CA Identity Manager

Programming Guide for Java Connector Server

r12.5 SP1

CA Technologies Product References

This document references the following CA products:

- CA Identity Manager

Contact CA Technologies

Contact Technical Support

For your convenience, CA Technologies provides one site where you can access the information you need for your Home Office, Small Business, and Enterprise CA Technologies products. At http://ca.com/support, you can access the following:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

Provide Feedback

If you have comments or questions about CA Technologies product documentation, you can send a message to techpubs@ca.com.

If you would like to provide feedback about CA Technologies product documentation, complete our short customer survey, which is available on the CA Support website at http://ca.com/docs.

Contents

Chapter 1: 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 (CCS). The Java CS is architecturally and functionally similar to the CCS, except that it is implemented in Java rather than C++. Consequently this allows you to write your connectors in Java. In addition, to the extent to which it is possible the Java CS is data-driven rather than code-driven, which allows the container (i.e. Java CS) to do much of the connector's work for it.

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

Note: For the most current technical information, see the JavaDoc included with the JCS SDK install. It may be slightly more up to date then the JavaDoc integrated with this Guide.

This section contains the following topics:

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. The Java CS is architecturally and functionally similar to the C++ Connector Server, except that it has a Java API instead of a C++ API. The Java API allows your connectors to be implemented in Java. In addition, the Java CS is data-driven rather than code-driven, which allows the container (or Java CS) to address more functionality 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

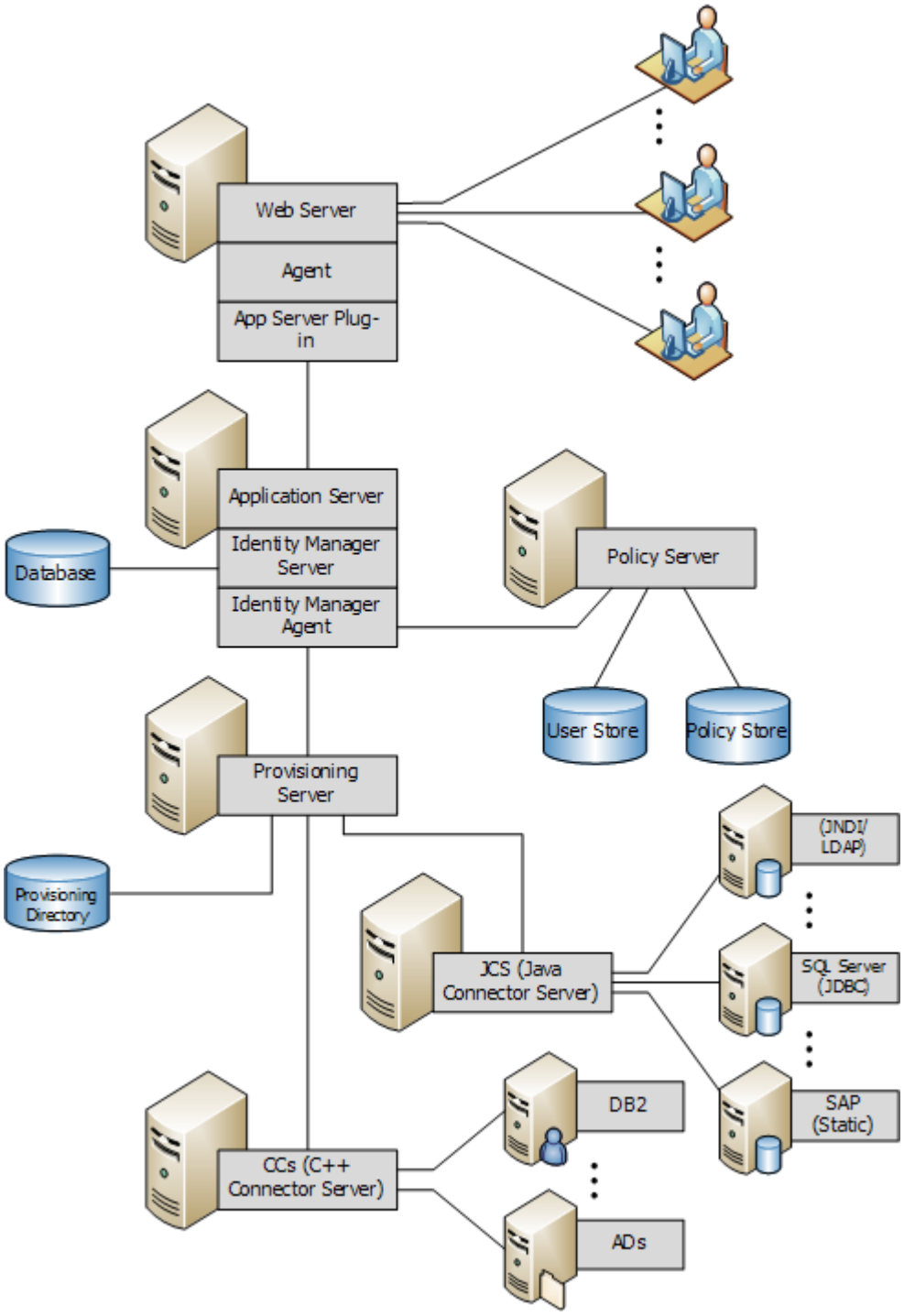
An *endpoint* is a specific target system or application, such as Active Directory or Microsoft Exchange, that the Provisioning Server manages.

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 in a consistent manner. Connectors translate add, modify, delete, rename, and search LDAP operations on those objects into corresponding actions against the endpoint system.

Users use Connector Xpress to generate and maintain XML metadata for JDBC and JNDI dynamic connectors. Developers can also maintain data for other connectors manually, or adjust metadata for released connectors (for instance adding site-specific mappings for custom attributes).

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 web sites:

- 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

The Java CS provides services to the connector implementations deployed to it while forcing standardized behavior 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 Converters 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 103)

Connector Implementation

You implement Connectors using a combination of the following:

- Core connector code targeting an endpoint type technology.
- Validator and converter plug-in code which can be wired into your implementation using XML configuration.
- User maintained XML metadata, managed with Connector Xpress, drives configured plug-ins.
- Reusing existing connector implementations through inheritance, and only coding the minimal specializations required.
- Specializing existing connector behavior by using method-style or script-style opbindings, which are run before, instead of, or after targeted LDAP operations on specified target object classes.

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 DNs, 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. The Partition Loader Service then passes LDAP requests back to the core ApacheDS serviced. As connectors are registered as ApacheDS partitions, they receive requests that fall into their subtree of the ApacheDS DIT.

Java CS can host any number of endpoint types, each of which is associated with a single metadata definition. Each 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)
jcs-sdk-home	C:\Program Files\CA\Identity Manager\Connector Server	/opt/CA/IdentityManager/Connector Server
jcs-home **Note:** Under production JCS, this has the same structure as JCS home	*jcs-sdk-home*\build\dist	*jcs-sdk-home*\build\dist
sample-home	C:\Program Files\CA\Identity Manager\Connector Server SDK\connectors\sdk	/opt/CA/IdentityManager/ConnectorServer SDK/connectors/sdk
im-home	C:\Program Files\CA\Identity Manager	/opt/CA/IdentityManager
imps-home	C:\Program Files\CA\Identity Manager\Provisioning Server	/opt/CA/IdentityManager/ProvisioningServer
conxp-home	C:\Program Files\CA\Identity Manager\Connector Xpress	/opt/CA/IdentityManager/ConnectorXpress

Chapter 2: Java CS SDK

This section contains the following topics:

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 associations between objects
- Writing new converter and validator plug-ins, and arranging to trigger core plug-ins using metadata
- Implementing or enhancing connector logic using JavaScript

Important: 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) override these methods. However, if you do override these methods, it is imperative that you invoke the matching super.*() method. For example, an overridden add() method must call super.add().

Building and Debugging

After installing the Java CS SDK refer to the README.txt file in the *jcs-sdk-home* directory. The file explains how the ANT build harness works, and information about a preconfigured project and launcher for the Eclipse IDE. Typically the best approach for debugging is to use Eclipse, or your alternative IDE.

However, to debug remotely, there are some flags you are required to provide on the "java" command-line used to run the Java CS. You can refer to these flags in the *jcs-sdk-home/bin/jcs.bat* file which can be used as an alternative way to invoke a development Java CS (rather than using your IDE). The flags are "-Xdebug
-Xrunjdwp:server=y,transport=dt_socket,address=8787,suspend=n".

Note: For more information, see http://www.ibm.com/developerworks/opensource/library/os-eclipse-javadebug/index.html which is a useful reference article.

