: Indirect association property only.

Java Constant: MD_ASSOC_TABLE_REF_COLUMN

Type: String

Association Metadata

In addition to describing objectclasses in metadata, you need to describe the association relationships between objects (for example, the list of groups to which a user belongs).

There are two types of associations:

- Direct (used by the JNDI connector)
- Indirect (used by the JDBC connector)

Note: For more information, see the class `sdk.com.ca.jcs.meta.MetadataUtil` in the JCS Javadoc.

Direct Associations

In Direct Associations, references are persisted directly into a multi-valued attribute on the endpoint, for example, in LDAP a group's member attribute directly stores reference to the accounts it contains and is represented by metadata of this form:

```
<property name="eTDYNMember">
  <doc>LDAP member [DN]</doc>
  <value>
    <setValue>
      <baseType>
        <FlexiStrValue type="DN" />
      </baseType>
    </setValue>
  </value>
  <metadata name="beanPropertyName">
    <value>
      <strValue>member</strValue>
    </value>
  </metadata>
  <metadata name="isMultiValued">
    <value>
      <boolValue>true</boolValue>
    </value>
  </metadata>
</property>
```

```
<metadata name="connectorMapToAmbiguous">
  <value>
    <sequenceValue>
      <baseType>
        <strValue />
      </baseType>
      <val>
        <strValue>
          groupOfUniqueNames:uniqueMember
        </strValue>
      </val>
      <val>
        <strValue>groupOfNames:member</strValue>
      </val>
    </sequenceValue>
  </value>
</metadata>
<metadata name="refObjectType">
  <value>
    <strValue>eDYNAccount</strValue>
  </value>
</metadata>
<metadata name="isDNAbsolute">
  <value>
    <boolValue>false</boolValue>
  </value>
</metadata>
<metadata name="DNLDapObjectClass">
  <value>
    <strValue>eTDYNAccount</strValue>
  </value>
</metadata>
```

Note: For more information about the various metadata settings, see the Java CS Javadoc for the constants in `MetaDataUtil.java` matching each metadata property's name.

In the Direct case it is often useful to also define a virtual attribute (so called because it is calculated on the fly at possibly considerable runtime cost) which can pass back information about the association in the reverse direction. For instance the *eTDYNMemberOf* for an account returns the list of groups to which each account belongs calculated entirely from the `group.member` attribute discussed above. Its metadata would be defined as follows:

```
<property name="eTDYNMemberOf">
  <doc>LDAP memberOf [DN]*</doc>
  <value>
    <setValue>
      <baseType>
        <flexiStrValue type="DN" />
      </baseType>
    </setValue>
  </value>
  <metadata name="beanPropertyName">
    <value>
      <strValue>groupNames</strValue>
    </value>
  </metadata>
  <metadata name="isMultivalued">
    <value>
      <boolValue>true</boolValue>
    </value>
  </metadata>
  <metadata name="connectorMapTo">
    <value>
      <strValue>memberOf</strValue>
    </value>
  </metadata>
  <metadata name="virtual">
    <value>
      <boolValue>true</boolValue>
    </value>
  </metadata>
  <metadata name="forceModificationMode">
    <value>
      <strValue>DELTA</strValue>
    </value>
  </metadata>
  <metadata name="refObjectType">
    <value>
      <strValue>eTDYNGroup</strValue>
    </value>
  </metadata>
  <metadata name="assocAttr">
    <value>
      <strValue>eTDYNMember</strValue>
    </value>
  </metadata>
  <metadata name="isDNAbsolute">
    <value>
      <boolValue>false</boolValue>
    </value>
  </metadata>
</property>
```

```
<metadata name="DNLDAPObjectClass">
  <value>
    <strValue>eTDYNGroup</strValue>
  </value>
</metadata>
</property>
```

In cases where the metadata values are similar to the member case, except that the *forceModificationMode=DELTA* setting, modifications are always expressed as a set of additions and deletions, easing the process of updating the various group.members list internally.

Indirect Associations

In Indirect Associations, links between objects are not stored on either object but rather in a table external to both which stores the keys to both objects. This is the scheme expected by the JDBC connector as it matches the way relational database models associations, and is expressed in the following metadata (where the table and column names would vary according to each target database).

account.member:

```
<property name="eTDYNMember">
  <value>
    <!-- Automatically triggers DN validators and converters, meaning that
    DNS are checked to ensure that they reference existing objects etc -->
    <setValue><baseType><flexiStrValue type="DN"/></baseType></setValue>
  </value>
  <metadata name="displayName">
    <value><strValue>Accounts</strValue></value>
  </metadata>
  <!-- still need a connector-speak name for the field -->
  <metadata name="connectorMapTo">
    <value><strValue>member</strValue></value>
  </metadata>
  <metadata name="refObjectType">
    <value><strValue>eTDYNAccount</strValue></value>
  </metadata>
  <metadata name="assocTable">
    <value><strValue>acc_grp_assoc</strValue></value>
  </metadata>
  <metadata name="assocTableObjNamingAttr">
    <value><strValue>grp_name</strValue></value>
  </metadata>
  <metadata name="assocTableRefNamingAttr">
    <value><strValue>acc_name</strValue></value>
  </metadata>
```

```

<metadata name="DNLDAPObjectClass">
  <value><strValue>eTDYNAccount</strValue></value>
</metadata>
<!-- Let the framework take care of verifying that contained DNS reference
existing objects, rather than having to do this explicitly in connector code. -->
<metadata name="DNTestExists">
  <value><boolValue>true</boolValue></value>
</metadata>
<!-- Connector wants only names of accounts by the time the data gets to it.
-->
<metadata name="DNNameOnly">
  <value><boolValue>true</boolValue></value>
</metadata>
</property>

```

group.memberOf:

```

<property name="eTDYNMemberOf">
  <value>
    <!-- Automatically triggers DN validators and converters, meaning that
DNS are checked to ensure that they reference existing objects etc -->
    <setValue><baseType><flexiStrValue type="DN"/></baseType></setValue>
  </value>
  <metadata name="displayName">
    <value><strValue>Groups</strValue></value>
  </metadata>
  <!-- still need a connector-speak name for the field -->
  <metadata name="connectorMapTo">
    <value><strValue>memberof</strValue></value>
  </metadata>
  <!-- Is "logically inserted" in Java CS processing
  <metadata name="isMultiValued">
    <value><boolValue>true</boolValue></value>
  </metadata>
  -->
  <metadata name="refObjectType">
    <value><strValue>eTDYNGroup</strValue></value>
  </metadata>
  <metadata name="assocTable">
    <value><strValue>acc_grp_assoc</strValue></value>
  </metadata>
  <metadata name="assocTableObjNamingAttr">
    <value><strValue>acc_name</strValue></value>
  </metadata>
  <metadata name="assocTableRefNamingAttr">
    <value><strValue>grp_name</strValue></value>
  </metadata>

```

```
<metadata name="DNLDAPObjectClass">
  <value><strValue>eTDYNGroup</strValue></value>
</metadata>
<!-- Let the framework take care of verifying that contained DNS reference
existing objects, rather than having to do this explicitly in connector code. -->
<metadata name="DNTestExists">
  <value><boolValue>true</boolValue></value>
</metadata>
<!-- Connector wants only names of groups by the time the data gets to it. --
>
<metadata name="DNNameOnly">
  <value><boolValue>true</boolValue></value>
</metadata>
</property>
```

With indirect associations, the runtime cost for looking up the association attribute is the same in either direction (for example, `group.member` or `account.memberOf`) unlike the direct case where an expensive virtual attribute has to be used in one direction.

How Metadata Is Used

The Java CS framework reads the metadata provided for an endpoint type, and from it learns both the data model for the connector and configures how each object class and attribute should be processed. Object classes and attributes which are mentioned in requests to a connector instance, for which there are no *connectorMapTo** settings, are simply logged and discarded, which allows the connector's implementation's coverage to be built incrementally (especially when porting an existing C++ connector to Java).

Some client applications, for example, the Provisioning Server, pass on some attributes to the Java CS even though they are irrelevant to it. To this end, the *acceptedUnknownAttrIds* property can be configured in `connector.xml` to list the names of attributes which are not relevant to the connector implementation, but should nonetheless not be logged when received. There is also a *acceptedUnknownAttrIds* setting in `server_jcs.xml` for attributes global to the whole Java CS, rather than a specific connector.

A message is logged at the INFO level to `jcs_daily.log` for each object class, summarizing all of its attributes which are mapped in metadata, for example:

```
INFO - class='eTDYNGroup': all mapped attributes=eTDYN-int-01;eTDYNMember [expensive];eTDYNGroupName;eTDYN-str-01
```

The syntax chosen for the list in the message aids cutting and pasting it into the list of requested attributes in JMeter search tests (after qualifications in square brackets have been deleted, for example *[expensive]* in the preceding expression, if required).

Association Related Code

At runtime, associations are encapsulated in instances of the `com.ca.jcs.assoc.Association` class and managed by attribute-style processors deriving from either `DefaultAssocDirectAttributeOpProcessor` (which may provide reverse association logic to you for free) or implementing `AssocIndirectAttributeOpProcessor` (the direct case is more amenable to a reusable default implementation). Therefore, consider carefully which class to use as the basis for your connector's attribute-style processor.

Note: The `AssocAttributeOpProcessorProx` can be used to provide default reverse association handling for connectors which use direct associations, and is triggered by your connector returning true from its `isAutoDirectAssocRequired()` method (which defaults to `!isIndirectAssociations()`), where the `getAutoDirectAssocExclusions()` method can also be overridden to name operations you wish to exempt from this processing, if required). Its full source file is bundled with the SDK and provides a reference for the appropriate pre and post processing needed to handle associations around each type of operation and LDAP request.

Chapter 8: Implementing Connectors

This section contains the following topics:

[How to Implement a Connector](#) (see page 85)

[Connector Base Classes](#) (see page 86)

[Style Processors](#) (see page 88)

[How To Test a Connection](#) (see page 94)

How to Implement a Connector

To implement a connector, do the following:

1. Determine which values need to be passed to the endpoint system in order to establish a connection, and decide which LDAP attributes should be used to pass these values (on the connector level of the DIT, or connector objectclass).
2. Write connector metadata, paying special attention to connection-related attributes on the connector's objectclass.
3. While defining the metadata, incrementally write and test the related connector logic.

Note: For more information, see [Create New Metadata](#) (see page 57).

4. Decide whether connection pooling support (between connector and endpoint) is required. If using the default support built-in to the Java CS, then you only need to write a class extending `org.apache.commons.pool.BasePoolableObjectFactory`.
5. Test first with JXplorer and then with a JMeter (or equivalent) component test.

Implementation Guidelines

Consider the following guidelines when designing and implementing a connector:

- Drive as much of the connector implementation logic as possible using metadata. This is the same approach used for the Java CS core framework where generic problems are encountered, such as adding support for *connectorMapToAmbiguous=metadata* mappings for the JNDI connector.
- When writing code, take advantage of the service provided by the Java CS framework, like pluggable validators and converters, and connection pooling support classes.
- Write custom connector code to address any additional specific coding requirements.

Note: This is not an either or situation of using metadata versus custom coding, but rather a case of treating custom coding of connector behavior as the last resort.

Connector Base Classes

You need to decide which type of connector you want to implement. The Java CS SDK provides several abstract connector base classes you can extend:

- `com.ca.jcs.BaseConnector`—Implement this class to implement all connector behavior in the Java code of the connector itself.
Note: Extending `MetaConnector` is a much faster and less error-prone alternative because most of the logic is driven by flexible metadata rather than static Java code.
- `com.ca.jcs.meta.MetaConnector`—A connector needs to translate LDAP concepts such as DNs to identifiers on the endpoint system, and to map LDAP attribute name to native object concepts. `MetaConnector` and all its subclasses perform this job, and also take care of basic attribute validation and conversion tasks as outlined in the data model metadata stored on their parent endpoint type. This class is the basis for all metadata-driven connectors.

Most endpoint systems have a flat (non-hierarchical) structure, which is reflected by extending from `MetaConnector` and overriding its `isBehaviourSearchSingleClass()` method to return *true*. This causes the framework to call your connectors' attribute-style processor's `doSearch()` method with one object class at a time (even when a SEARCH filter matches multiple object classes), greatly simplifying the implementation of this method.

However, for performance and logical grouping reasons, it is generally best to present objects of the same type (accounts/groups) as contained in their own virtual container. This class takes care of presenting these logical virtual containers on your behalf where virtual containers are specified in the connector's `conf/connector.xml` configuration file.

Note: For more information, see the SDK example connector.

Note: It is now considered better practice to define virtual containers more dynamically in metadata. For an example, see the definition of the `eTDYNAccountContainer` class in `jcs-sdk-home/connectors/sdkdyn/conf/sdkdyn_metadata.xml`). Most custom connectors can be implemented by extending this class.