Be aware of the following:

- The example port chosen in this configuration (8787) can be varied as long as the Remote Java Application used to connect to the remote Java CS server is also updated to match.
- The debug support in the JVM uses this port and is independent of the port on which the Java CS can be contacted, which is configured in server_jcs.properties and defaults to 20410 for production Java CS instances.
- If "suspend=y" is used instead of "suspend=n" then the JCS does not proceed with its start-up sequence until a Remote Java Application attaches to it. This is useful if you want to debug the start-up sequence.
- The production Java CS install uses procrun to launch JVM as a Windows service. To configure its command-line arguments use the Regedit operating system utility to change the following key:

  LOCAL_MACHINE\SOFTWARE\ComputerAssociates\Identity Manager\Procrun 2.0\im_jcs\Parameters\Java
- The Options key contains the list of all options passed to the target JVM, add the debug arguments described previously.
- Restart the Java CS service (or development JVM) after you change the command-line arguments.

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:

- SDKDYN connector (see page 21)
- SDK connector (see page 56)
- SDKFS connector (see page 22)
- SDKSCRIPT (see page 22)

- SDKCOMPOUND (see page 22)
- SDK (see page 56) (deprecated)
- JDBC connector (see page 23)
- JNDI connector (see page 23)

The full code and configuration for these connectors and some carefully chosen sample code from the Java CS framework code are also included in the SDK (under *jcs-sdk-home*/src/). The samples help ensure the connector developer's understanding of core classes and best practices.

Note: For more information about the SDKs contents or building, see *index.html* under *jcs-sdk-home* or the readme file.

SDKDYN

The SDKDYN Connector 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

The SDKDYN takes its Java implementation from the SDK connector, and differs only in that it uses the DYN schema instead of the SDK schema. Consequently, it uses 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 the 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.

SDK Connector

The SDK connector 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

SDKFS Connector

The SDKFS connector shares the same Java Implementation as the SDKDYN connector but it is hierarchical, where the containers are represented as file system directories. This means it needs its own distinct metadata document, sdkfs_metadata.xml.

SDKSCRIPT Connector

The SDKSCRIPT connector uses a copy of the metadata used by the SDKDYN connector, but all its logic is implemented in JavaScript instead of Java. The script operation bindings that can be found in /conf/sdkscript_opbindings.xml, which wrap a stub connector as configured through /conf/connector.xml achieve this.

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

SDKCOMPOUND Connector

The SDKCOMPOUND connector takes its Java implementation from the SDKDYN connector, and differs only in that it also demonstrates attributes with *compound values.* That is, each value represents a complex state rather than a single simple value. These compound values are represented using JSON syntax (a subset of JavaScript syntax). Cases where compound values are handy include when you want to store a group of values in each value stored for an attribute. For example, you may want to store an object not managed by your connector "inline", or there may be data that must be stored on an associative link. For example, a permission which is granted between specified start and end times. Each value is represented as a JSON object, for example, {"name": "seqjim", "intV" : 47, "strV": "seqstrval2"} specifies an object which has three fields, "name" (a string), "intV" (an integer) and "strV" (a string).

When the order of values for a multi-valued attribute is important, in which case the attribute should have a single JSON array value, for example, [{"name": "seqjack", "strV" : "seqstrval1"}, {"name": "seqjim", "intV" : 47, "strV": "seqstrval2"}]

JDBC Connector

The JDBC connector is now included in the SDK in compiled form, rather than source code. The distribution includes the connector JAR, a set of test metadata, and JMeter test plan files.

The tests run against the HSQL database and demonstrate various JDBC DYN based mappings and functional items such as:

- Account and group management
- Associations
- Virtual container management and use of multiple objects in a single virtual container as opposed to multiple virtual containers
- UNICODE characters
- Forcing modification operation styles, flattening
- Compound attributes and associations, where contents of a single attribute value are stored in separate tables.
- Use of sequences and identity columns, where attribute values are provided dynamically by the endpoint system rather than being passed in by the client.
- JDBC op-bindings, transactions and scripted op-bindings

JNDI Connector

The JNDI connector is now included in the SDK in compiled form, rather than in source code. The distribution includes the connector JAR, as a set of test metadata, and JMeter test plan files.

JNDI distribution includes a common set of LDAP – inetOrgPerson mappings in resources/jndi_inetorgperson_common_metadata.xml. You can use these mappings as a starting point or template for a full inetOrgPerson JNDI connector.

The tests demonstrate various JNDI DYN-based mappings and items such as the following:

- Account and group management
- Associations including, NIS, and through key attributes.

  For example, posixGroup has a memberUid associative attribute containing values like "uid=123" instead of referencing an accounts name.
- Auxiliary and derived classes
- Search container per objectclass

- Operational attributes
- LDAP—Full inetOrgPerson mappings tests (tested against ApacheDS as the LDAP vendor backend)

Sources for a real world example of a connection factory and connection pool implementation are included in JNDIConnectionFactory.java and JNDIConnectionPool.java.

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, and such. 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 about the packages in the SDK, see the JCS Javadoc.

How you Deploy Java CS Connectors

Ant 1.70+ is now required to build the SDK.

To deploy Java CS connectors, do the following:

1. Place the target connector's .jar file in a Java CS lib/ directory (see page 25). The Java CS can be either:
   - An SDK Java CS using a default port 20412. This port is commonly used during the development and component testing of the connector.
   - A production Java CS using a default port 20410. This port is commonly used after the connector has undergone comprehensive testing and you consider it to be release quality.
2. For C++ connectors, configure the Provisioning server using POP scripts.

   The POP script creates the top level of their managed namespace in the Provisioning Server.
3. For Java connectors, configure the Provisioning Server.

   Configuring the Provisioning Server tells the Provisioning Server about the connector type and its associated metadata.
4. Restart the Java CS.

   The Java CS registers the connector.

Build your Custom Connector Using the ANT Build Harness

To build a .zip file for your custom connector, use the ANT Build Harness.

To build your custom connector using the ant build harness

1. Copy your connector's *jcs-sdk-home*/build/inst/jcs-connector-*.zip file to a directory into which the Java CS installer has been unzipped
2. Run the installer to install your custom connector to a production Java CS.
3. Restart the Java CS.

Place the Target Connector's .jar file in a JCS lib/ Directory

When deploying Java CS connectors, place the target connector's .jar file in a Java CS lib/ directory. The method you use depends on your environment:

1. In a testing or debugging environment, do the following:
   a. Run Ant inst.

b. Copy a jcs-sdk-home /lib/jcs-connector-*.jar file (and any third-party .jar files on which it depends) to a production jcs-home lib/ (and dependant .jars to jcs-home extlib/)

2. To install a released connector, do any of the following.
   - Use the ANT build harness provided with the Java CS SDK to build your custom connector, hosted by the Java CS SDK on default port 20412. (see page 25)
   - Use ant inst with the Java CS SDK to build a .zip file for your custom connector, then use the Java CS installer.
3. Restart the Java CS.

Configure the Provisioning Server

To tell the Provisioning Server about the connector type and its associated metadata, configure the Provisioning Server.

To configure the Provisioning Server

1. Do *one* of the following:
   - Use the Java CS installer to register released connectors and custom connectors.
   - Use one of the Connector Xpress templates bundled for the released connectors to deploy its new connector type to the Provisioning Server.
   - Create a Connector Xpress project by importing the metadata for your custom connector, and deploying a new connector type to the Provisioning Server using it.

   Note: For more information, see the *Connector Xpress* Guide.
2. Restart the Java CS.

Classloader Requirements

To meet classloader requirements in Java CS, the classloader requires that custom connectors dependencies are available to the classloader by placing them in the jcshome/extlib directory. The jcshome/extlib directory is on the classpath of the installed Java CS by default. Pure-scripted connectors may also require external JARs.

Any jar files for libraries that the Java CS does not use, or any other custom connectors you have written that do not use jar files for libraries, can be included in your connector's .zip file. However, any jar files for libraries already used by the Java CS (that is, clashing jar files) cannot be included in your connector. Instead, use the version the Java CS uses.

If your connector uses a different version of a library that the Java CS uses, do either of the following:

- Use the same version the Java CS uses
- If the required version is later than the JCS one, certify the Java CS against the later version.

Note: For assistance, contact CA Support at http://ca.com/support.

Make Custom Connectors Dependencies Available to the Classloader

To make custom connectors dependencies that do not clash available to the classloader, copy any third-party dependency jars for the connector to *jcshome*/extlib, then restart the Java CS.

Important! To avoid possible JAR hell scenarios, check that the versions of the jars are the same as the versions bundled with Java CS and do not include the dependancy jar.

Make Pure-scripted Connectors Dependencies Available to the Classloader

To make pure-scripted connectors dependencies available to the classloader, package pure-scripted connectors as a stand-alone connector and include the dependency .jar file in the connectors .zip file, then restart the Java CS.

Note: If you do not include the .zip files in the dependency .jar, use the versions of the dependency .jar files already included in the Java CS by default.

Chapter 3: High Level Connector Design Considerations

When designing and implementing a new connector, there are a number of major questions to consider. These questions 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

When designing a connector you 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 CA Identity Manager Provisioning Server. Using the built-in DYN parser table support greatly simplifies the connector development process. That is, you do not 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 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 managed objects can reference them. For example, there can be a requirement 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 managed.

- What are the minimum attributes you require to implement on each object class to provide a useful *shallow* connector?

  We recommend that you address this requirement in your 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 consider the containment mode you 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 flat connectors, define *Virtual Containers* in the connector's metadata, or for legacy support, in the connector's /conf/connector.xml file. This makes objects of each type appear grouped into top-level logical containers for the user.

Virtual containers are not 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.

Consider the following when designing a connector:

- 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, consider the following when deciding which technology or API the Connector uses to connect to the endpoint:

- 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 can ask to process any number of requests concurrently.

This makes using 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 can be other objects that you need to guard using 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 their particular advantages to using native connection pooling support, if it is available?

  - Connection pools have two main advantages:
    - Improving scalability and throughput when creating 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 does not grow in an unbounded way, even under sever loading.

- If your connector deals with any multivalued attributes, then use the Java CS to 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 added to existing list is provided
  - REMOVE –List of values removed from existing list is provided. A null list means that the attribute is removed.

  This means that for multivalued attributes that are modified, determine whether the chosen technology API best suits updating using:

  - forceReplaceMode=REPLACE metadata setting for each attribute. Your connector is provided with the complete list of new values, regardless of the mode chosen by the client application.
  - forceReplaceMode=DELTA–Separate lists of items added or removed are 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).There is also forceReplaceMode=PRESERVE which disables all modification item rewriting for an attribute on which it is set.

- Are you allowed to bundle the necessary libraries with your connector?

  If you are not allowed to bundle the libraries, document the location of the libraries and the location where they are copied, so the Java CS can find them, with any additional configuration burden on the Java CS.

- Your connector can use its resource or directory to include any configuration files and utilities, or both, which are not directly part of the connector code. If your connector is a port of an existing connector, and migration is required, we recommend that you put migration scripts here.
- 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 high level of confidence in the library concerned, host the connector in a separate Java CS instance to help ensure that if any JNI problems cause the host Java CS to crash, other connectors are not impacted.
- Does the connector need to use multiple APIs to communicate with the endpoint? If so, the design and implementation are greatly complicated, particularly in the areas of connection pooling, resiliency, and search result streaming.

Special Character Considerations

Pay careful attention to the testing the handling of special characters early in the design and prototyping stage. Problems handling special characters in RDN values have been known to force which API is used. It is important to quote any characters that are special to the connector's chosen API correctly on the way into and out of the native endpoint system, and to quote any characters special to LDAP.

The SimpleLdapName and SimpleRdn classes come in handy when dealing with native names and DNs.

Quote the following special characters 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
- , (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 does the search start?
2. What scope applies 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 is 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 are returned for each object falling within the specified scope and matching the specified filter?

How Your Containment Model Impacts Your Search Strategy

Your decision to implement a flat or hierarchical containment model affects the way you implement your connectors search strategy and the type of filter conditions you apply.

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 automatically has 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,* with some concrete implementations that provide useful simplifications of the filters themselves. For example, as a map of the attribute assertions contained.

Search Strategy Considerations

Consider the following when implementing a search strategy:

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

  Where possible, connectors should use filtering restrictions at the lowest possible level in the implementation. Filtering restrictions prevent extra search results from being returned which you then filter 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 through metadata, deal with performing post-filtering on search results. This accounts for cases where the connector cannot natively respect all the clauses in a search filter.

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

  The ability to stream large sets of search results is an important behavioral characteristic for a connector. While it is possible to achieve streaming with most APIs, a performance cost can be incurred. Where streaming cannot be achieved (for example, with DYN or JDBC), do the following:

  - 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 consider the following:

- Decide if the Java connector is completely deprecating the C++ connector. If so, the Provisioning Manager plug-in does not need to concern itself with backward compatibility. Alternatively, consider whether the connectors Provisioning Manager plug-in detects 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-backward compatible changes (that is, changes rather than simply additions) to an existing schema, parser table, or format of any attribute values, then 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 necessary.
- 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.
- The Provisioning Server now supports the eTAgentOnly operational attribute, which when included in the requested return attribute ids for a search, stipulates it is 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. Use 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 can mean extending an existing connector implementation).

   b. Create new derived classes as required, and verify that they are referenced properly in connector.xml.

   c. Derive basic metadata for the object classes 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 the connection to the endpoint, which requires connector-level metadata settings are complete and correct. 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 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). Implement and test early and carefully as the ApacheDS framework on which the Java CS is built uses lookup operations internally to verify the sanity of the other operations. Hence, any bugs at this stage are a road-block for the connector's implementation.

   When the array of requested attribute ids is null, all attributes (including expensive ones) should be returned. This behavior 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 multivalued attributes are supported, then carefully consider whether using the *forceModificationMode=REPLACE* or *forceModificationMode=DELTA* metadata settings on them 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 use 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 uses *search one class at a time* support, it can make sense to implement selective 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, when you have written streaming logic such a phased approach is unlikely to be required.
- After getting searches on managed object classes working, implement searches targeting unmanaged object classes as these searches are often required when using the Provisioning Manager, as described in step 9.

8. We recommend that you start writing any custom validator and converter plug-ins required by any of your connector's attributes at this point. 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 test incrementally if desired, but we recommend that 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 can produce problems we recommend that you test the implementation.

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

     Note: For more information about 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 set up routing rules to your development JCS for your connector's endpoint type, if you stipulate that the endpoint type is 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 uses direct associations, consider the following:

    - Can its attribute-style processor derive from com.ca.jcs.assoc.DefaultAssocDirectAttributeOpProcessor to do the heavy lifting? In any case, verify 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 automatically takes care of reverse association handling?

    - If your connector returns true, is it necessary to exclude any operations from this automatic handling using *getAutoDirectAssocExlusions()?*

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). We recommend that you test to determine that your connector behaves properly when any of its connection-related attributes are modified at this stage of the implementation. Consider the following:

    - Activation is treated specially, that is, the ADD request for a connector instance is not retried. If connectivity cannot be established at this stage then the ADD fails, and the customer has to retry the ADD manually. The resiliency support comes into effect only after the first successful ADD for the connector instance (after a JCS restart).

    - Testing for resiliency involves trying to determine all the failure conditions your connector can 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 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. Configure retries to run for minutes rather than hours. At some point the connector needs to stop trying to reestablish connectivity 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: Connector Concepts

This section contains the following topics:

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 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 through 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 ignored for any JavaBeans mentioned in the override file.

Note: For more information, see com.ca.jcs.ImplBundle, the SDK sample connectors, 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.
- Helps 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 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 deactivates, and reactivates the connector. The Java CS closes the current connection manager and then reopens a new connection configured with the new settings.

Some connectors use multiple forms of connectivity to the endpoint. In this case, 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(). However, all connection managers can use the connection manager support classes allowing for minimal coding and customizing through the connector.xml file.

Typically, connection managers are set up and torn down in the activate() and deactivate() methods for your connector, which are defined in the com.ca.jcs. Activatable interface (also implemented by all styles of processors). As a general guide it is a good idea to setup anything of interest to multiple styles of processors in the connector's activate() method, to avoid concerns about the exact order of activation.

Disable Connection Attributes Rollback

Typically, when an active or live connector has its connection attributes modified, you can perform a rollback of the connection attributes if the new connection attributes (such as a new password) do not lead to a successful connection. If necessary, you can disable the rollback.

To disable the rollback

1. Do one of the following:
   - Set either of the Connector instance attributes CONN_ROLLBACK_CONNECTION_ATTRS or !rollbackConnectionAttrs! to false.

Note: You can set the virtual attribute rollbackConnectionAttrs! when acquiring your connector. You can map it to any attribute as long as it has a connectorMapTo=!rollbackConnectionAttrs!

- Set the rollbackConnectionAttrs property in Connector.xml to false.
- **Note:** If the connector instance attribute is set, it takes precedence over the rollbackConnectionAttrs property.