If your custom connector supports a hierarchy, (such as an LDAP/JNDI directory), and you want to represent this information in your connector, this class (or one of its derived classes) is the recommended starting point. If the endpoint system search semantics map clearly to LDAP search filters that can match multiple objectclasses, then `isBehaviourSearchSingleClass()` should not be defined by your connector (it will then default to false).

Other Boolean behavioral methods include:

- **isBehaviourSearchObjAsLookup()**—Should a SEARCH on a single object be turned into a `doLookup()` call to simplify the implementation of your connector's attribute-style processor's `doSearch()` method?
- **isObjectClassRequired()**—Should the objectclass be passed through in the attributes passed to your connector (by default this is set to `!isBehaviourSearchSingleClass()`).
- **isHiddenLdapBaseDn()**—Should the base DN be hidden for a hierarchical connector?
- **isBehaviourStrictConnectorDns()**—Should connector-speak DNs be handled as strictly RFC 2253 conformant? Defaults to false.
- **isIndirectAssociations()**—Does your connector represent associations between objects by using a *table* external to both of them (as JDBC does)? Otherwise *direct* associations are assumed instead as used by JNDI where the members of a group are stored directly on the group concerned.
- **isHiddenLdapBaseDn()**—Should the base DN be hidden for a hierarchical connector?

- **isAutoDirectAssocRequired()**—If your connector makes of direct associations, then returning true from this method will cause the Java CS framework to make use of the `com.ca.jcs.assoc.AssocAttributeOpProcessorProxy` class to implement virtual reverse association. attributes for your connector.

Note: For more information, see the SDK example and the Java CS Javadoc.

Style Processors

There are three Java CS style processor types for implementing connectors:

- [Attribute Style Processor](#) (see page 91)
- [Method Style Processor](#) (see page 88)
- [Scripting Style Processor](#) (see page 91)

Method Style Processor

The Method Style Processor maps LDAP operations to native PRE,OP and POST methods invoked on the endpoint system (for example, stored procedure support in JDBC connector). These opbindings can therefore be used to customize or replace the logic coded for `doAdd()` / `doModify()` methods on your connector's attribute-style processor.

The metadata used to express this style adheres to the `opbindings.xsd` and `opattributes.xsd` schemas. The method-style languages on endpoint systems are assumed to be much less powerful than those accessed using script-style bindings, and hence much of the content for opbindings of this style is concerned with formatting the parameters to be passed into or out of the native methods which are invoked.

Method Payload Parameters

The `MethodPayloadType` in `jcs-sdk-home/conf/xsd/opbindings.xsd` specifies a method payload which consists of the name of a native method to execute (its method attribute) and a number of bindings for each parameter this method expects. Each parameter (of type `ParameterBindingType`) specifies:

- An attribute name
- The native method parameter name to be bound to it when the method is called
- The Boolean flags specifying whether the parameter should be used to pass a value into the native method (input), or out of the native method (output), or both.
- The handling of multi-valued attributes is addressed through the `multiValuedFlattenStyle` and `multiValuedModifyMode` attributes, where multiple values can be *flattened* (or indeed the collection of all attribute values could be encoded using `<jcs-sdk-home>/conf/xsd/opattributes.xsd`) into a single string literal. The motivation behind this functionality is that the native languages behind the method-style mappings (for example, Stored Procedures for JDBC) are likely to have limited power to work with structured parameters, so the Java CS framework performs flattening and mapping simplifications of parameters on their behalf.

In addition to the attributes mapped in the datamodel metadata for the object class targeted by an method opbinding, the following table shows the special values that can also be used as the value for the attribute value of any method payload parameter. Use of the special attribute names allow runtime context information known to the Java CS framework to be passed into the native method, or in the case of `*ErrorStatus*`, to be passed out of it.

Contextual Attribute	Direction	Description	Applicable LDAP Operations
<code>*NAME*</code>	IN	Target object's most nested RDN value by itself	All
<code>*DN*</code>	OUT	Target object's full distinguished name	All
<code>*ErrorStatus*</code>	IN	Should remain null unless an error occurs in which case a description can be passed back.	All update operations: ADD / MODIFY / DELETE / MODIFY_RN

Contextual Attribute	Direction	Description	Applicable LDAP Operations
AddModify_AttrsAsXML	IN	XML representation of entire ADD or MODIFY is passed in as a single string (refer opattributes.xsd)	ADD or MODIFY
ModifRny_NewRdn	IN	New RDN	MODIFY_RN
Move_NewParentName	IN	New parent name	MOVE
MoveRename_NewRdn	IN	New RDN (may be null)	MOVE

Note: Method-style bindings to the query LDAP operations (LOOKUP and SEARCH) are not supported. Also, the setting of the lookUpLevel attribute is important to the handling of POST delete method opbindings, which are special as attribute values may need to be cached before the target object is actually deleted (however, always caching may impose to much of a performance burden).

Scripting Style Processor

The Scripting Style Processor maps LDAP operations and attribute into scripted output which is then submitted to the endpoint system for processing.

It is similar to method-style processing except that the opbindings are tied to scripts to be executed (for example, in JDBC scripts can be executed that perform logic directly, or alternatively, scripts can be executed to generate SQL which is then executed as a separate step).

Scripting support for `executedDirectly=true` opbindings is provided via the Java CS framework and does not require any special support from your connector. A script-style processor is only needed to support opbindings for which `executedDirectly=false`, in which case the script generates a string of native code to be executed. For example scripts bound to the JDBC connector might produce text containing SQL commands to be later executed by `com.ca.jcs.jdbc.JDBCScriptStyleOpProcessor`.

The metadata used to drive this style of processing adheres to the `opbindings.xsd` schema, but unlike method-style processing, its opbindings have scripting payloads. Because of the relative power of scripting languages, the arguments to Java methods are passed in as-is to the scripts, rather than mappings for them having to be defined in the metadata as is required for method-style processing. Currently, only script-style opbindings can be bound to query operations: LOOKUP, and SEARCH.

Attribute Style Processor

The Attribute Style Processor maps LDAP attributes to endpoint attributes, usually through Java CS framework support driven by metadata.

The most commonly implemented style using metadata-driven mappings are from LDAP objectclasses and attributes to connector equivalents on the endpoint system. For example, JDBC objectclasses are mapped to table names and attributes are mapped to column names.

The metadata used to drive this style of processing adheres to the `datamodel.xsd` XML schema.

Style Processor Methods

Connectors advertise their support for each style of processor by implementing the corresponding method in the `com.ca.jcs.processor.OpProcessorStyleFactory` interface to return a processor instance as shown in the following table:

Style Processor Type	Methods to Implement
Attribute Style Processor	<code>createAttributeStyleOpProcessor ()</code>
Method Style Processor	<code>createMethodStyleOpProcessor()</code>
Scripting Style Processor	<code>createScriptStyleOpProcessor()</code>

Connectors may implement one or more of the preceding styles. Where multiple styles are implemented, this can result in multiple processors being applied to a single LDAP request (as dictated by data model and opbindings metadata content). For example, a web service connector would only implement method and script-style processor (RPC or document-style) and the JDBC connector implements all three styles.

Note: For an example, see the SDK connector.

How Connectors Work

The Java CS connectors return an instance of the `com.ca.jcs.ConnectionManager` class using the `getConnectionManager()` method to their constituent processors and to the Java CS framework.

As the SDK connector works by persisting data to local files, its notion of a connection is a bit contrived, and is implemented to return a reference to the parent directory into which object data files should be written. The SDK attribute-style processor's methods have been wrapped in try and catch blocks, to demonstrate how code which needs to access a connection manager should be structured.

Note: For more information see, `com.ca.jcs.sdk.SDKAttributeStyleOpProcessor`, `com.ca.jcs.sdk.SDKConnectionManager` and the source files included with the SDK.

Connection Pooling Considerations

Wherever possible, you should write `ConnectionFactory` implementations as pools to provide scalability benefits. The utility class `com.ca.jcs.cfg.GenericObjectPoolConnectionFactory` is useful in case the endpoint system doesn't have connection pooling built-in. If this approach is used, then most of the work is done by the class derived from `org.apache.commons.pool.PoolableObjectFactory` (from the Jakarta Commons Pool open source library), which opens and closes connections accessed using the pool.

In most cases, you may need to:

- Write a small class which extends `BasePoolableObjectFactory` and gets and releases connections in its `makeObject()` and `destroyObject()` methods, which is used by the pool to manage the raw connections contained by it.
- Write another small stub class extending `GenericObjectPoolConnectionFactory` which contains any additional custom logic you require.

Note: For an example of how you should code such a manager and factory, see `com.ca.jcs.jndi.JNDIConnectionFactory`, `com.ca.jcs.jndi.JNDIConnectionPool`, and the source files included with the SDK, which are bundled with the SDK.

Note: The `com.ca.jcs.cfg.GenericObjectPoolConfigBeanWrapper` class provides a mechanism for you to configure common properties on your pool using your connector's `conf/connector.xml` file.

The `activate()` method is invoked when your connector's LDAP interface receives its first LDAP request after the Java CS is started, and also when the client modifies any attribute you have flagged with the `isConnection` metadata. When invoked, this method should look up the attribute values using the `getAttributes()` method, establish a connection to the endpoint system (or preferably a connection pool), and throw appropriate LDAP exceptions if the endpoint system cannot be contacted or the supplied credentials are invalid.

Override the `deactivate()` method of your connector to perform any cleanup routines that you want to perform inside your connector when the connector receives a message from the Java CS that it is about to be shutdown, for example:

- Closing session pools
- Deleting temporary files
- Closing open references to files

How To Test a Connection

To test a connection, do the following:

1. Deploy your connector to the Java CS by running *ant dist* from the top-level *jcs-sdk-home* directory.
2. Start the Java CS using an IDE configuration (this SDK includes configurations for the Eclipse and IDEA IDEs). Starting from within an IDE allows you to debug, but you can also run the Java CS using *jcs-dir/build/dist/bin/jcs.bat* (Windows) or *jcs-dir/build/dist/bin/jcs.sh* (Solaris).
3. Submit the LDAP ADD request to create the parent endpoint type, and check for a successful response code. At this stage the Java CS will have validated and stored the metadata you provided for the endpoint type. For static endpoint types, it will have been read from within the connector's *.jar* as configured through the contained *connector.xml* file. For dynamic cases, the metadata will need to be included as the value for the *eTMetaData* attribute in the ADD request.
4. Submit a second LDAP ADD request for an endpoint using a DN directly under the parent endpoint type you just created) containing all of the required attributes for the connector (both connection-related and otherwise).
5. Check for a successful response code.

Chapter 9: Writing Scripts

This chapter describes how logic written in scripts can be used to customize the logic of an existing connector (for example, like a program exit) or write a complete connector (like the SDKSCRIPT connector included in this SDK). In both cases, the logic is bound to the processing of LDAP operations (ADD/MODIFY and such) using opbindings metadata documents with script payloads.

Note: For more information see, *Metadata Syntaxes*. (see page 51)

The chapter precedes sections that describe each area of connector implementation, as the content of these chapters is largely independent of whether implementation is undertaken in Java or JavaScript.

This section contains the following topics:

[Implementing in Java or JavaScript Considerations](#) (see page 96)

[How you Pass Data to and from Scripts](#) (see page 98)

[LOOKUP and SEARCH Query Operations through Script Opbindings Considerations](#) (see page 100)

[Pure Scripted Connectors](#) (see page 101)

[Scripted Logic Update Considerations](#) (see page 102)

Implementing in Java or JavaScript Considerations

Whether to implement in Java or JavaScript encompasses a number of considerations:

- Almost anything that can be implemented in Java can be implemented in JavaScript, so the relative power of each approach is not a big consideration. In particular, note that the Java CS scripting support allows streaming of search results through the use of the *searchResultsBlockingQueue* scripting variable.
- Scripting languages tend to speed up the edit and test cycle as no recompilation or JCS restarts are required when JavaScript code is changed. However, they are much less strict in their type safety checking, so thorough testing is the only way to find the bugs that a Java compiler would pick-up at build time. Therefore, scripting is perfect for minor customizations or proof of concepts, but Java must be considered seriously for larger production connectors.
 - Where minor customer-specific customizations of an existing connector are the focus, scripted opbindings are a good option.
 - Where most of the logic for a connector depends on fairly simple textual manipulation (for instance preparing specially formatted arguments to be passed as command line arguments to existing native endpoint system executables), use of script should be seriously considered.
- If the customizations are more far-reaching then writing a custom connector derived from the classes of the existing connector is easy to achieve using the JCS framework, where writing a few specialized classes and referencing their names in a new connector.xml may be all that is required. Note that in such circumstances implementation code can be shared even if a completely different LDAP schema is to be used for the new specialized connector – this is the concrete benefit of referencing only connector terminology attribute names in connector code.
- There is a slight performance cost to using scripts compared to Java code, and this cost grows proportionate to the total size of the script being executed (currently each script has to be reparsed in its entirety whenever any part of it is executed).
- It is quite feasible to start a connector off as a scripted solution and later migrate it to Java if its code grows beyond initial expectations and becomes hard to maintain and or nonperformant.

Important! It is very important that you pay careful attention to XML quoting issues so that scripts are not corrupted when they are included as fields within an opbindings XML metadata document. For example, you need to replace < characters in a script with < when the script is included in an XML document.

Note: The scripting language supported for the Java CS is JavaScript as provided by the Rhino opensource project, which Sun Microsystems bundle with JDK 6 onwards. For more information, see the Rhino opensource project at <http://www.mozilla.org>.

How you Pass Data to and from Scripts

You can execute either of the following formats of script (as reflected in the opbindings.xsd XML schema definition):

1. One or more global scripts can be defined at the head of the opbindings XML file.

As a result, the individual bindings cause the execution of individual functions within these scripts. When this style is used, the exact same arguments passed to the attribute-style processor's method are passed to the corresponding target scripting function. For example, a scripting function targeting an ADD operation will be passed an `ObjectInfo` instance as its first argument and an `Attributes` object as its second argument because these are the arguments to `com.ca.jcs.processor.OpProcessor.doAdd(ObjectInfo, Attributes)`. This is the preferred approach for all but the simplest scripts, as it allows reuse of utility functions between multiple scripts.

Note: For more information on the arguments passed to script function targeting a MODIFY operation, see `com.ca.jcs.processor.OpProcessor.doModify(ObjectInfo, ModificationItem[])`.

2. Alternatively, each opbinding can be tied to a complete self-contained script (instead of to a function contained with a script). In this case, each of the arguments to the attribute-style processor's method are bound to script variable names using the exact arguments names and Java structures as defined in the `OpProcessor`'s `JavaDoc`. For example, a script targeted an ADD operation `doAdd(ObjectInfo objInfo, Attributes attrs)` will be called with two scripting variables defined:

- `objInfo` will be bound to a Java object of type `ObjectInfo`
- `attrs` will be bound to a Java object of type `Attributes`.

For both formats the following additional scripting variables are also bound:

- The zeroth element of special scripting variable `statusArray` (of type `String[]`) can be assigned a string value to signify an error condition, which will then be passed back to the client (and if `strictCompletion` is true may cause the whole LDAP operation to fail). In JavaScript it is usually better to throw an exception than to make use of this variable.
- The variable `connector` is bound to the parent connector which owns the attribute-style processor being invoked. Through this variable the script can access the connector's parent connector type (and hence the metadata settings) and the JCS framework. For example, a script looking up the value of an attribute stored on the connector in order to modulate its behavior accordingly.

- For the opbindings targeting the MODIFY operation, the script variable `currAttrs` can be used to access the current state of the target object (in connector-speak) where this required. For example, a script may need to check that a single-valued attribute currently has no values before adding a new value for it.

Note: For more information on query related scripting variables and other notes on queries, see LOOKUP and SEARCH query operations through Script opbindings Considerations.

Only script opbindings with their `executedDirectly` Boolean field set to false require a connector to have a script-style processor, as such opbindings produce connector-specific text (for example, SQL for the JDBC connector) which only the connector knows how to execute. All other opbindings will be invoked by the Java CS framework without any special support being required from the target connector.

LOOKUP and SEARCH Query Operations through Script Opbindings Considerations

Consider the following when dealing with the LOOKUP and SEARCH query operations through script opbindings:

- Query Opbindings with Timing=PRE are not supported.
- LOOKUP opbindings with Timing=POST can access the attributes returned from the operation using the `attrs` scripting variable.
- Directly-executed SEARCH opbindings with Timing=OP can return search results via a `NamingEnumeration` as some form of `java.util.Collection`, where a simple synchronous implantation is chosen. However, it is also possible to specify that directly-executed SEARCH opbindings with Timing=OP should be executed asynchronously by setting `asynchronousSearch=true` on their guards. When this is done, the triggered script must queue search results by adding them to the queue bound to the `searchResultsBlockingQueue` scripting variable, and should just return null when it has finished its processing. In this case, the Java CS framework automatically runs the SEARCH script in a separate thread and streams results back to the client asynchronously as they become available.
- Opbindings with Timing=OP (instead-of) are supported and mean that the script will process the query, rather than the target connector being wrapped. Note that the most difficult aspect in implementing SEARCH operations (whether using Java code or scripting) is the issue of manipulating and filter based on the provided LDAP filter.

Note: For more information, see *Querying Connector Objects*.

- The Java CS framework allows some flexibility in the way that LOOKUP and SEARCH results are represented. Instead of returning an `Attributes` object LOOKUP, operations may choose to return a `Map` and SEARCH operations can return either a `NamingEnumeration` or any form of `java.util.Collection` of `Maps` instead of equivalent `javax.naming.SearchResult` values. Where use of `Map` is chosen, they must be compatible with the signature `Map<String, Object>`.

Note: `BaseConnector.CONN_NAME (!name!)` and `LdapUtil.LDAP_OBJECT_CLASS (objectclass)` are mandatory map entries with special significance to the framework.

- POST bindings on SEARCH operations need the following special handling:
 - Directly-executed synchronous SEARCH opbindings with Timing=POST can step through search results queued by the SEARCH operation itself by using the `NamingEnumeration` bound to the `searchResults` scripting variable.

- Search results need to be processed by iterating through them one-by-one (recall that `doSearch()` returns a `NamingEnumeration`). This means that the streaming of search results is not supported when a `SEARCH` opbinding with `Timing=POST` timing is registered. It also means that `ScriptStyleOpProxyHandler` has to collate all results against each opbindings guard, so that each script is only invoked once for all objects matching it (rather than a script being executed for each individual search result).
- Each `NamingEnumeration` can contain objects with any number of objectclasses depending on the LDAP filter passed in. This is one of the prime motivators that opbinding guard's allow multiple objectclasses to be specified.

Pure Scripted Connectors

The fully functional SDK script connector bundled with this SDK is an example of a 100% scripted connector.

`OpBindings` always wrap the operations of a target connector, so fully a scripted connector can make use of `com.ca.jcs.stub.StubMetaConnector` for this purpose. It is a connector that does nothing, except:

- Allow you to customize settings through `connector.xml` that are normally statically coded in a concrete `MetaConnector` implementation.
- Register an attribute-style processor with `do*()` method implementations that throw `LdapOperationNotSupportedException` when they are intended to always be wrapped by `OpBindings` which having `Timing=OP`. If your connector makes use of direct associations, then you should consider specifying *`autoDirectAssocRequired=true`* in your `connector.xml` (for example, like the sample `SDKSCRIPT` does) so that `StubDirectAssocAttributeStyleOpProcessor` will be used and the Java CS will provide reverse association support without any coding being required on your part.

Scripted Logic Update Considerations

While writing scripts or enhancing them, consider commenting out the *staticMethodScriptStyleMetaDataFile* property from your connector's `connector.xml` file, and instead provide values for the endpoint type's *eTopBindingsMetaData* attribute explicitly in LDAP ADD and MODIFY requests. This allows you to test new scripting logic through:

- A simple LDAP MODIFY passing in the value of a variable assigned using a *Var From File Controller*, through which the new `<connector>_opbindings.xml` file can be read.
- Cutting and pasting the content of the new `<connector>_opbindings.xml` into JXplorer.

However, for production usage, we recommend that you reinstate the *staticMethodScriptStyleMetaDataFile* property setting in `connector.xml`, therefore stopping changes to the endpoint type's *eTopBindingsMetaData* attribute which may subvert your connector's implementation.

Chapter 10: Creating Endpoint Objects

This chapter describes how to create new objects on the endpoints with which your connector communicates. Creation of all object classes managed by the connector (for example, accounts and groups) is routed through the same `doAdd()` method of the attribute-style processor you returned from your connector's `createAttributeStyleOpProcessor()` method.

Creation of a new connector instance itself is not routed through this method, but instead is handled by the Java CS framework which calls your connector's constructor and `activate()` methods. In some cases, you may want to define and register a specialized `com.ca.jcs.processor.ConnectorAttributesProcessor` to handle the attributes stored for the connector itself, for example, if you want to support an attribute which has a value that needs to be calculated in code.

If your method needs to create an object that already exists, an `LdapNameAlreadyBoundException` should be thrown.

This section contains the following topics:

[How you Create an Object](#) (see page 103)

[Add Operation Testing](#) (see page 105)

How you Create an Object

Depending on your choice of endpoint type, implementing the add operation involves extending `AbstractAttributeStyleProcessor` and implementing the following methods:

Note: See the Java CS Javadoc for descriptions of parameters, and the SDK Sample connector for a complete sample implementation.

Implementing `AttributeStyle`

```
public void doAdd(final ObjectInfo objInfo,  
                 final Attributes attrs)  
    throws NamingException
```

Note: For more information, see `com.ca.jcs.processor.OpProcessor.html#doAdd(com.ca.jcs.ObjectInfo,%20javax.naming.directory.Attributes)`

Implementing AssocAttributeProcessor (implementing associations)

```
public void doModifyAssocs(final ObjectInfo objInfo,
                           final AssocModificationItem[] items,
                           final Object context)
    throws NamingException

public void addAttrAssocs(final ObjectInfo objInfo,
                           final Association assoc,
                           final Attribute attr,
                           final Object context)
    throws NamingException
```

Implementing doAdd(ObjectInfo objInfo, Attributes attrs) throws NamingException

Consider checking the existence of a object with the same name before trying to add the object.

You need to update the endpoint system to record the object's creation, given a reference to it (`objInfo`) and the attributes to be assigned to it (`attrs`). If multiple object types can be created, you can distinguish which type is being requested by examining `objInfo.getObjectClassMapping().getConnectorClassName()`. This yields the *connectorMapTo* defined in the metadata for this object (that is, the connector terminology representation of the class name). In some cases the alias or LDAP terminology class name may also be useful in distinguishing which object class is being targeted.

As with all `do*()` methods in the connector interface, all attribute names, values, and filters have been validated, converted, and mapped to connector-terminology before your method is called.

The provided attributes need to be persisted on the endpoint system. For example, a JDBC connector would translate this list into the column names and values to be mentioned in an SQL INSERT clause to be executed on the endpoint.

To minimize references to LDAP attribute names in your connector code, consider using the values of the *connectorMapTo* or *connectorMapToAlias* metadata properties when deciding how to process an object.

To minimize or avoid checks for syntactic validity on attribute values, consider using an attribute validator.

To minimize or avoid manipulating attribute values, consider introducing an attribute converter.

The Java CS SDK includes a library of built-in validators and converters. However you can also write your own and connect them to objectclasses and attributes using metadata definitions.

If your connector needs to handle associations then you also need to get the list of associations and handle adding them. This can be accomplished by the following code snippet:

```
// splitAssocAttrs will remove the associations from attrs and return them
assocAttrItems = objInfo.getObjectClassMapping().splitAssocAttrs(attrs);

// create context and implementation of object creation here

// now hand off adding the associations
if (assocAttrItems != null)
    doModifyAssocs(objInfo, assocAttrItems, context);
```

Implementing doModifyAssocs(ObjectInfo objInfo, AssocModificationItem[] items, Object context) throws NamingException

This method is passed a list of modification items, including additions, deletions and replacements. This method is responsible for calling getModificationOp() on each modification item and then handing off the work to addAttrAssocs() or removeAttrAssocs() as appropriate. Use this method if your modification items are not independent of each other (for example, if ordering is significant).

Note: For this release, modifications are limited to additions only.

Implementing addAttrAssocs(ObjectInfo objInfo, Association assoc, final Attribute attr, Object context) throws NamingException

This method is called to create a single association of the type described by assoc from the objInfo object.

Coupled with objInfo.getName() and attr.get().getValue(), which return the name of the source object and the name of the target association respectively, you can construct the endpoint relationship.

Note: For more information, see com.ca.jcs.assoc.Association.

attr can hold multiple values indicating several relationships of the same type to different target objects. Depending on the capabilities of the endpoint system, you may need to issue separate creation statements or a single statement listing multiple parameters.

Add Operation Testing

At this stage you should be able to do the following:

1. Create the endpoint type object within the Provisioning Manager.
2. Create the connector object within the Provisioning Manager.
3. Create managed object types.
4. Create associations between managed objects.

The Jmeter test performs step1 and 2 and then performs multiple instances of steps 3 and 4. For your initial development work you may want to drive simple creations from a LDAP client. However, once your add operation is complete you will need to create an automated test scenario that covers the following test coverage objectives:

- Creation of all object types and creation of all possible attributes.
- Creation of all possible object associations.
- Failure cases where objects cannot be created because the object already exists, access permissions are insufficient, or other appropriate endpoint failures cases.

These tests cases should make use of the scripted tests for creating connectors and directories you developed earlier to construct a working environment.