Example: Setting rollbackConnectionAttrs to false

This example shows you how to set the rollbackConnectionAttrs property to false. For example, the AS400 connector sets the property to false because by default three attempts to connect with the wrong password causes the user acquiring the connection to be locked out:

```
<property name="rollbackConnectionAttrs">
    <value>false</value>
</property>
```

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 triggered, 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 64).*

Validators

A *validator* is a Java class that verifies the validity of either individual values, attributes, or an entire object class. The Java CS uses validators to help 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 checks modification requests to verify that values are always in the accepted set. The built-in com.ca.jcs.validator.meta.EnumValueValidator already does this verification 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. The Java CS infrastructure uses converters 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 can use a US date style, for example, 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. Converters can assume that 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.

As well as running on attribute values, the converters are applied to RDN components of a DN. For example, ForceCaseConverter is used to force appropriate components of a DN into upper case (for ORA connector) due to inconsistency in the native system.

Note: See *Converter and Validator Plug-Ins Registration* (see page 64) for instructions about how to register converter plug-ins.

Exceptions

Translate local exceptions from your endpoint system Java API into exceptions provided by the LDAP protocol with the closest possible ResultCodeEnum values.

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

When raising exceptions in your connector code (possibly by translating native endpoint exceptions), use your connectors *ldapExceptionPrefix* as the start of all exception messages. Using this prefix distinguishes 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.

Note: For more information on exceptions, see the JCS Javadoc for either the ApacheDS (included in the SDK installer) or the Java CS.

Note: For more information on other implementing classes that are not listed, see org.apache.directory.shared.ldap.message.ResultCodeEnum and org.apache.directory.shared.ldap.exception.LdapException,

LdapNameAlreadyBoundException

Thrown when an object with the same name as the one you are trying to create on the endpoint system 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 exception when you are having communication exceptions with the endpoint system.

Important! This exception is 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 the result code of the exception to ResultCodeEnum.UNAVAILABLE. The resiliency support retries the operation which caused the failure.

LdapConfigurationException

Thrown when an error in the configuration of a connector or the Java CS is encountered. Try to use more specific exceptions. Avoid using this error code if possible, and provide details of the error 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. So that handling size limits are not an issue, use sdk.com.ca.jcs.enumeration.RawNamingEnumeration or one of its derived classes.

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 removes 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 uses additional JCS API calls not present in the SDK connector code, you do not need to recompile it. Some classes in the 8.1SP2 JCS API have been moved into sub-packages in this release. If you have used extra API calls, make minor updates to import statements if necessary.

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

Chapter 5: SDK Sample Connectors

This section contains the following topics:

Sample Connector Overview

The SDKDYN 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 computer running the Java CS. Like all connectors, it has an attribute-style processor that 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.

The SDKDYN sample connector is a typical DYN connector (with developer-maintained metadata). That is, 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 contrived example.

As recommended, it makes extensive use of metadata and sample validator and converter plug-ins. The SDK sample connector is also 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.

Note: The sample SDK static connector is located in jcs-sdk-deprecated.zip in the SDK distribution. The new DYN-based SDK connectors (SDKDYN, SDKFS, SDKSCRIPT, SDKCOMPOUND) supersede the static SDK sample.

Possible Clients

The Provisioning Manager can serve as the main client to drive the DYN-based SDK sample connectors (SDKDYN, SDKFS, SDKSCRIPT, SDKCOMPOUND).The DYN User Interface plug-in built in to the Provisioning Manager is used for these sample connectors. To drive the SDK sample connector using the Provisioning Manager, you must install the Admin SDK Option as the SDK User Interface plug-in is required for the SDK sample connector.

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

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

The CA Identity Manager Web GUI can also be used as the client to drive the following DYN-based SDK sample connectors: SDKDYN, SDKFS, and SDKSCRIPT. However, before the SDK samples can be appear in the CA Identity Manager UI, run the Role Definition Generator based on the SDK samples' metadata, and copy the Role Definition Generator output files to the specific folders.

Note: For more information about running the Role Definition Generator, see How you Generate CA Identity Manager User Console Account Screens in the *Connector Xpress User 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, regardless of whether you use JMeter or another LDAP enabled testing framework. Informal manual testing can 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 top-level 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.
- The Java CS spawned for the test run is automatically shut down when all tests are completed.

  Note: Verify that you do not have a development Java CS (listening on port 20412) already running before executing them.

The JXplorer utility can also be useful for adhoc one-off testing during connector development, particularly for running manual queries to verify the state of a connector during the early stages of development. The JXplorer utility is included as part of the Provisioning Manager installation.

Important! Be careful to include any important test steps in your JMeter test to help ensure that regressions do not go unnoticed during development.

Compiling the Sample Connectors

To compile the SDK Sample Connector, 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.

If the default development Java CS is started on the same computer as a non-development Java CS service, conflicts can occur. Therefore, the default development Java CS is configured to start on port 20412 (non-secure) or 20413 (secure TLS).

You can change this port by:

- Editing the ports specified in *jcs-sdk-home*/conf/server_jcs.xml.
- Changing the *jcs.test.port* setting in *jcs-sdk-home*/build.xml

However 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 the Java CS

Sample Connector Upgrading

The Java CS SDK uses Java deprecation to simplify upgrading connectors between different versions of CA 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 about recommended alternatives to deprecated interfaces, see the Java CS Javadoc.

SDKCOMPOUND Connector

The SDKCOMPOUND connector reuses SDKDYN code but has its own specialized metadata.

The SDKCOMPOUND connector demonstrates how the Java CS framework can handle compound values nested to any level. Compound values are single attribute values that contain multiple subcomponents, for example, an address with street, zip code, and country components.

Some connector technologies (for example, JDBC) impose restrictions on the number of compound value levels that can be supported. Also, the CA Identity Manager and Provisioning Manager GUI technologies do not currently support compound values nested inside other compound values.

The following are important metadata settings used by the SDKCOMPOUND connector:

- isCompoundValue on the compound classes
- compoundValueClassRef on attributes which have compound values
- assocType=COMPOUND_PARENT in association attribute in compound value class which identifies its parent, or assocType=COMPOUND_CHILD, if the association attribute is an attribute in the parent class.

The SDKCOMPOUND connector does not use either assocType=COMPOUND_PARENT or assocType=COMPOUND_CHILD as it stores JSON value literally. However, JDBC compound value support does rely on assocType=COMPOUND_PARENT being set, as the parent and compound value are stored in separate database tables.

The latter table references the primary key of the prior table.

Note: For more information, see *sdkcompound_metadata.xml* and jdbc_compound*_metadata.xml files and their related JMeter tests.

JSON (JavaScript Object Notation), a subset of JavaScript syntax, is used to represent compound values. You can find examples of JSON attribute values in the sdkcompound_core_basic.jmx JMeter test.

Registering the com.ca.jcs.converter.meta.JSONCompoundValueClassConverter class in this connector's connector.xml file does the opposite of the JSONReverseCompoundValueClassConverter converter used internally by the Java CS framework to convert incoming JSON values into nested JNDI Attributes objects. It may be of interest for connectors dealing with compound values which want to be passed JSON objects or JSON strings, rather than nested JNDI Attributes.

An extension was made to allow property and class validator and converter plug-ins to be triggered by type, rather than exclusively by listing metadata settings in "metadataPropNames" in a connector's connector.xxml file (or server_jcs.xml at the JCS-wide level). For more information, see the section where the com.ca.jcs.converter.meta.JSONCompoundValueClassConverter converter is registered in the SDKCOMPOUND connector's connector.xml file:

```
<bean class="com.ca.jcs.cfg.MetaPluginConfig">
                    <property name="type" value="COMPOUND_VALUE_CLASS_REF"/>
                    <property name="pluginClass">
                      <value>com.ca.jcs.converter.meta.JSONCompoundValueClassConverter</value>
                    </property>
                    <property name="pluginConfig">
                      <bean class="com.ca.jcs.converter.meta.JSONCompoundValueClassConverter$Config">
                        <property name="convertToString" value="true"/>
                      </bean>
                    </property>
                  </bean>
```

The class converter is triggered for any attribute of type "COMPOUND_VALUE_CLASS_REF", rather than the presence of metadata settings on target attributes.

Compound Value Support

The SDKCOMPOUND connector demonstrates configuration and use of compound values in a fully functional connector. However, as it has the luxury of storing the compound values as simple strings in property files stored on the JCS's local filesystem, it is able to bypass some of the complexity of compound value support in the JDBC connector where compound values need to be split up and stored in separate database tables.