Chapter 11: Removing Endpoint Objects

This section contains the following topics:

[How you Delete an Object](#) (see page 107)

[Example: Implementing doDeleteAssocs](#) (see page 109)

[Example: Calling doDeleteAssocs\(\) Methods Inside the doDelete and doModifyRn](#) (see page 109)

[Delete Operation Testing](#) (see page 110)

How you Delete an Object

Removing an endpoint object may be the easiest LDAP operation to implement in a connector. In general, this operation does not involve any object attributes, but simply requires passing the naming identifier of the object to the delete method on the endpoint system.

To remove an object from the endpoint system, implement the doDelete method in the Java CS SDK:

```
public void doDelete(ObjectInfo info)
```

You need to verify that the object you want to delete is actually there. To do this, retrieve the naming identifier from the ObjectInfo parameter and call your endpoint system SDK existence check or search method.

Note: For more information, see [com/ca/jcs/processor/OpProcessor.html#doDelete\(com.ca.jcs.ObjectInfo\)](#) and, the SDK Sample connector for a complete sample implementation.

If the object is not in the endpoint system, you need to throw the following exception:

```
org.apache.directory.shared.ldap.exception.LdapNameNotFoundException  
    boolean isThere = api.searchForObject(info.getName());  
    if(!isThere)  
    {  
        throw new LdapNameNotFoundException(  
            info.getLdapDn() + " does not exist");  
    }
```

Next, call the delete method using your endpoint system API.

A common problem when deleting objects is that the credentials being used by the connector contain insufficient privileges to perform a delete (`org.apache.directory.shared.ldap.exception.LdapNoPermissionException`).

Your code should also account for a possible transient condition, such as a communication exception. In this case, you need to throw the following exception:
`org.apache.directory.shared.ldap.exception.LdapServiceUnavailableException`.

Lastly, you may need to perform subsequent cleanup on any other objects that contain references to the object that you just deleted. For example, membership references to this account may exist in other group objects.

Some APIs (especially those not supporting transactional behavior) may prevent an object from being deleted before all references to it have been cleaned up. In this case you will either need to inform your customers of this restriction, or code your `doDelete()` method to clean up references prior to deleting the target object (probably by calling its implementation of `com.ca.jcs.assoc.AssocAttributeOpProcesso.doDeleteAssocs()`).

Where association handling logic built into the JCS is not sufficient, it may be possible to use the `AttributeStyleProcessor` `doSearch(ObjectInfo info)` and `doDelete(ObjectInfo info)` methods as a basis for your own custom logic, but you will need to be very sure that you understand exactly what format connector terminology speak search results (especially if your connector makes use of any structural converters)..

Example: Implementing doDeleteAssocs

The following is an example of how you implement doDeleteAssocs:

```
public void doDeleteAssocs(final ObjectInfo objInfo, final Object context)
throws NamingException
{
    doAssocUpdateReferencesTo(objInfo, null, connector);
}
```

This command has the following format:

objInfo

Specifies the object to be modified or deleted.

context

(Optional) Provides additional context for the requested updates. For example, transactional connectors may want the updates of the associative relationships to occur within a larger transaction.

The

com.ca.jcs.assoc.DefaultAssocDirectAttributeOpProcessor.doAssocUpdateReferencesTo() method constructs the search filter and invokes the doSearch method of this connector's attribute-style processor to retrieve the associative objects (for example, Account). It then constructs the modification item and invokes the doModify() method to update the associative attribute for the associative object (group members attribute in the Account).

Example: Calling doDeleteAssocs() Methods Inside the doDelete and doModifyRn

The following is an example of Calling doDeleteAssocs() Methods inside the doDelete and doModifyRn:

```
// if there are any associations referencing objects of this type defined in
the meta data file

if (!info.getObjectClassMapping().getToAssociations().isEmpty())

    doDeleteAssocs(info, null);
```

Delete Operation Testing

At this stage you should be able to do the following:

1. Create the endpoint type object within the Provisioning Manager.
2. Create the connector object within Provisioning Manager.
3. Create managed object types
4. Create associations between managed objects
5. Delete all managed object types

The new items in step 5 are operations that require tests. For your initial development work, you may want to drive simple creations from a LDAP client. However, once the delete operation is complete, you will need to create an automated test scenario that covers the following objectives:

- Deletion of all object types
- Associated deletion of all related object associations.
- Failure cases where objects cannot be deleted because access permissions are insufficient, or other endpoint failures cases

These tests cases should make use of the scripted tests for creating connectors and directories you developed earlier to construct a working environment.

Chapter 12: Querying Connector Objects

This section contains the following topics:

[How You Search for an Object](#) (see page 111)

[How you Test the Search Operation](#) (see page 115)

How You Search for an Object

The Java CS framework provides an abstract class `AbstractAttributeStyleProcessor` which your connector can implement. You need to implement the following methods to handle the search for endpoint objects:

```
Attributes doLookup(ObjectInfo objInfo, String[] attrIds) throws
NamingException;
NamingEnumeration doSearch(Name baseName,
    FilterInfo filterInfo,
    Map environment, SearchControls searchControls)
    throws NamingException;
NamingEnumeration doList(ObjectInfo objInfo) throws NamingException;
```

Note: For more information, see

`com/ca/jcs/processor/OpProcessor.html#doLookup(com.ca.jcs.ObjectInfo,%20java.lang.String[]` and `com/ca/jcs/processor/OpProcessor.html#doSearch(com.ca.jcs.ObjectInfo,%20com.ca.jcs.filter.FilterInfo,%20java.util.Map,%20javax.naming.directory.SearchControls`. See the SDK Sample connector for a complete sample implementation. For descriptions of parameters, see the Java CS Javadoc.

All values of DN, search filter, and return attribute IDs will have been mapped to the connector by the time these methods are called (that is, all values are in connector terminology) if you provided *connectorMapTo=values* in your connector metadata file.

Any required validators or converters have been run by the Java CS framework prior to this method being invoked, that is, all data is in connector terminology.

How you Implement doLookup

To implement this method, use the object reference and attribute names provided, and return a `javax.naming.directory.Attributes` object containing the values for any of the named attributes which have values. If an attribute has no value, then do not include it in the returned attributes. Otherwise, the Java CS framework throws an exception as such attributes are known to upset the ApacheDS framework over which the JCS is built.

Note: This method on your connector is called by the ApacheDS SchemaService to sanity test MODIFY and other operations. Therefore, it should be implemented as one of the first operations for your connector, before everything except, perhaps, the ADD operation. Also, if you an operation makes it to the expected method call on `PartitionLoaderService` (for example, `modify()`), but it does not make it to the corresponding call on `MetaConnector`, (for example, `modify()`), then it is worth putting a breakpoint in your connector's attribute-style processor's `doLookup()` method to see if a problem is occurring here. Or if `doLookup` is not being executed, in `MetaConnector.lookup()` or `MetaConnector.search()`.

How you Implement doSearch

You should be able to search and retrieve the corresponding objects on an endpoint system, provided with the search information details.

1. Distinguish the search object type

If multiple object types can be searched and your connector returns true from `isBehaviourSearchSingleClass()`, you can distinguish which type is being requested by examining `filterInfo.getObjectClassMapping().getConnectorClassName()` which will yield the connectorMapTo defined in the metadata for this object. Where your connector does not return true from `isBehaviourSearchSingleClass()`, the list of object classes matching the provided search filter can instead be accessed using `filterInfo.getObjectClassMappings()`.

2. Determine the scope of the search

The scope on search controls can be OBJECT, ONE-LEVEL or SUBTREE with appropriate logic for the types of objects that need to be returned. For flat connectors (like SDK/JDBC) this logic is taken care of for you, but needs to be addressed for hierarchical connectors like JNDI-based ones.

3. Use the search filter

Your connector should support search filtering as supported by the endpoint system API to improve performance. You can get the `ExprNode` by calling `filterInfo.getMappedFilter()` or convert it to a string representation (in LDAP filter syntax, for example `(!(name=f*)(memberOf=*))`) using `filterInfo.getMappedFilterString()`.

If required, the mappedFilter can be converted to a different syntax or in-memory representation using a class deriving from `com.ca.jcs.filter.SimpleFilterVisitor` (or perhaps `org.apache.directory.shared.ldap.filter.FilterVisitor`), which you would then invoke using `filterInfo.getMappedFilter().accept(myFilterVisitor)`.

Note: For an example, see `LDAPFilterToFileSearchVisitor` in the SDK sample.

Note: The `com.ca.jcs.filter.SimpleLDAPFilterToMapVisitor` provides a useful utility visitor for connectors that only implement basic filter semantics. Such connectors probably also benefit from using the `isConnectorFilterable` Boolean metadata setting on any attributes that can appear in customer-provided filters but which the connector itself can not process the filtering for. When it is used, the Java CS framework will automatically reevaluate search results returned by the connector against the provided filter, and throw away any that are found to not to match it before they are returned to the client. This post-checking has no negative impact on search result streaming (if implemented by the connector), but does incur a slight performance impact.

4. Retrieve the return attributes

If possible given the API of the endpoint system, it is best to return only the attributes named in `searchControls.getReturningAttributes()` to maximize performance. Note that the default behavior for the Java CS is to *not* return attributes which have been flagged with the `isExpensive` metadata setting (like photos or associations) in one-level and subtree searches unless they are explicitly requested in the return attributes, and in particular not when a search specifies a null return attributes array (which usually is interpreted to mean *all attributes*). This can be controlled using the `ConnectorConfig.setSearchExpensiveAttrs(boolean)` method for your connector's configuration, usually set using your connector's `conf/connector.xml` file.

5. Base classes for NamingEnumeration objects returned from search operations can be found in the `com.ca.jcs.enumeration` package. In particular, `RawNamingEnumeration` takes care of handling size and time limits for its derived classes and `AppendingNamingEnumeration` can be used to wrap a prepared collection of results or multiple sub-enumerations are stepped through in order.
6. Streaming search support

The ApacheDS and Java CS frameworks support a streaming search mechanism, which is marginally harder to implement but has very considerable scalability advantages (that is, search results are passed back to the client as soon as they become available and peak memory usage during searches is greatly reduced). To make use of streaming search, you need to implement your own `NamingEnumeration` which processes and returns each `SearchResult` object one at a time, and return this `NamingEnumeration` from `doSearch()`, rather than caching all search results in memory prior to returning. The Java CS then processes and passes back search object one by one after the `doSearch()` call has already finished executing, rather than waiting for all results to become available before any are passed back to the client application.

`SDKAttributeStyleOpProcessor` conditionally uses `com.ca.jcs.sdk.SDKSearchEnumeration` when streaming is enabled (configured by setting `eTDYNDirectory.eTDYN-str-multi-ca-01=1`, because it has `connectorMapTo=isStreaming` in `sdkdyn_metadata.xml`).

Note: For an example, see `com.ca.jcs.sdk.SDKSearchEnumeration`.

In some cases, it may make sense for your search method to support both normal search and streaming search mechanisms, in which case the `ConnectorConfig.getStreamingQueryThreshold()` threshold number which is configurable by `connector.xml` file may provide a useful comparison point. In particular, if the number of objects is bigger than the threshold number, you could use a streaming search resulting in higher scalability, otherwise you could use the non-streaming search for possibly better runtime performance. Use of this threshold approach assumes there is a way to efficiently determine the number of results a search may possibly return or did actually return, or somehow tracking the total number of objects of a type that may exist based on a rough calculation during `connector activate()` and keeping running totals..

The streaming search may make debugging and connection management more difficult, because the actual querying takes place after the `doSearch()` method has returned. If you plan to support both modes, it is recommended that you get the non-streaming implementation working first. When debugging search problems, the following setting in `jcs-home/conf/log4j.properties` may be useful:

```
# when above setting is active, comment out if you want every search result
logged (lots of output)
```

```
log4j.Logger.org.apache.directory.server.ldap.support.SearchHandler.logEveryResult=ERROR
```

Note: If you need to implement your own enumeration, then see the Java CS Javadoc for these classes of interest:

- `com.ca.jcs.enumeration.ProcessingNamingEnumeration`
- `com.ca.jcs.enumeration.AbsoluteQueryResultNamingEnumeration`
- `com.ca.jcs.meta.MapSearchResultsFromConnectorEnum`—This calls your connector's `MetaConnector.convertAttributesFromConnector()`

How you Test the Search Operation

After implementing the `doSearch` method, you should be able to load your connector and directory object correctly. To test the `doSearch` function, you also have to implement the `doAdd` function first so you can add some objects within the Provisioning Manager.

Assuming there are some account and group objects in the system, you can do the following searches for testing:

1. Test searching all type of object types.

Issue a one-level search under the connector (or appropriate virtual containers where your connector makes use of them), by providing filters of the form (*objectclass=<class1>*) for each object type. One level searches are particularly important as the Provisioning Server favors their use.

2. Test searching objects with filtering, for example using a filter of the form (*namingAttribute=a**).

For unsupported filters your connector should throw `com.ca.jcs.LdapNotSupportedException`.

3. Test search objects with return attributes.

Try to establish a search by specifying some specific attributes to be returned.

4. Test the streaming search.

Turn on the streaming search and start a subtree level search under the connector.

If no user interface is available at the time you do the testing, you can use JXplorer or other LDAP clients to issue the search requests.

Chapter 13: Updating Endpoint Objects

This section contains the following topics:

[Endpoint Object Update](#) (see page 117)

[Updating an Object](#) (see page 117)

[Update Operation Testing](#) (see page 119)

Endpoint Object Update

You update endpoint objects, such as accounts and groups, by using LDAP MODIFY requests which are modeled in JNDI as an array of `ModificationItem` objects, each of which specifies a target attribute and new attribute values. For multi-valued attributes, a specified mode indicates exactly how the target attribute is to be modified. For example, whether new values are to be added, or existing ones deleted.

Updating an Object

Implementing the modify operation involves extending `AbstractAttributeStyleProcessor` (or a class deriving from it) and implementing the following method:

implementing `AttributeStyle`

```
public void doModify(ObjectInfo objInfo,  
                    ModificationItem[] items)  
    throws NamingException
```

Note: For more information, see [com/ca/jcs/processor/OpProcessor.html#doModify\(com.ca.jcs.ObjectInfo,%20javax.naming.directory.ModificationItem\[\]\)](http://com/ca/jcs/processor/OpProcessor.html#doModify(com.ca.jcs.ObjectInfo,%20javax.naming.directory.ModificationItem[])) and the SDK Sample connector for a complete sample implementation.

And for method-style involves implementing:

Implementing `OpBindingsProcessor`

```
void doUpdate(OpBindingType opBinding,  
             ObjectInfo info,  
             Map parameterValues)  
    throws NamingException;
```

Note: See the add operation for descriptions of the missing parameters. For further information about these parameters, see the JCS Javadoc.

ModificationItem[] items

Contains the list of attributes that are to be modified. `getModificationOp()` can be used to indicate what kind of operation should be performed (like ADD, REPLACE, or REMOVE) and `getAttribute()` provides the attribute details.

Implementing `public void doModify(ObjectInfo objInfo, ModificationItem[ ] items)` throws `NamingException`

You need to update the endpoint system to record the object's modification, given a reference to it (`objInfo`) and an array of modification items. If multiple object types can be modified, you can distinguish which type is being requested by examining `objInfo.getObjectClassMapping().getConnectorClassName()`, which yields the `connectorMapTo` defined in the metadata for this object.

Any required validators or converters have been run by the Java CS framework prior to this method being invoked.

The `forceModificationMode` metadata setting can significantly simplify coding of modification logic for multi-valued attributes in many cases by normalizing all modifications to either an explicit assignment to a new set of values (= "REPLACE") or to a set of additions and deletions from the current set of values (= "DELTA"), according to the requirements of the endpoint system with which your connector interacts. For instance the SDK example connector uses this metadata setting on the `eTSDKAccount.eTSDKGroupMembers` attribute so that it is always handed the new set of values to persist, regardless of whether the original modification mode was ADD, REMOVE, or REPLACE and consequently simplifying its code considerably.

Note: See the JCS Javadoc for information on the `MetaDataUtil.MD_FORCE_MOD_MODE` constant.

The provided attributes need to be modified on the endpoint system, for example a JDBC connector would translate this list into the column name and values of an update statement. An endpoint called by a Java API would need to translate these into the appropriate endpoint objects to represent a rename.

As the modification attributes are provided in an array for modification, rather than the List of attributes supplied when adding a new entry the code snippet for splitting the associations from ordinary attributes changes slightly.

```
SplitModificationItems splitItems =
objInfo.getObjectClassMapping().splitAssocModificationItems(items);
if (splitItems != null)
    items = splitItems.nonAssocItems;

if (items.length > 0)
    // handle modifying attributes

if ((splitItems != null) && (splitItems.assocItems != null))
    doModifyAssocs(objInfo, splitItems.assocItems, jt);
```

Update Operation Testing

At this stage you should be able to do the following:

- Modify all managed object types
- Modify associations between managed object types

For your initial development work you may want to drive simple creations from a LDAP client. However, once your add operation is complete you will need to create an automated test scenario that covers the following objectives:

- Modification of all attributes of all object types
- Modification of all possible object associations
- Failure cases where objects can be created because the object doesn't exist, access permissions are insufficient or other appropriate endpoint failures cases

These tests cases should make use of the scripted tests for creating connectors, directories and endpoint objects you developed earlier to construct a working environment.

Chapter 14: Implementing Associations

This section contains the following topics:

[Associations](#) (see page 121)

[AssocAttributeProcessor Methods](#) (see page 121)

[Defining Associations](#) (see page 122)

[Reverse Associations](#) (see page 122)

[Handling DN Conversion](#) (see page 123)

Associations

You may need to create associations between the managed objects in your endpoint system. You'll need to perform some specific actions for the objects being associated after the associative attributes get added, deleted, or modified (or other LDAP operations).

You could implement custom actions in *do* methods, however, Java CS provides a uniform way to handle these situations.

AssocAttributeProcessor Methods

To use the JCS association support, you need to implement `com.ca.jcs.assoc.AssocAttributeOpProcessor` which has the following methods:

- `doDeleteAssocs`
- `doModifyRnAssocs`
- `doModifyAssocs`
- `doLookupAssocs`
- `doMoveAssocs`
- `doSearchAssocs`
- `addAttrAssocs`
- `removeAttrAssocs`

Where your connector makes use of direct associations, the `com.ca.jcs.assoc.DefaultAssocDirectAttributeOpProcessor` is likely to provide a useful base class, and you may well not have to code any association logic at all. Otherwise you will need to implement the `com.ca.jcs.assoc.AssocIndirectAttributeOpProcessor` interface to handle your connector's indirect associations.

Defining Associations

Associations between objects need to be defined in the connector metadata file.

In the SDK sample connector, the Group is associated with the group members attribute in the Account. When a Group is deleted or renamed, the group members attribute in all Account objects which are the member of the group needs to be updated to reflect the changes. Alternatively, the SDK sample could have added the code for updating the group members attribute for the Account objects in `doDelete` and `doModifyRn`. However, it uses the association to provide an example how to take the advantage provided by the framework.

Define the association between Group and group member attribute in Account.

```
<property name="eTSDKGroupMembers">
    ...
    <!-- when no "assocAttr" is specified it means a uni-directional
link from this object
        to another one using primary key (i.e. naming attribute)
-->
    <metadata name="refObjectType">
        <value><strValue>eTSDKGroup</strValue></value>
    </metadata>
</property>
```

Reverse Associations

You need to be aware of the reverse association support offered to connectors which make use of direct associations by the Java CS.

Note: For more information, see [SDK Overview](#) (see page 19).

Handling DN Conversion

An important consideration when implementing associations is the representations that DNs should have in connector terminology. To help you understand how this is configured and implemented the source code for `com.ca.jcs.converter.connector.DNPropertyConverter` (which handles these conversions) is bundled with SDK. It is important you understand the following metadata settings and their impact on the DN conversion process:

- `MetaDataUtil.MD_DN_NAME_ONLY` ("DNNameOnly")
- `MetaDataUtil.MD_DN_TEST_EXISTS` ("DNTestExists")
- `MetaDataUtil.MD_DN_LDAP_OBJECTCLASS` ("DNLdapObjectClass")
- `MetaDataUtil.MD_DN_IS_ABSOLUTE` ("isDNAbsolute")

Chapter 15: Renaming Endpoint Objects

This section contains the following topics:

[How You Rename the Object](#) (see page 125)

[Example: Implementing doModifyRnAssocs](#) (see page 126)

[Example: Calling doModifyRnAssocs\(\) inside doModifyRn\(\)](#) (see page 127)

[Rename Operation Testing](#) (see page 127)

How You Rename the Object

Implementing the optional rename operation (passed in as an LDAP MODIFYRN request) involves extending `AbstractAttributeStyleOpProcessor` and implementing the following method:

```
implementing AttributeStyle
    public void doModifyRn(ObjectInfo objInfo, final Rdn newRdn)
        throws NamingException
```

This operation renames the object identified by `objInfo` to the new name `newRn`. The type of object being renamed is indicated by `objInfo.getObjectClassMapping().getConnectorClassName()`, the old object name is `objInfo.getName()` and the new object name is `newRn`.

You need to modify the appropriate endpoint objects to perform the equivalent of a rename operation. If your endpoint does not support a rename operation, you may be able to simulate the behavior following these steps:

1. Call `doLookup` to retrieve the existing attributes.
2. Call `doAdd` to create a new object with the new name.
3. Call `doDelete` to remove the old object.

If your endpoint does not support renaming objects, your connector should raise an `LdapNotImplementedException`.

Example: Implementing doModifyRnAssocs

The following is an example of implementing doModifyRnAssocs:

```
public void doModifyRnAssocs(final ObjectInfo objInfo, final Rdn newRn, final
Object context) throws NamingException
{
    final Name      connDn = objInfo.getConnectorDn();

    final Name      newName = connDn.getPrefix(connDn.size() - 1);

    newName.add(newRn.toString());

    doAssocUpdateReferencesTo(objInfo, newName, connector);
}
```

This code contains the following formats:

objInfo

Specifies the object to be modified or deleted.

newRnValue

New relative name (RN) for object, "namingAttr=newName".

context

Optional field which can provide additional context for the requested updates, for example, transactional connectors may want the updates of the associative relationships to occur within a larger transaction.

Thecom.ca.jcs.assoc.DefaultAssocDirectAttributeOpProcessor.doAssocUpdateReferencesTo() method constructs the search filter and invokes the doSearch method of this connector's attribute-style processor to retrieve the all objects referencing the target object, which is to be renamed, through associations (for example, Groups referencing an Account which is being renamed). It then constructs a modification item for each and invokes the doModify() method to update the associative attributes on any impacted referencing objects (for example, the member attribute on each Group that references the renamed Account).

Example: Calling doModifyRnAssocs()inside doModifyRn()

The following is an example of calling doModifyRnAssocs()inside doModifyRn():

```
// need to replace every reference to the renamed object, with the new name
if (!objInfo.getObjectClassMapping().getToAssociations().isEmpty())
    doModifyRnAssocs(objInfo, newRn, null);
```

Rename Operation Testing

At this stage you should be able to rename all managed object types. Your test cases should cover renaming all managed object types and testing failure scenarios like renaming non-existent objects or renaming an object to the name of an existing object.

Chapter 16: Moving Endpoint Objects

This section contains the following topics:

[How You Move the Object](#) (see page 130)

[Example: Implementing doMoveAssocs](#) (see page 131)

[Example: Calling doMoveAssocs\(\) inside doMove\(\)](#) (see page 131)

[Move Operation Testing](#) (see page 132)

How You Move the Object

Implementing the optional move operation (passed in as an LDAP MOVE request), involves extending `AbstractAttributeStyleOpProcessor` and implementing the following methods:

implementing `AttributeStyle`

```
public void doMove(ObjectInfo objInfo,  
                   Name newParentName)  
    throws NamingException  
  
public void doMove(ObjectInfo objInfo,  
                   Name newParentName,  
                   Rdn newRdn)  
    throws NamingException
```

Both operations move the object identified by `objInfo` so it now has a new parent, with the second flavor also changing its name at the same time. The type of object being renamed is indicated by `objInfo.getObjectClassMapping().getConnectorClassName()`, the old object DN is `objInfo.getConnectorDN()` and the new object name is constructed using `newParentName` combined with either `newRdn` or the object's existing RDN.

You need to modify the appropriate endpoint objects to perform the equivalent of the move operation. If your endpoint does not support a move operation, you may be able to simulate the behavior by following these steps:

1. Call `doLookup` to retrieve the existing attributes.
2. Call `doAdd` to create a new object under the specified parent (with either the same name or a new name depending on which `doMove()` variant was called).
3. Call `doDelete` to remove the old object.

If your endpoint does not support moving objects, your connector should raise an `LdapNotImplementedException`.

Example: Implementing doMoveAssocs

The following shows an example of implementing doMoveAssocs:

```
public void doMoveAssocs(final ObjectInfo objInfo, final Name newName, final
Object context) throws NamingException
{
    doAssocUpdateReferencesTo(objInfo, newName, null);
}
```

This code contains the following parameters:

objInfo

Specifies the object to be modified or deleted.

newName

Specifies the new full DN for the target object.

context

(Optional) Provides additional context for the requested updates, for example, transactional connectors may want the updates of the associative relationships to occur within a larger transaction.

com.ca.jcs.assoc.DefaultAssocDirectAttributeOpProcessor.doAssocUpdateReferencesTo() constructs the search filter and invokes the doSearch method of this connector's attribute-style processor to retrieve the all objects referencing the target object, which is to be moved, via associations (for example, Groups referencing an Account which is being moved). It then constructs a modification item for each and invokes the doModify() method to update the associative attributes on any impacted referencing objects (for example, the member attribute on each Group that references the moved Account).