If you determine that you require compound value support for your connector, consider the following:

- The metadata format (defined in datamodel.xsd) and the JCS framework impose no restrictions on the level to which compound value classes can be nested. Hence the SDKCOMPOUND connector includes the *PersonalizationOption* compound value class which demonstrates two levels of nesting. Hence any restrictions are due to either the connector (because of the endpoint system technology) or higher level clients with User Interfaces.
- Compound values can reference other object instances in associations (using DNs for instance), but no other objects can reference them.
- Because the individual attributes in compound values are not represented by LDAP attributes (but rather as values of a JSON object) it is often the case that connectorMapTo settings are not required for them. For this reason the *"isCompoundValue"* setting (which has a similar effect to connectorMapToSame) signifies that connectorMapTo values need only be provided when a compound attribute's name is unacceptable to the endpoint system. For example, due to restrictions on the characters permitted in SQL column names.
- The CA Identity Manager user interface imposes a restriction where nested compound values are not displayed. This is not a problem for DYN JDBC connectors but it is a problem for the SDKCOMPOUND connector.
- In CA Identity Manager Provisioning Manager, nested compound values are not displayed in an intuitive way, that is, user needs to understand JSON syntax.
- The JDBC connector does not support nested compound values as doing so would necessitate support for compound primary keys.

  All schemes are supported for specifying the keys of parent and compound value objects, including generated keys on either or both ends.

  In relational 1:N associations, the table on the "N" side must be responsible for storing the key for the "1" side. Therefore, the table storing compound values needs a column specifying a key for the parent object, and the *COMPOUND_PARENT* assocType is used on the attribute defining the association between them.
- Both indirect and direct associations are now supported, with metadata specifying which style applies to which attributes. Associations between a compound object and its parent are modeled as direct associations.

SKDFS Connector

The SDKFS connector reuses SDKDYN code but has its own specialized metadata.

The SDKFS connector shares the same Java Implementation as the SDKDYN connector but it is hierarchical. The containers are represented as file system directories. As a result, the SDKFS connector requires its own distinct metadata document, sdkfs_metadata.xml.

The SDKFS connector demonstrates a hierarchical namespace by extending the SDKDYN connector to handle management of file system directories, in addition to the account and group files.

The connector's metadata document defines a real container class, rather than the virtual containers defined for SDKDYN.

Note: For more information, see the following sections of com.ca.jcs.sdk.SDKAttributeStyleOpProcessor:

- CONTAINER_TYPE—Indicates that containers (file system directories) are being managed.
- getContainerPath() method—Handles resolving possibly nested Distinguished Names to file system paths.

SDKSCRIPT Connector

The SDKSCRIPT connector reuses SDKDYN metadata but its implementation is entirely in JavaScript. You can also deploy the connector using Connector Xpress using the provided template.

The SDKSCRIPT connector demonstrates only flat functionality, unlike SDKFS.

The SDKSCRIPT connector uses a copy of the metadata used by the SDKDYN connector, but all its logic is implemented in JavaScript instead of Java. This is achieved by the script operation bindings that can be found in /conf/sdkscript_opbindings.xml, which wrap a stub connector as configured through /conf/connector.xml.

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

SDKUPO Connector

The SDKUPO connector is an example of a connector that invokes user-specified external applications in response to user provisioning requests. The connector invokes user-specified external applications by using JavaScript in the operation bindings. Sample operation bindings are Pre Add Account, Post Add Account, and such. The script in the operation bindings calls external services, also known as program exits.

The SDKUPO Connector reuses the SDKDYN metadata but with some additional metadata relating to the program exits. Two exits are provided: one for sending email messages in response to provisioning requests, and one for logging the requests to a file. Thus, as with the SDKDYN connector, the SDKUPO connector has a flat namespace with virtual containers.

The script for the SDKUPO connector is based on the SDKSCRIPT connector. The code and bindings are in sdkuposcript_opbindings.xml. Additional functionality is added to the SCKSCRIPT connector code to handle the operation bindings and the execution of the exits. Although the bindings and exits can be defined and organized using an external editor, you can achieve the same result using Connector Xpress.

Note: For more information see the *Connector Xpress Guide*.

The SDKUPO Connector can operate in two modes:

- Managed mode–performs essentially the same operations as the SDKSCRIPT connector.
- Non-managed mode–demonstrates the use of program exits

SDK Connector

The SDK connector is deprecated. The connector is available in jcs-sdk-deprecated.zip. The SDK connector reuses the SDKDYN source but the metadata is mapped against the SDK schema rather than the DYN schema.

The SDK connector shares the same Java implementation as the SDKDYN connector, and differs only in that it uses the SDK schema instead of the DYN schema. Consequently, it is unable to use the DYN User Interface plug-in built in to the Provisioning Manager. For the Provisioning Server to make the SDK schema available and Provisioning Manager to make the SDK User Interface plug-in available, the connector requires parts of the C++ SDK, which has more limited functionality, installed.

Note: For more information, see Install SDK Connector Pre-requisites (see page 57)

Install Deprecated SDK Connector Pre-requisites

Although the deprecated (non-DYN) version of the SDK connector implementation is in Java, you need to install install several components from the Admin installation. The Provisioning Server C++ SDK Admin installation is a prerequisite for the deprecated (non-DYN) version of the SDK connector.

Install the Provisioning Server C++ SDK

1. Install the following components from the Admin installation:
   - The SDK schema
   - The POP script that adds the namespace definition and its default policy container definition.
   - The SDK plug-in to the Admin Manager GUI.
2. If you want to run against the Java CS SDK so that you can debug and change the connector, do the following:
   a. Run the Java CS SDK installer.
   b. Navigate to the installation directory and then enter the following command:

      ant dist

   c. Start the Java CS using the provided Eclipse or IDEA launch targets> alternatively run *jcs-sdk-home*/bin/jcs.bat if your target seems to have a misconfigured classpath or similar.
   d. Use Connector Xpress to help ensure that the SDK namespace for the target Provisioning Server is being routed to this Java CS.
   e. Configure the default ports 20412, and 20413 for ssl /tls connectivity.

      Note: For a production Java CS, use the ports 20410, and 20411 for ssl /tls connectivity.
3. If you want to run the binary sample, do the following:
   a. Unzip the Java CS installer to a directory ${DIR}
   b. Unzip the Java CS samples to the same directory ${DIR}
   c. Run the Java CS installer from ${DIR} and register the SDK connector (may as well register all connectors).

   You can now acquire the SDK directory using the SDK plug-in to the Provisioning Manager GUI.

Release a Customized SDK Connector Example

Often it easiest to become familiar with the Java CS SDK variants by simply using them unchanged, and debugging using an IDE. Perhaps this progresses to changing the existing metadata and logic to try out various changes. However, at some point in a connector's development it will be necessary to choose a proper name for the connector that identifies its function and ensures it is not confused with the SDK connectors released with the Java CS.

This example shows you how to release a connector based on SDKDYN as a new connector called *myconnector*. Do the following:

1. Copy the connectors/sdkdyn/ directory and all sub-directories to connectors/myconnector/.
2. Rename all the classes from SDK* to MyConnector*.
3. Change references in conf/connector.xml to associate the namespace and update any class references to the SDK connector to point to *MyConnector*.
4. Move the conf/override/ directory out of the myconnector/conf/override directory.
5. Restore the directory and update the SAMPLE.connector.xml file before you deploy the connector.

   Note: If the override files are present, they provide overriding settings for the connector. The installer does not overwrite the override files, which means any custom settings placed in this directory and deployed always override the configuration.
6. Change all references to sdkdyn in the build.xml file to match your connector's name. For example change the conn.pkg property used in the .jar .zip, and pop.ldif file name and such.
7. Change the following settings in your conf/*_metadata.xml document to match you connector's name. For example, SDK to MyConnector.

```
<namespace name="connectorname Namespace">
<doc/>
<metadata name="abbreviation">
<value>
<strValue>DYN</strValue>
</value>
</metadata>
<metadata name="version">
<value>
<strValue>1.0</strValue>
</value>
</metadata>
```

```
<metadata name="implementationBundle">
<value>
<strValue>connectorname=</strValue>
</value>
</metadata>
```

namespace name

Defines the name of your connector's namespace/endpoint-type in the metadata.

Example: namespace name="*MyConnector* Namespace"

metadata name

Defines

Value:

Example: <strValue>*MyConnector*</strValue>

8. Change the following settings in your connector's conf/connector.xml file to match your connector's name and new class names. For example, *SDKDYN* to *MyConnector*.

```
<beanclass="com.ca.jcs.ImplBundle"id="connectorname">
<propertyname="name">
<value>connectorname</value>
</property>
...
<propertyname="connectorTypeName">
<value>connectornameNamespace</value>
</property>
...
<propertyname="messageResourceBundle"><value>conf/com/ca/jcs/sdk/validator/validator</value>
</property>

<propertyname="staticMetadataFile">
<value>/conf/connectorname_metadata.xml</value>
</property>
<propertyname="indirectAssociations">
<value>false</value>
</property>
<propertyname="connNamingAttr">
<value>eTDYNDirectoryName</value>
</property>
<propertyname="connectorTypeClass">
<value>com.ca.jcs.meta.MetaConnectorType</value>
</property>
<propertyname="connectorClass">
<value>com.ca.jcs.sdk.connectornameMetaConnector</value>
</property>
```

bean class

Defines

Example: bean class="com.ca.jcs.ImplBundle" id="*MyConnector*"\

connectorTypeName

Defines

Example: *MyConnector* Namespace

name

Defines

Example: *MyConenctor* Namespace

staticMetadataFile

Defines

Example: /conf/*MyConnector*_metadata.xml

connectorClass

Defines

Example: com.ca.jcs.sdk.*MyConnector*MetaConnector

Note: The build.xml file for each DYN based connector includes an XSLT transform (popldif.xslt) which uses your metadata to generate a build/_uninst/*pop.ldif file. The Java CS installer uses this file to register your connector when you run ant inst and copy the resulting .zip for your connector to ${DIR}, as shown in the preceding example.

DYN Class Names

The DYN (extended) schema provides a set of generic classes that you can map to.

The set of standard classes includes:

eTDYNNamespace

Lets you map the endpoint-type and namespace level.

eTDYNDirectory

Represents endpoint or connector level.

eTDYNAccoun

Specifies the account.

eTDYNAccountContainer

Specifies the accounts container.

eTDYNGroup

Specifies the group and its container.

eTDYNGroupContainer

Specifies the groups container.

eTDYNPolicy

Specifies the policy.

eTDYNPolicyContainer

Specifies the policies container.

The following set of generic classes are also available:

- eTDYNObject001-eTDYNObject035 (35 generic classes)
- eTDYNContainer001-eTDYNContainer010 (10 generic containers)

Simple single account and single group use cases can get by with eTDYNAccount/eTDYNGroup mappings. When mapping multiple groups, you can use eTDYNObject* classes You can also map any other additional classes using eTDYNObject.

eTDYNContainer* classes have many attributes that can be mapped, like the eTDYNObject* classes, so you can use them as more than simply containers for other objects if necessary.

Chapter 6: Configuration Files

This section contains the following topics:

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, which eliminates unnecessary code and offers a high level of configurability for virtually all the Java CS and connectors.
- The Spring Framework reads global configuration for Java CS from the server_jcs.xml file. You can change server-side configuration settings, and add new global-scoped validators and converters in the server_jcs.xml file.
- 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 overrides 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 registered are activated for each objectclass or attribute. You can change this by using normal LDAP modify operations targeting the endpoint type.

Configuring connector-specific settings is achieved by JavaBeans deriving from ConnectorConfig that introduce any connector-specific properties that are required. Instances 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

The *-dist* target packages the connector in its *build.xml* file as a single file named *jcs-connector-*.jar* which can 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/.

The zip file includes every file it deploys to a production Java CS. For example, its *jcs-connector-*.jar* file, any third-party libraries, and all sources of configurations settings.

The *.zip* file is directly extracted under a jcs-home directory. Folder names must exactly match the standard names used by the Java CS on which it depends. For example, *lib/, extlib/, conf/override/<connector>/ .*

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

Note: 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—** Connectors that make their own connector-specific schemas known to the ApacheDS server hosting the Java CS use this optional file. Verify that this file is in the format dictated by the OpenLDAP standard.
- **/conf/*_metadata.xml—**Connectors that have static metadata or metadata maintained by developers rather than users can use this optional file. Such connectors are referred to as static connectors, as opposed to dynamic connectors where metadata is generated dynamically in Connector Xpress by users.

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 about value strings, see com.ca.commons.datamodel.DataModelValue.

- **propertyPluginConfigs** –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. However, it 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**–List of plug-ins driven by metadata settings, where the plug-ins work over all 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.EncyptPropertyConverter* in the *server_jcs.xml* file.

Connector.xml Files

Connector.xml files serve as specific placeholders for connector configuration. Primarily, they tie the connector type name with the implementation bundle for that connector, and specify settings for general connector behaviors, connection pool settings, and registration of plug-ins (validators and converters).

A connector.xml file contains a definition for one or more ImplBundle Spring beans with a number of properties set.

The following are some important properties of Connector.xml files:

Name

Ties the name of the connector type (endpoint-type) to the bundle implementing this type of connector.

connectorTypeName

Specifies a secondary means to associate connector type to the bundle implementing this type of connector, that is, to match the eTNamespaceName attribute value provided in a target DN.

For example, MyConnTypeName in eTNamespaceName=MyConnTypeName,dc=DOM,dc=etasa

staticMetadataFile

Defines any static metadata file used by the connector.

Note: Not required for dynamic deployed connectors because Connector Xpress users create the metadata, rather than connector developers.

staticMethodScriptStyleMetaDataFile

Defines any static op-bindings and scripts for a method or script style connector.

Note: Not required for dynamic deployed connectors because Connector Xpress users create the metadata, rather than connector developers.

defaultConnectorConfig

Sets various connector behaviors. This property is a bean.

allowMetadataModify

Specifies whether metadata modifications are allowed or disallowed.

converters

Defines where all connector-specific converters are registered. This is a MetaPluginConfigSuite bean.

connNamingAttr

Designates the naming attribute for the connector as it appears in target DNs. For example, eTDYNDirectoryName in eTDYNDirectoryName=MyConnName,eTNamespaceName=MyConnTypeName,dc=DOM,dc=etasa.

connectorTypeClass

Defines the name of the implementation class implementing the ConnectorType interface. For example, usually MetaConnectorType or the class which extends it.

connectorClass

Defines the name of the connector class.

connectionManagerClass

Defines the name of the class that implements the ConnectionManager interface, which commonly provides connection pooling facilities.

rollbackConnectionAttrs

Defines a Boolean property, which determines if a rollback of connection-related attributes occurs when connection attributes are modified with values that result in unsuccessful connection to the endpoint.

Value: True by default if unspecified.

This attribute can be overridden through a connector level attribute mapped to !rollbackConnectionAttrs!

Note: For more information, see Disable Connection Attributes Rollback.

Chapter 7: The Object Model

This section contains the following topics:

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

The following are the two different syntaxes for metadata documents:

- Data Model (see page 69)
- Operation Bindings (see page 72)

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. The data model defines object classes and their properties, and adheres to the 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. The schema includes a *SimpleValueGroup* group (which in turn references *PrimitiveValueGroup*) that defines the choices available for a property's value. The SimpleValue complex type allows default values to be specified using the default= XML attribute, which are used for a property when no explicit value is provided in an LDAP ADD request.

An example data model element can 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 interact with the data model. For example, the data model described previously requires a Boolean *isNaming* metadata property with the value *true* specified for the *eTDYNAccountName* property. This specifies that this property describes the naming attribute for its parent object class.

Data Model Types

You can find the list of available data types which can be represented by referencing the *SimpleValueGroup* production in *jcs-sdk-home*/conf/xsd/datamodel.xsd XML schema.

The basic types, which are part of the XML schema specifications, are:

boolValue

Represents a Boolean value, true or false.

intValue

Represents a 32-bit signed integer in the range [-2,147,483,648, 2,147,483,647]

longValue

Represents 64-bit signed integer.

FloatValue

A 32-bit floating-point decimal number as specified in the IEEE 754-1985 standard.

dblValue

A 64-bit floating-point decimal number as specified in the IEEE 754-1985 standard. The external form is the same as the float datatype.

dateValue

Defines a UTC date in "YYYY-MM-DD" syntax

Example: 1970-01-01

dateTimeValue

Defines a UTC date and time in "YYYY-MM-DD'T'HH:MM:SS" syntax.

Example: 1970-01-01T00:00:00

timeValue

UTC time value in "HH:MM:SS" syntax

Example: 00:00:00

enumValue

Referenced to fixed set of alternatives defined in metadata.

Note: For more information, see the SoftdrinkVarieties enum in the SDKDYN connector's sdkdyn_metadata.xml for an example.

flexiStrValue

Basically a string, but allows validator/converter plug-ins to be triggered.

Note: For more information, see <flexiStrValue type="noComma"> type in the SDKDYN connector's metadata.xml, referenced in its connector.xml, for an example.

binaryValue

Signifies the value is a raw binary value which should be passed through unchanged.

In addition to these basic types the following types are also types built on top of them:

- setValue

  An unordered collection of basic types, usually the right choice as the order of the values in an LDAP attribute is not guaranteed to be preserved. Remember to provide a baseType definition.