Example: Calling doMoveAssocs() inside doMove()

The following is an example of calling doMoveAssocs() inside doMove()

```
// need to replace every reference to the renamed object, with the new name
final Name  newName = ...; // depends on which variant of doMove()
doMoveRnAssocs(objInfo, newName, null);
```

Move Operation Testing

At this stage you should be able to move all managed object types. Your test cases should cover moving all managed object types and testing failure scenarios like moving non-existent objects or moving an object to the name of an existing object.

Chapter 17: Custom Connector Deployment

This section contains the following topics:

[Java CS Installer Connector Deployment](#) (see page 133)

[Example Deployment](#) (see page 134)

[Deploying with Ant](#) (see page 135)

Java CS Installer Connector Deployment

You can use the Java CS installer to deploy connectors created with the Java CS SDK without writing code to perform the installation.

Note: After you deploy a connector, be sure to acquire the endpoint, then explore and correlate the endpoint. For details, see the *Provisioning Guide*.

Connectors must be distributed in a zip archive format built by running *ant inst* in the top-level *jcs-sdk-home* directory. The zip file name must match the format where * is a short name for the connector of 3 to 5 characters:

jcs-connector-*.zip

The short name should match the short name given to the implementation .jar containing the new connector, for example:

lib/jcs-connector-abc.jar

Important! The zip file contents are extracted directly into the installation folder during the install process, and we recommend that you pay careful attention to the directory used when files are included in the zip file.

For an example of the required approach, see the *-inst* target of *jcs-sdk-home/connectors/sdk/build.xml*.

The zip file can contain the following folders:

- **lib**—(Mandatory) Contains connector jar file *jcs-connector-*.jar* and dependencies
- **conf**—Contains static configuration files
Note: We recommend that your connector provides a `conf/override/<connector>/SAMPLE.connector.xml` file outside of the connector .jar, to allow easy customization by customers as required.
- **extlib**—Contains third party JAR files such as JDBC or JNDI drivers which are independent of the connector's implementation.
- **doc**—Contains documentation files
Note: Only JAR files from /lib and /extlib are added to the class path.

The Java CS installer can deploy both static and dynamic connectors. The installer reads the `connector.xml` from the *jcs-connector-*.jar* to determine the RDN of the connector being installed. This allows custom-created static endpoint types to also be registered with the server at install time with no coding effort.

In this release the installer can now read `_uninst/*.pop.ldif` files included in the .jar files for DYN based connectors, meaning that their target endpoint types are fully functional immediately after installation (note that Connector Xpress does the equivalent job of populating endpoint type details and account template containers in the Provisioning Server for endpoint types created using it).

Note: For more information, see *jcs-sdk-home/connectors/sdkdyn* and *sdksript*.

You can use the upgrade or repair mechanism in the installer to deploy a new connector to an existing Java CS.

Example Deployment

The following example uses *abc* as the short connector name.

To deploy a connector using the Java CS installer

1. Use *ant inst* to create the zip file *jcs-connector-abc.zip* such that *jcs-connector-abc.jar* and any dependent jars are in the /lib folder, deployed documentation files are the /doc folder, and static configuration files in the /conf folder. For example:
 - /lib/jcs-connector-abc.jar
 - /conf/abc.xml
 - /doc/abc_readme.html

2. Copy `jcs-connector-abc.zip` to the same folder as the Java CS installers setup program.
3. Launch and run the installer as normal.
4. Verify that the new connector has been installed as expected.
5. Verify that the new connector can be used before deploying it outside of a test environment.

Deploying with Ant

The Apache Ant tool automatically deploys the `.jar` file and supporting libraries for connectors to the SDK's *jcs-home*, allowing for testing against a development JCS prior to deploying using a `.zip` to a more formal JCS, as described above.

Note: The Java CS SDK requires Apache Ant 1.6.5+.

Appendix A: Testing with JMeter

This section contains the following topics:

[JMeter](#) (see page 137)

[Execute JMeter Test Cases Interactively](#) (see page 137)

[Test Case Contents](#) (see page 138)

[Run a JMeter Test Case](#) (see page 140)

[Editing Test Files](#) (see page 141)

JMeter

Apache JMeter is an Open Source Java component testing application designed to load test functional behavior and measure performance. It has both a desktop user interface, and an interface that can be invoked from Ant for bulk testing. The JMeter application provides a workable framework for the Java CS component testing. We strongly encourage you to use either JMeter or an equivalent application supporting LDAP testing so that component tests can be performed concurrently with connector development.

Note: See the [Apache JMeter](#) website for complete documentation.

Execute JMeter Test Cases Interactively

To execute JMeter test cases interactively

1. Run the following Ant task.

```
ant jmeter.core.init
```

This replaces @VAR@ sequences with real paths while copying test files from *jcs-sdk-home/connectors/*/test/* to *<jcs-sdk-home>/build/tests/*, along with any supporting XML files required for each test.

2. Run the following command to start the JMeter application:
jcs-sdk-home/thirdparty/jakarta-jmeter-/bin/jmeter.bat*
3. In JMeter, select File, Open and open a test case from the following directory:
jcs-sdk-home/build/testcases

Note: JMeter tests can be executed as a batch process with a bundled Ant task which produces summary webpage results in *jcs-sdk-home/build/jmeter/index.html*, for example:

```
cd jcs-sdk-home
```

```
ant jmeter.core
```

A common sequence is to do such a bulk test run, and then investigate any failing testcases individually using the GUI, usually running against a JCS that is running inside your IDE's debugger so you trace exactly what the code is doing.

Note: For more information, see *jcs-sdk-home/build.xml*. For more information on JMeter testing, see *jcs-sdk-home/README.txt*

Test Case Contents

Each bundled test case consists of a top-level test plan and a thread group both sharing the name of the test case's file (like *sdk_core_basic*). When saving a file, click on the top-level node so that the save dialog assumes the right default file name. The basic tests use a single thread group with a single thread so that the steps of the test are executed sequentially and rigorously checked for validity using response assertions. (Load tests that are run after basic behavior has been validated may choose to spawn multiple threads.)

Next come the View Results Tree and View Results in Table listeners that record output from each test step along with timings. These are followed by an Include Controller, a standard JMeter controller which allows one .jmx file to import another. In this case, the *jcs_global_vars.jmx* file is imported which is used to pass on a number of variable assignments from the top-level *build.xml* to be used by all component tests (which port to use / the absolute *\${DIST}* path etc). Once values are known for these variables the next node (a BIND Ldap Extended Request) can then bind to the Java CS.

Tests can include one or more User Defined Variables, Config Elements which allow for variables to be assigned to literal values, where these values can then be referenced using JMeter's *\${var}* syntax. This allows for swapping to a different value.

Tests for dynamic connectors (like JDBC) can make use of a Variable From File Controller, one of the CA custom extensions to JMeter, to read metadata files. These controllers allow the contents of a file to be read into a variable reference, after which it can be referenced using JMeter *\${var}* syntax. These controllers are used widely in the component tests to read in data model and opbindings metadata from local files, making it possible to both share them between multiple tests and to edit them more easily.

The variables are assigned by these controllers in a pre-pass before any test steps are executed, so each variable can only be assigned one value in each .jmx file (that is, use a different variable name for each Variable From File Controller).

Nodes in the body of the tests after the bind are grouped in Recording Controllers or If Controllers for visual clarity. *If Controllers* make it easy to skip the block of test steps nested within them (for instance to skip a nodes deleting all objects after the test case has run, to allow some post-mortem examination of their exact state). The first nodes normally create the parent endpoint type (where the metadata is one of the attribute values that need to be provided for non-static connectors) and the connector. These steps (like almost all following ones) have Response Assertions specified for them, which instruct JMeter how to verify that each step has generated the output expected by it.

Note: Nodes and assertions can be temporarily enabled and disabled in the JMeter GUI via the right-click menu, which can be a very useful technique for zeroing in on one particular problem in a complex test (by reducing irrelevant operations while you are debugging).

The simplest response assertions simply check that the Text Response contains:

```
<responsecode>0</responsecode><responsemessage>Success</responsemessage>
```

This means that the operation was successful. All Java CS related response assertions should check Text Response, however be sure to check the Ignore Status box in cases where you are checking the result of a test step which you expect to fail. The Contains and Matches Pattern Matching Rules look for regular expressions in the Text Response, where '.' matches any character and special characters like '*' and '(' can be quoted with '\' to be treated as literals (they differ in that Matches means that the whole string must match the provided regular expression, rather than just containing it. The Not modifier inverts the check to *not Contains* or *not Matches*.

The Equals test is a CA custom extension to JMeter and allows you to check the entire Text Response text against a provided string literal, without having to quote any regular expression special characters. This means these values can be easily cut and paste from the nodes in the View Results Tree display to their corresponding assertion after visual verification that they are as complete, after which they guarantee the response value will not vary or the component test will fail.

Use of Equals makes it easier to write comprehensive test plans quickly, but the flip side of this is that the cause of a failure may be harder to determine. Hence it is usually best to gain confidence of correct behavior using more specific test cases earlier in your test plan, and later use coarser tests such as an Equals to check on a whole subtree search result.

Tests are then made up of sequences of test step / response assertion pairs.

Run a JMeter Test Case

To run a JMeter test case

1. Ensure that your Java CS is running and that the endpoint referenced by the component test is functional. The core JDBC and SDK tests (run by *ant jmeter.core*) need no external endpoints, and the endpoint for the JNDI related tests can be started externally using either *ant - jcs.test.start.apacheds.lda* or by running:
jcs-sdk-home/build/dist/bin/apacheds_plain.bat
2. In JMeter, click on the node entitled View Results Tree and then select Run, Start from the top menu-bar. Each test step is listed (in red if it failed), and the output for each can be retrieved by clicking.

When Equals tests on large test responses fail (for example, comparing the entire results of a wide-ranging search request) the best technique for investigation is to cut-and-paste the expected and received lines of text to two lines of a temporary file. Then, working backwards from the end, insert spaces in the shorter string until doing so causes later text to re-align with the longer string. In this way, you can identify the sections where the two strings differ and verify that there is a logical reason for the differences. If no such reason can be found then debug the connector until the strings are in agreement again. Otherwise, cut-and-paste the response from the "View Results Tree" to the matching response assertion to record that this is now the correct expected behavior.

There are CA extensions to JMeter Ldap Extended Request Sampler to sort search results and the order of attributes within them, to support such a stable textual comparison.

Stopping or restarting a test before it completes triggers different behavior on the next run due to created objects not being deleted at the end of the previous run. Therefore, a final verification of response assertions for a test should only be performed after the directly previous attempt has run to completion and so that all created objects have been cleaned up.

Editing Test Files

When editing test files, we recommend that you save them outside of *jcs-sdk-home* /build/ (even though you opened them under *jcs-sdk-home*>/build/tests/) so that you do not accidentally lose your work after doing an ant clean. Once you have completed your edits and verified everything is working as intended, you need to:

1. Click the `jcs_global_vars` Include controller near the start of the test and change the Filename field to `@TESTS@/jcs_global_vars.jmx`.
2. Save the test to the path *jcs-sdk-home* /connectors/<myconnector>/test/ (or a subdirectory of it).

Note: After saving, we recommend that you commit your changes to revision control and then update a different working directory with them and rerun *ant jmeter* in this other working directory. This helps catch the case where your actual absolute working directory sneaks into a .jmx file, rather than the `@TESTS@` reference to which is substituted to a real directory by `ant jmeter.core.init`.

Note: See *jcs-sdk-home*/README.txt for more information.

Appendix B: Frequently Asked Questions

This section contains the following topics:

[Design Questions](#) (see page 143)

[Implementation Questions](#) (see page 146)

Design Questions

What is the difference between Metadata and Property tags in the Data Model metadata.xml file?

The properties and classes describe the actual data model and closely relate to the LDAP schema (for example, objectclass `eTDYNAccount` has a string attribute called `eTDYNAccountName`). These can then have metadata property settings specified that spell out specific details regarding their behavior affecting various sections of the architecture (for example, *eTDYNAccount* maps to database table *accounts*, and *eTDYNAccountName* is its naming attribute).

Some metadata settings are only relevant to specific layers of the architecture (for example, `connectorMapTo=` is only important to the Java CS) whereas the data model itself is relevant to all layers.

Is there any documentation on the connector.xml file?

This file is turned into a `com.ca.jcs.ImplBundle` JavaBean by the Spring XML Framework. The best reference is the Java CS Javadoc for this class and all the child classes it references. Look for the class names passed as arguments to `<bean class=` within the `connector.xml` file.

The `connector.xml` for the SDK sample connector also contains some instructional comments. One of the most important parts of the constructed JavaBean is the configuration JavaBean for the connector itself, which is an instance of a class extending `com.ca.jcs.meta.MetaConnectorConfig`. If you are writing a connector which requires some extra custom configuration settings not offered by the `MetaConnectorConfig` base class, then write your own class to extend it and configure it automatically using Spring XML by changing the class name specified to the matching `<bean class=` construct in `connector.xml`, and add extra XML to set the values of the fields you added to your JavaBean.

What is connectorTypeClass?

This field specifies the ConnectorType class for storing data model and opbindings XML metadata documents, as well as to act as the parental container for all connector instances which are driven them. Connector implementations will rarely (if ever) need to change this setting from `com.ca.jcs.meta.MetaConnectorType`. An instance of this bean handles LDAP requests targeting the Namespace level of the Admin DIT, referred to as the ConnectorType level of the DIT by the Java CS.

What is the difference between server_jcs.xml and connector.xml?

These files share some similar content (like validator and converter plug-in configuration) and both are read and converted into JavaBean instances by Spring XML. However `server_jcs.xml` deals with global configuration for all ApacheDS and Java CS settings across the whole server (including plug-ins which have global scope), whereas `connector.xml` deals with configuration for each specific connector implementation (and configured plug-ins are visible to it alone).

Is there any documentation on attributes in metadata.xml file?

These are covered in the Java CS Javadoc for `com.ca.jcs.meta.MetadataUtil` for the Java CS and `com.ca.commons.datamodel.DataModelUtil` for JIAM, where constants are defined for each standardized setting. You may also add extra per-connector metadata settings (after you've checked the existing standard attributes). In this case, create your own class which defines constants for each of the settings.

Note: For more information, see the example in the SDK sample connector `com.ca.jcs.sdk.MetadataConsts` class.

Is there any documentation on attribute connectorMapTo?

See the Java CS Javadoc for `com.ca.jcs.meta.MetadataUtil` and look for the matching constant `MD_CONN_MAP_TO`. Pay attention to the related settings `MD_CONN_MAP_ALIAS` and `MD_CONN_MAP_TO_AMBIGUOUS`.

Which attributes do I need to add after translation parser table into metadata? Only the *connectorMapTo* attribute, or are others also required?

You are very likely to need additional settings from the Java CS Javadoc for `com.ca.jcs.meta.MetadataUtil` for the Java CS and `com.ca.commons.datamodel.DataModelUtil` for JIAM.

Note: For more information, see the preceding question.

My connector has an LDAP objectclass or attribute names which potentially map to multiple connector names. What should I do?

See `com.ca.jcs.meta.MetadataUtil.MD_CONN_MAP_TO_AMBIGUOUS` and `MD_CONN_MAP_TO_AMBIGUOUS_CHOICE_ATTR` which can be used in this case to specify such a relationship. The Java CS framework then takes care of the required handling, but there can be a considerable performance penalty when trying to determine the connector DN for a provided LDAP DN, especially where there are multiple levels of containers.

How does my connector deal with case sensitivity in the endpoint system?

Use the `ConnectorConfig.setCaseSensitive()` method which is usually configured using `conf/connector.xml` (see the SDK sample connector).

How can I validate data passed to and from my connector?

The settings `validateToConnector` and `validateFromConnector` in your `conf/connector.xml` control whether all registered validators (in `conf/server_jcs.xml` / `conf/connector.xml`) triggered by available metadata should be executed. You should never set `validateToConnector=false` outside of development because this turns off all validation of LDAP information being passed to your connector.

`validateFromConnector` defaults to false, but can be set to true if you suspect bad data either pre-exists or is being written to the endpoint system by some other interface, in which validation is performed on query results before they are returned to the client.

How do I write and register a custom validator/converter plug-in?

The [Plug-In Classes](#) (see page 37), and the SDK sample connector code have examples of both.

Why document implementing operations? Are they not described in metadata?

In implementing attribute-style processing driven by the data model metadata, there is still a need to write code to interface with the endpoint system. Where the endpoint system supports method-style processing (like JDBC stored procedures), you can write this code in a language other than Java and use opbindings metadata to instruct the Java CS how to call it. You can also use opbindings and write this code in a scripting language like JavaScript.

Note: For more information, see [Writing Scripts](#) (see page 95).

Implementation Questions

Why is my custom connector implementation not found?

Consider the following:

1. Does the implementationBundle metadata setting on your connector's metadata match the <property name="name"> value in connector.xml? It must, otherwise the Java CS will not know which ImplBundle to use to create an appropriate connector instance, and there will be explanatory log messages in jcs_daily.log.
2. Is the Java CS noticing your connector implementation exists?
 - a. A summary of all connectors is logged to logs/jcs_daily.log at start-up at INFO log-level, for example:

```
INFO    - loaded 10 connectors:
```

```
loaded connector "AS400" [connectorTypeName='OS400',  
connectorTypeLdapObjClass=eTAS4Namespace]
```

```
loaded connector "JDBC" [connectorTypeName='null',  
connectorTypeLdapObjClass=eTDYNNamespace]
```

```
loaded connector "JNDI" [connectorTypeName='null',  
connectorTypeLdapObjClass=eTDYNNamespace]
```

```
loaded connector "KRB" [connectorTypeName='KRB Namespace',  
connectorTypeLdapObjClass=eTKRBNamespace]
```

```
loaded connector "ORA" [connectorTypeName='Oracle Server',  
connectorTypeLdapObjClass=eTORANamespace]
```

```
loaded connector "SAP" [connectorTypeName='SAP R3',  
connectorTypeLdapObjClass=eTSAPNamespace]
```

```
loaded connector "SDK" [connectorTypeName='SDK Namespace',  
connectorTypeLdapObjClass=null]
```

```
loaded connector "SDKDYN" [connectorTypeName='SDK DYN Namespace',  
connectorTypeLdapObjClass=null]
```

```
loaded connector "SDKSCRIPT" [connectorTypeName='SDK Script DYN  
Namespace', connectorTypeLdapObjClass=null]
```

- b. Also logged at the INFO level is a summary of the information read from schemas contributed by static connectors. For example:

```
INFO    - '/conf/eta_sql_openldap.schema': registered 9 objectClasses  
(skipped 0)
```

Errors encountered processing schemas are also logged, and can be a reason that your connector implementation is not being found by the Java CS.

- c. If your connector does not appear in this list and you are running within a Java IDE like Eclipse or IDEA, then you need to add your connector's jar file to the Java CS's classpath.
- d. If your connector does not appear in this list and you are running from the command line (`jcs.bat` or `jcs.sh`) then your connector jar is probably malformed or has not been copied to `jcs-home/lib/`, make sure it has a valid `/conf/connector.xml` file within it (refer to the structure of the SDK connector's jar file).

Why does the Java CS appear to execute without triggering breakpoints in the debugger?

In some circumstances the Java CS has trouble shutting down and there have been observations of a phantom Java CS running in the background which is servicing LDAP requests but to which the debugger is not connected. You may need to run the task manager and manually shut down the phantom Java CS `java.exe` process.

Why are exception breakpoints I set in the Java CS not being triggered?

Setting exception breakpoints in your IDE is complicated by the Java CS use of Java proxies in its implementation. In particular, if you observe an exception but then find a matching exception breakpoint isn't triggered as you'd expect, try setting a breakpoint on `InvocationTargetException` which may be wrapping the original exception,

Why does the debugger step into JDK code if I trace into the end of a call on `MetaConnector`, to `search()` for example?

The Java CS framework (and some connectors themselves) uses Java proxies. You may be stepping into the call on the proxy method. Try inserting a breakpoint in the related method of the target class (like `JDBCAttributeStyleOpProcessor.doSearch()`) to skip through the proxy code.

How does my code access custom metadata settings I have added?

The Java CS uses JAXB generated code to convert the metadata files into `JavaBean` instances, which are then wrapped in instances of classes from the `com.ca.commons.datamodel` packages (like `DataModelClass` / `DataModelProperty`). These are then cached inside instances of the `com.ca.jcs.meta.MetaObjectClassMapping` for efficient runtime access using the Java CS. To get at your extra custom metadata settings, you need a reference to the parent `DataModelProperty` (say using `MetaObjectClassMapping.getDataModelProperty(String)`), and then look up its metadata settings using the `getMetadataProperty(String)` method.

Note: For an example, see the `com.ca.jcs.sdk.converter.DummyFlattenPropertyConverter` class reference `MetadataConsts.MD_FLATTEN_SEPARATOR` in the SDK sample connector.

Why does the Java CS silently hang when performing a search?

ApacheDS worker threads occasionally hang in two circumstances, outlined below. Please report an issue with CA technical support if you observe this behavior.

When attributes are passed from `doLookup()` or `doSearch()` with no values, this upsets the ApacheDS LDAP codec. In this case the attribute names shouldn't be included at all, rather than being added with no values. To guard against this occurrence, the Java CS framework checks attributes passed by your connector using the `com.ca.jcs.LdapUtil.checkAttrsValid()` method, so a descriptive assertion failure will single out the errant attribute.

The Java CS code may need to parse internally generated LDAP filter expressions (like determining which objects have associations with an existing object that is being deleted, renamed, or returned in search results etc) using `org.apache.directory.shared.ldap.filter.FilterParserImpl`. In the presence of various special characters, there have been some cases where these expressions are not well formed, in which case the ApacheDS filter parsing code doesn't fail gracefully but instead outputs an error message to stdout (and may therefore not be noticed) and throws an exception which causes the thread executing the current LDAP request to never return to the client.

How can I customize the behavior of the connector level of the DIT, for instance to calculate the values of some virtual attributes?

You will need to implement a class extending `com.ca.jcs.processor.ConnectorAttributesProcessor` and register it by calling the `setConnectorAttributesProcessor()` method in your connector's constructor.

How can I insert the values of virtual attributes in search results returned by my connector?

Usually it will be sufficient to override the `convertAttributesFromConnector()` method in your connector (make sure you call `super.convertAttributesFromConnector()` !) to handle whatever logic is required, as this method is called by `com.ca.jcs.meta.MapSearchResultsFromConnectorEnum` for each search result to map them to LDAP. It is recommended that you add your virtual attributes first and then call `super.convertAttributesFromConnector()` afterwards so that you can keep your logic free from needing to reference LDAP attribute names.

How can I determine which objectclass instance my code has been passed when my connectorMapTo settings are long involved expressions (like a complicated SELECT statement in SQL)? I don't want my code to refer to LDAP objectclass names.

You should use the metadata setting `com.ca.jcs.meta.MetadataUtil.MD_CONN_MAP_ALIAS` to specify a short alias for your objectclass (say `connectorMapToAlias=account`) and use this as the discriminator in your code. CA recommends that you use your own utility method to lookup the value of this metadata setting where required (see previous question for tips).

What does an exception mean of this form mean: "ERROR - ... LdapInvalidAttributeIdentifierException: eTLNDDeleteOldReplicas not found in attribute registry!"?

The metadata document for your connector refers to an attribute not known in the following:

- Your connector's LDA schema (in this case, `connectors/Ind/conf/eta_Ind_openldap.schema`), registered using `conf/connector.xml`
- Any other global schema files loaded by the Java CS as driven by `conf/server_jcs.xml`.

This means that either an incorrect attribute name has been referenced or that you need to add the attribute name to the appropriate `.schema` file.

Appendix C: Debugging Tips

The following are suggested breakpoint locations for debugging custom connectors. These breakpoints are listed in order, starting with those nearest your custom connector to those deepest within the ApacheDS runtime stack:

- Methods in one of your connector's processing styles (for example, `SDKAttributeStyleOpProcessor.doAdd()`) which are called after the framework has performed all validations and conversions as specified by your configuration metadata. You can check arguments here in the debugger before they are used by your connector-specific code.
- LDAP methods within `MetaConnector`, for instance `MetaConnector.add()`. Although the arguments to these methods have been normalized, they are still in LDAP terminology and may provide clues if you are having problems during the name mapping, validation, and conversion phases.
- Be aware that the ApacheDS `SchemaService` does `lookup()` calls on your connector to sanity test `MODIFY` and other operations. Therefore, if you see an operation make it to the expected method call on `PartitionLoaderService` (say `modify()`), but it doesn't make it to the corresponding call on `MetaConnector` (say `modify()`) then it is worth putting a breakpoint in your connector's attribute-style processor's `doLookup()` method to see if a problem is occurring here (or `MetaConnector.lookup()` / `MetaConnector.search()` if execution is not reaching `doLookup()`)

`NamingEnumerations`. These are returned by the `search()` method. Some streaming varieties mean that results are not retrieved at the time that the `search()` method returns, but rather at some later time when the ApacheDS framework steps through the entries in the returned enumeration. When problems occur whilst stepping through search results the most interesting break point candidate is at the start of `MapSearchResultsFromConnectorEnum.processNext()` where you can see each result prior to it being converted from connector terminology to LDAP.