- sequenceValue

  An ordered collection of basic types. Remember to provide a baseType definition.

- mapValue

  A map made up of entries consisting of a key (of a basic type) which maps to a value (either a basic type, collection, or sub-map).

- *compoundValueClassRef*

  For attributes which have compound values this setting allows the class which defines their content to be named.

  Note: For more information, see the *SDKCOMPOUND sample connector.*

Operation Bindings

Operation Bindings metadata is stored as the value of the *eTOpBindingsMetaData* attribute on the *eTNamespace* object for an endpoint type. The metadata 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 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. The guard 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: If no objectClass elements are specified, signifies the 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 that the connector's script-style processor executes.
- A timing which can be:
  - **PRE**—Execute the opbinding before the target Operation. For example, it can 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 record.

- A strictCompletion Boolean. If true, specifies that a failure encountered while executing the payload (either an exception, or a non-null textual error status return) causes the framework to treat the whole LDAP operation as failed. Where the endpoint system supports transactional behavior (such as DYN JDBC) a failure when strictCompletion is true is treated as a reason to roll back the transaction.
  - An order integer

    If there is more then one guard condition matching a particular LDAP request and their order of execution is important, the order integer specifies the order in which opbindings are executed.

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

Note: For more information about coding of script payloads, see Writing Scripts (see page 111).

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. We recommend this method of implementation for what were previously known a connector program exits in CA Identity Manager 8.1 SP2. For example, the AS400 and OS400 connector which is part of CA Identity Manager r12, includes a sample opbinding.

Enumerations

You can define a set of enumerated values in the metadata and link the enumeration to an attribute (property). This is useful where an attribute has one or more values coming from a fixed set of choices.

Define each enumeration once in the metadata and then link it to every attribute where you require the enumeration, for example:

```
<enum name="SoftdrinkVarieties">
    <val ordinal="4" displayName="Orange">orange</val>
    <val ordinal="3" displayName="Lemon">lemon</val>
    <val ordinal="2" displayName="Lime">lime</val>
    <val ordinal="1" displayName="Cola">cola</val>
</enum>
<property name="eTDYN-str-multi-01">
    <value>
```

```
<enumValue def="SoftdrinkVarieties"></enumValue>

</value>
```

The Provisioning Manager GUI plug-ins and CA Identity Manager web-screen renders these as drop-down lists.

Dynamic Enumerations

Like a static enumeration, a dynamic enumeration also allows an attribute to be a set of enumerated choices. However the choices are dynamic, that is, searches obtain the values. Dynamic enumerations are implemented as an association in the metadata where the other end of the association is a read-only class with one (or at least a small number) of display attributes. To retrieve the names of the objects you want you want the user to select, you search for the objects of the end class.

Such associations do not require a reverse association attribute, as the association is only retrieved in one direction.

The client (that is, the Provisioning Manager GUI) displays this as an association however.

Metadata Definition

The process of writing the metadata for a connector depends on the type 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 the metadata.

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 are identical to the original connector.

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

Note: You must have the Admin SDK installed to gain access to the .PTY files required 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. Manually add some additional metadata items required by the Java CS to the XML document. In particular, metadata properties like connectorMapTo and metadata properties used to trigger validator and converter plug-ins.

Note: For a complete list of all metadata properties, see com.ca.commons.datamodel.MetaDataDefs in the Java CS Javadoc.

Reimplement (in Java) any logic the Java CS framework or the base connector you derive from does not automatically handle, based on the metadata.

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

DYN Schema Extensions

The DYN Schema has been extended for the following attributes:

- Container objects (eTDYNContainerXXX) are extended to 10.
- Generic objects (eTDYNObjectXXX) are extended to 35.
- All capability attributes (for example, eTDYN-str-multi-ic-, eTDYN-str-ic-, eTDYN-str-c-, eTDYN-bool-c-, eTDYN-int-multi-c-, eTDYN-int-c-) are extended to 99, with the exception of the multivalued case-sensitive attributes (eTDYN-str-multi-c-), which are extended to 500.
- The noncapability multivalued case-sensitive or case-insensitive attributes (for example, eTDYN-str-multi-, eTDYN-str-multi-i-) are extended to 500.

- The generic cached (that is, data location equals BOTH) multivalued case-sensitive attributes (eTDYN-str-multi-ca-) are extended to 99. These attributes are included in every DYN object.

  Cached attributes have their values stored on the provisioning server and are therefore accessible without contacting the endpoint system. Where their values are provided by the endpoint system, an explore operation is needed to update the values stored for them to match that on the endpoint system.

- Ninety-nine connection-specific cached multivalued case-sensitive attributes (eTDYN-str-multi-ca-sec-) are added for DYN Directory only. These attributes are added in the "encryptwith" line of the DYN password attribute, which can be used to obfuscate sensitive settings.

DYN Attribute Name Selection

When manually assigning attributes for a DYN endpoint type (rather than using Connector Xpress), eTDYN-str-multi- is usually the best choice as you are less likely to run out of attributes.

If caching is required, we recommend that you use eTDYN-str-multi-ca, as the Provisioning Server does not typically cache the attribute values in the DYN namespace (except for a set of well-known attributes, for example, eTDYNConnectionURL, eTDYNHost, and such). Use a –ca- variant for cases requiring caching, for example, extra connection-related attributes.

For attributes in classes other than accounts and account templates, and non-capability attributes on these classes, the fact that these underlying LDAP attributes are multi-valued and strings is not important, as the metadata specified for them controls whether they accept multiple values and their real type.

For more information about available classes and attributes you may choose to mention in your metadata see DYN Schema Extensions (see page 75).

For capability attributes on accounts and account templates where policy merging by the provisioning server comes into play, the distinction between single verses multi-valued attributes, real type, and case-sensitivity become important at the LDAP level. We recommend that you consider:

- Using attributes with –multi- in their names only for attributes which are really multi-valued. In this case, the Provisioning Server performs a union instead of a greater than test when merging accounts and account templates.
- Using case sensitivity, where "-i" signifies case insensitivity. For example, eTDYN-str-i-01 is a case-insensitive string whereas eTDYN-str-01 is case-sensitive.

- Using -c to represent capability attributes. For example, eTDYN-str-c-01 is a case-sensitive capability string whereas eTDYN-str-ic-01 is a case-insensitive capability string
- Using -bool- and –int- as required, noting that bools accept 1 or 0, and ints accept any integer value.

Note: Where an attribute id matches one of the regexes found in sensitiveAttrIdRegexes property in server_jcs.xml, the attributes are automatically treated as a sensitive attribute and are obscured in logging output (even when logging is turned on at the lowest levels of the ApacheDS code). The substrings password, pwd, and cred are defined to trigger this behavior by default. A good practice is to use attribute names with higher numbered suffixes for such sensitive attributes, allowing them to be excluded from logs across all connectors without negative impact.

DYN Class Name Selection

When manually assigning class names for a DYN endpoint type (rather than using Connector Xpress), consider the following:

- Use eTDYNDirectory for the top-level representation of the connector. Cached attributes (which have –ca- in their names), should typically be chosen to store values which are to connect to the endpoint system, so that the Provisioning Server stores them.
- Use eTDYNAccount for the representation of accounts on the endpoint system.
- Use eTDYNPolicy for the representation of the account templates used to create and manage accounts on the endpoint system. There is usually a strong correlation between the attributes in this class and those defined for eTDYNAccount, with the same attribute names doing double duty in both classes.
- Use eTDYNObjectXXX classes for native objects which are not accounts or containers. This includes live enumerations which are native objects which are not themselves managed (That is, they cannot be created or deleted) but are mapped so that clients can search for all instances. For example, the names of all native permissions configured on the endpoint system.
- eTDYNGroup can still be used instead of one of the eTDYNObjectXXX classes if you prefer.

- Use eTDYNContainerXXX for container native objects, which have the same attributes available for mapping as the eTDYNObjectXXX classes, but in addition can act as containers for them (and other containers). The list of classes permitted within a container is specified using the childTypes metadata setting.
- Configure eTDYNContainer, eTDYNAccountContainer, and eTDYNGroupContainer the same way as the eTDYNContainerXXX classes. You can use these classes if you prefer.

Padding Of Int Attribute Values

The Provisioning Server applies some "0" padding to the value of attributes whose name contain an '-int-' string. This happens if you select the Integer Data Type and check the Synchronized checkbox in Connector Xpress.

For this reason, we recommend avoiding these attributes in favor of standard string attributes where possible. You should only use the "–int-" string for integer capability attributes involved in account template merging/synchronization.

For example, when a value of "22" is assigned to the attribute "eTDYN-int-c-01", the Provisioning Server will pass on the value "0000000022" to the Java CS. This makes it impossible to tell whether the original value was exactly "22", or something else, for example, "022".