- LDAP methods within the `PartitionLoaderService` (like `add()`), which are called as soon as a new LDAP request is submitted to the ApacheDS interceptor chain. These may be useful to look at requests before they are processed by ApacheDS. This class acts as the front end to the Java CS in that it triggers the activation of new connector instances when DN's are received which access objects within their DIT. If methods in `MetaConnector` are not being called, then this is the next layer down in the JCS architecture.

- `org.apache.directory.server.ldap.LdapProtocolProvider.messageReceived()` . This is the deepest point in the ApacheDS stack handling LDAP requests and should only be useful if requests or responses are failing to be encoded or decoded according to the LDAP protocol for some reason.
- Setting exception breakpoints in your IDE is complicated by the Java CS' use of Java proxies in its implementation. In particular, if you observe an exception but then find a matching exception breakpoint isn't triggered as you'd expect, then try instead setting a breakpoint on `InvocationTargetException` which may be wrapping the original exception.
- When porting a C++ connector to the Java CS, you can compare the objects and attributes found in each. To do this, explore the same endpoint in each connector, and then connect to the ETADB. Be sure to delete everything under the connector level in ETADB and restart your Provisioning Server. Also, remove `systemdb` in the Java CS before starting it.
- During development, using the secret C++ Connector password in both the Provisioning Manager and the Java CS saves time in resetting the password if you need to clean up 'systemdb' on a regular basis. To reset the password manually you must bind to a running Java CS with the secret password and set the `userPassword` on the `uid=admin,ou=system` object, where it will then be saved to `systemdb`.

Appendix D: Connector Review Checklist

The following checklists provides a list of important design and implementation areas you can use to gauge the quality of the coding effort and the corresponding associated cost of maintaining and extending the connector. We recommend that you rate your adherence to the following Java development standards, on a scale of 0 - 10, where 10 is the highest and 0 the lowest.

This will allow you to summarize important aspects of the connector's implementation, which will assist Quality Assurance and future developers to get a quick, high-level, understanding of the connectors' implementation.

Holistic Design Considerations

You should consider the following important aspects of the holistic design and requirements for the connector when implementing your connector:

1. Were you able to use the DYN schema for your connector? If not, summarize the areas where it was deficient.
2. Are you porting an existing C++ connector to Java?
 - a. If so, you need to answer yes to one of the following two questions:
 - Is the Java connector completely deprecating the C++ one (in which case the Provisioning Manager plug-in doesn't need to concern itself with backward compatibility)?
 - Is the associated Provisioning Manager plug-in going to detect whether it is communicating with a C++ or Java connector at the back-end by testing for the presence of the *eTMetaData* attribute on the associated endpoint type?
 - b. Are any changes to existing schema or parser table required to support the new Java connector?
 - If there are, are the changes backward compatible?
 - If there are no changes, has C++ connector been updated to match the changes?
 - c. Have you listed any migration steps (including those due to the preceding step b) required to migrate customers of the C++ connector and confirm where they are documented?
3. What are the expected peak numbers of each object class that your connector manages, with particular attention to the most numerous ones? Has the connector been tested against these peak numbers?
4. Does the connector support rename (MODIFYRN) requests and does the Provisioning Manager UI plug-in expose this functionality?
5. Does the connector support MOVE requests and does the Provisioning Manager UI plug-in expose this functionality?
6. Does the connector support any custom behavioral attributes, for example, an attribute passed in a MODIFY that actually selects a function to be performed (often with reference to other attributes) on the target object, rather than being stored and later retrieved? If it does, detail the object class and attributes grouped by each function that can be performed.
7. Does connector make use of any third-party libraries?
 - If it does, do you have permission to bundle them with the connector?
 - If it does not, have you documented the instructions telling customers where to find them and how to install them (this usually involves copying jars to *jcs-home/lib* and recycling JCS)?

- Does the connector depend on JNI (Java Native Interface) support, directly or indirectly?
- 8. Does the connector depend on any special configuration on the JCS server that you can not work-around using some sort of URL scheme for connection details? Have the details of any environmental pre-conditions regarding third party software installation been documented?
- 9. Does the connector impose any operating system requirements on its host JCS?
- 10. Does the connector need to support a notion of custom attributes, which are mapped to native attributes by the customer after deployment? If so, we recommend that they should be configured through `conf/override/<myconnector>/connector.xml`.
- 11. Does the connector have any compound attributes where a single value for an attribute actually contains multiple pieces of information ? Such attributes tend to be required to because the ordering of LDAP attribute values cannot be guaranteed.
- 12. If there is an existing C++ connector, does it make use of a plug-in to the Provisioning Manager (note this is outside of the plug-in to the CCS for the connector)? Have the details about exactly what the plug-in does been documented?

Java Development Standards Considerations

Consider the following when determining how well your connector implementation has adhered to Java development standards:

1. Have basic Java standards been adhered to?
2. Have constants been used rather than magic numbers and magic strings?
3. Does the Javadoc meet the following quality and coverage considerations?
 - Have class header comments (especially on the core classes) explaining requirements and gotchas been included?
 - Have method comments, especially important for methods that may be confusing, been included?

Note: Some leeway in documenting just one of get or set method for a property is acceptable.

 - Are package.html files on sub-packages provided where required?
4. Have the following logging standards been met?
 - Are appropriate levels used?
 - Has careful attention been paid to logging error messages?
 - Have lower-level severity messages been wrapped in "if (log.isDebugEnabled()) checks for runtime efficiency?
 - When logging exceptions, has (log.debug(msg, ex) been used rather than log.debug(msg + ex))?
5. Are JDK 1.5 generics used where applicable and allowed by your chosen API?
6. Are repeatedly referenced complex expressions remembered in stack variables rather than repeated multiple times (use of basic refactoring in IDE)?
7. Has attention been paid to threading issues (for example, activate and deactivate calls should be synchronized, and such)?
 - Has some testing been performed where multiple threads access the connector concurrently?
8. Has dead, commented out code been cleaned up?

Metadata Use Considerations

As a JCS connector is expected to be a fairly thin adapter between LDAP and the native endpoint system, optimal use of metadata significantly reduces the amount of custom coding required (typically, 80-90% code reduction for static C++ options that have been ported). Consider the following when you rate the degree to which metadata has been used correctly:

1. Has `connectorMapTo` and similar supporting values (possibly with extra connector-specific metadata settings being added), been used to minimize coding?
2. Have you ensured that no LDAP object classes or attributes are referenced in the connector's code, and that `connectorMapTo` or `connectorMapToAlias` values have been used instead?
3. Have optimal choices of data model value types been used?
 - Has the correct value definition for `datamodel` properties been used?
 - Have metadata *enum* definitions been used where appropriate?
 - Have *flexistr* values been used where required?
4. Do all appropriate metadata items on the Connector object class have *isConnection=true*?
 - Have you fully tested changing of connection related attributes?

Connector Coding Considerations

Consider the following when assessing the general coding of the connector's logic:

1. Is the coding and configuration of the connection pool correct?
2. Is the streaming of search results supported? streaming always / threshold between streaming and not / no. If streaming is supported:
 - Is super-streaming of search results supported?
Note: You do not need all ids / primary keys in memory at once.
 - Is streaming used for all object classes, and if it is not, is the subset for which it is used listed?
 - Is a new connection taken from the pool and released for each search result (or a small number of search results), so that many concurrent searches can be active and share available connections fairly? If not, then detail what the connector does in this regard, including the reason that the same connection is used for the whole search (for example, this may be necessary to support super-streaming or because the native API mandates it).
3. Has maximum use been made of JCS framework services and existing connector implementations, or both?
 - Has any reuse that was possible and any specialization that was required been summarized?
 - Have any extra connector-specific metadata settings that were required been summarized?
 - Have any connector-specific validator or converter plug-ins that were written for the connector been summarized?
4. Have Validator and Converter plug-Ins been used to minimize custom coding?
5. To what degree is the conversion LDAP search filters to native filters supported?
 - Minimal—Only expressions like (*objectclass=eTDYNAccount*) and (*eTDYNAccountName=a**) are supported
 - Partial—A richer set of filter syntax and attributes are supported, presumably by morphing the LDAP filter using a FilterVisitor)
 - Complete—Complete filter syntax is supported.
 - If support is not complete, has the `isConnectorFilterable=false` metadata setting been used to flag attributes which the connector is unable to respect in filter assertions?
6. Are complete one-level and subtree search semantics supported so that clients other than the Provisioning Manager can make use of them?

7. Are all attributes that can be returned from a object scope search also supported for one-level and subtree cases (this is highly desirable)?
8. How has exception handling been implemented?
 - Are exceptions not being swallowed, that is, caught and simply ignored?
 - Is `LdapExceptionPrefix` prepended to the message for all exceptions raised in the connector's code?
 - Are native exceptions being carefully mapped into `LdapNamingExceptions` with suitable `ResultCodeEnum` codes, especially:
 - `LdapNameAlreadyBoundException` for ADD requests
 - `LdapNameNotFoundException` for other requests
 - Have retrieable exceptions been distinguished explicitly in code where required?

Note: For more information, see `RetryOpProcessorProxy`.
 - Has resiliency support been properly configured and tested, that is, are retry group messages in `connector.xml` accurate and complete?
9. Are the remaining TODOs small in number and minor in consequence?
10. Has optimal caching of information during `activate()` been implemented, (to improve performance) where applicable?
11. Does the connector depend on any objects with specialized lifecycles? For example, connector sibling objects, objects which clients poll and change asynchronously and such.

Component Test Considerations

We recommend that component tests (for example, JMeter or equivalent software) should grow in strict tandem with the functionality of the connector. A useful maxim is that any connector functionality not covered by automated component tests can not be considered to exist. Even if manual testing proves the connector works today, this may not be adequate when the connector needs to be modified in the future. Without the support of component tests with good coverage, there can be no confidence future changes are safe. Consider the following when designing component tests:

1. Is coverage of component tests adequate?
 - Are all object classes tested for all supported operations?
 - Are examples of attributes with all supported datamodel values present?
 - Are all validator and converter plug-ins relevant for the connector covered by test cases?
 - Are a suitable number of multi-valued attributes tested, including different modes (for example, replace, add, remove) in MODIFY requests?
 - Are all associations tested from all supported directions, for example, group.member and account.memberOf) with a range of containment levels for DNs in direct associations?
 - Have a number of characters special to LDAP been tested in object RDNs to verify correct handling? This is especially important for object classes with DNs that are stored in association attribute values.
 - Have a number of characters special to the connector's chosen API been tested in object RDNs to verify correct handling? This is especially important for object classes with DNs that are stored in association attribute values.
2. Are strict response assertions being used to ensure correct behavior (for example, search after modify to ensure correct change)?
 - Are basic tests split into the smallest units to allow easy tracking from failures back to minimal root causes?
 - Is confidence established in earlier test steps before blunter equals assertions on larger sets of search results appear?
3. Are basic error cases tested (for example, modify, search with base, delete, modify on non-existent object, adding an existing object)?
4. Is there coverage of different supported connection schemes, if multiple flavors of connection are required by the connector?

Index

A

ANT • 48, 139, 141
Apache Directory Server • 15, 150, 155
associations • 125

C

Connector Xpress • 12, 58
connector.xml file • 40, 41, 51, 147
connectors
 defined • 12, 17
 deploying • 137
 implementation • 89, 92
 jar files • 51, 52, 137
 testing • 98
 types • 90
converters • 41, 42, 52, 109, 147

D

deployment • 137

E

endpoints
 associating • 125
 creating • 107
 defined • 12
 deleting • 111
 renaming • 129
 searching • 115
 testing • 109, 114, 119, 123, 131
 updating • 121

F

files
 connector.xml • 39, 41, 51, 147
 jar • 51, 52, 137
 metadata.xml • 52, 59, 147
 server_jcs.xml • 41, 42, 48, 51, 52, 147, 150, 155

I

installation • 19

J

Java Connector Server
 architecture • 13
 defined • 11
 directories • 19
 functionality • 15
 installing • 19
Java Connector Server SDK
 installing • 19
 overview • 21
 packages • 23
 sample • 47, 59, 126
JavaBeans • 39, 51, 52, 147
JMeter • 47, 48, 89, 141

L

LDAP
 attributes • 58, 89
 data conversion • 15, 42
 data validation • 15, 41
 exceptions • 43
 mapping • 17, 90, 92
 operations • 18, 23, 47, 48, 51, 98, 147, 150, 155
 overview • 18

M

metadata • 15, 17, 21, 23, 41, 47, 52, 55, 89, 90, 147
metadata.xml file • 52, 59, 147

P

pooling • 89, 96
Provisioning Server • 12, 13, 18

S

server_jcs.xml file • 41, 42, 48, 51, 52, 147, 150, 155
Spring Framework • 15, 39, 41, 51, 147
SuperAgent • 11, 15, 48, 58

T

testing • 109, 114, 119, 123, 131, 141, 155

V

validators • 41, 52, 107, 147