The minimum and maximum length restrictions for an attribute affect how the Java CS (specifically the com.ca.jcs.validator.core.LengthRangeValidator plug-in) strips the "0" padding. For example, if the value "0000000022" is received and minLength=3, then the value will become "022". This means you will not be able to tell what the original value was.

How You Define Metadata for a New Connector

If you are creating a connector for which there is no pre-existing specialized schema, we recommend that you create a specialized data model (see page 79) mapping to and from the generic DYN schema. We recommend that you write metadata from scratch that annotates your LDAP schema with all the information required by the Java CS, the JIAM and CA Identity Management account management functionality.

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 (defined using the class name=' XML syntax), is mapped to a database table and its properties are mapped to columns within its parent table.

Note: For more information, see the SDKDYN.

Some connectors can require the similar connectorMapToAmbiguous metadata property. For example, JNDI-based connectors can 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/sdkdyn_metadata.xml file in the SDKDYN sample connector (ignoring metadata properties starting with *pt*.). For a list of supported metadata properties and values, see com.ca.commons.datamodel.MetaDataDefs in the Java CS Javadoc. com.ca.commons.datamodel.MetaDataDefs

Handling Sensitive Data

Among the many metadata properties which can be set for attributes, there are some you can use to protect sensitive data.

Note: For more information, see com.ca.commons.datamodel.MetaDataDefsin the Java CS Javadoc.

DYN Based Connector Creation

You should 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 approach 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 for all connector development as doing so means it is not necessary to write a custom parser table and/or Provisioning Manager C++ User Interface plug-in. Using a DYN schema also simplifies enhancing released connectors as it is only necessary to change metadata mappings. 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.

If the DYN plug-in does not meet your requirements, you can write your own custom Provisioning Manager plug-in for the DYN schema.

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, however we recommend that you use a DYN based approach.

Note: If you cannot use the DYN schema and want to use this approach, see the *Programming Guide for Provisioning* in the CA 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 as they establish a standardized facade over all connectors. JIAM (Java Identity and Access Management) is a Java front end to the Provisioning Server.

Note: The only JIAM-specific setting remaining in r12.1 is beanPropertyName. We have deprecated the following metadata settings**.**

- policyClass
- endPointClass
- accountclass
- policyContainerClass
- groupclass
- rdnAttribute

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

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: Similar expressions are written in short-hand as *connectorMapTo=connector* for the remainder of this section.

2. Verify that the connector's naming attribute has both a *connectorMapTo='*mapping (usually *=name'*) and has *isNaming=true'*

3. Add *connectorMapTo*= values for all the connector's properties which are connection-related or are otherwise singled out for special handling that derived connectors want to use.

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

   This is because the Java CS framework informs your connector that it wants 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.

   The possibilities for reuse in the code that establishes connections are greatly increased, 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. Select a *connectorMapTo*= value.

   b. Verify that its naming attribute has a *connectorMapTo*= value and has *isNaming=true.*

   c. Verify all properties which the connector implementation supports (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.commons.datamodel.MetaDataDefs are correctly applied.

Special connectorMapTo Values

There are some connectorMapTo values provided by the Java CS framework that you can choose to map attributes to, where the framework provides the value rather than the endpoint system. Each of these values has a constant defined for it in the com.ca.jcs.BaseConnector class., As with all values which are special to the Java CS framework these constants start and end with the '!' character (the value of the BaseConnector.CONN_SPEC constant).

We recommend that you familiarize yourself with the constants defined in this class through their JavaDoc, especially CONN_DN ("!dn!") and CONN_NAME ("!nameId!"). CONN_ROLLBACK_CONNECTION_ATTRS and SEARCH_RESULTS_STREAMING may also be of interest.

Natively Generated Attribute Values

Some endpoint systems are capable of generating the values for some attributes, instead of them being passed in by client applications.

For example, the JDBC connector supports generation of primary or alternate keys for objects in two ways (refer to the JDBC compound metadata documents and related JMeter tests):

- Through "sequences" (used by Oracle) where the sequence name can be provided using the "connectorGenerator" string metadata setting on the attribute for which values are to be generated. This sequence is then used to generate the value during the ADD operation.
- Through "identity columns" (used by Microsoft SQL) where the attribute is simply tagged with the "isConnectorGenerated" Boolean metadata setting. The setting informs the Java CS framework that no value should be provided for this attribute during the ADD request as the endpoint system automatically assigns it a value.

  You can use these metadata settings for the same generic purpose in any connector implementation, where the endpoint system supports similar ways of automatically generating attribute values.

Container Definition

When you are defining metadata mappings for classes which are containers (that is, they can contain objects of other classes and other containers), consider the following points:

- Containers can be mapped like any other class, and have any number of attributes mapped (including ambiguous mappings). A crucial difference is that containers have a *childTypes* metadata setting which list the names of the LDAP class names which the container is permitted to contain. As the deprecated *isContainer=true* metadata setting is no longer used, confirm that *childTypes* is defined. Also confirm that it contains all class names which can appear as children, including possibly the container class itself where nesting is permitted. If class names are missing then objects with these classes will not appear in the results of search operations.

  Note: This setting is of critical importance for all containers, whether real containers or Virtual Containers.

- The top level eTDYNDirectory class acts as a container itself and will consequently need a childTypes setting which names all of the container classes which can appear as its direct children (regardless of whether they are real containers or Virtual Containers).

- If the endpoint is hierarchical, define a mapping to a class that actually exists on the endpoint, as compared to Virtual Containers.

- Use ambiguous mappings if there are multiple varieties of containers and you want to simplify the view you present to clients of the connector. For example, mappings for the JNDI connector often use a connectorMapToAmbiguous setting for eTDYNContainer.

  Note: For more information, see the JavaDoc.

- If the endpoint is flat, then define Virtual Containers as a way of grouping objects of each class and providing a cleaner view of the endpoint for the customer. These containers are *virtual* because they do not actually exist on the endpoint, but are an abstraction introduced by the Java CS.

  For backward compatibility, you can define Virtual Containers in a connector's connector.xml file, however this is deprecated in favor of defining virtual containers in metadata.

- Map every container class, whether real or virtual, to one of the eleven container classes available. For example, eTDYNContainer and eTDYNContainer001-010.

- Avoid implementing a *containerList* attribute which lists all the containers under a parent container (or your root connector). Doing this breaks the LDAP containment model, as asking for attributes on the parent object involves searching for all children. Instead, inform the Provisioning Server whether the search targets containers the customer has asked to manage (the default) or all containers that exist on the endpoint. Include eTAgentOnly in the searches requested return attributes.

Metadata Settings for a Real Container Class Example

The following is an example of important metadata settings for a real, that is, not virtual, container class. This example is from the metadata for the SDKFS sample, sdkfs_metadata.xml:

```
<class name="eTDYNContainer001">

        <metadata name="childTypes">
                <value>
                <setValue>
                        <baseType>
                                <strValue/>
                                                </baseType>
                                                <val>

        <strValue>eTDYNContainer001</strValue>
                                                </val>
                                                <val>

        <strValue>eTDYNAccount</strValue>
                                                </val>
                                                <val>

        <strValue>eTDYNObject001</strValue>
                                                </val>
                                        </setValue>
                                </value>
                        </metadata>
                        <metadata name="connectorMapTo">
                                <value>
                                        <strValue>folder</strValue>
                                </value>
                        </metadata>
                        <property name="eTDYNContainer001Name">
                                <value>
                                        <strValue>Container Name</strValue>
                                </value>
                                <metadata name="isNaming">
                                        <value>
                                                <boolValue>true</boolValue>
                                        </value>
                                </metadata>
                                <metadata name="isRequired">
                                        <value>
                                                <boolValue>true</boolValue>
                                        </value>
                                </metadata>
```

```
                                        <metadata name="displayName">
                                                <value>
                                                        <strValue>folder name</strValue>
                                                </value>
                                        </metadata>
                                        <metadata name="connectorMapTo">
                                                <value>
                                                        <strValue>dirname</strValue>
                                                </value>
                                        </metadata>
                                </property>
    … arbitrary other properties can be mapped for the class…

    </class>
```

The following settings in the above example are important to consider:

groupMappings / groupContents / displayName

These settings are required by the CA CA Identity Manager and CA CA Identity Manager Provisioning Manager user interfaces.

Value: As shown in the code example above.

isVirtual

Distinguishes virtual containers from real ones.

Value: *true*

childTypes

Specifies all classes in the datamodel that can be contained under this class (the class on which childTypes appears).

This setting is used the same way as for real containers. You can specify more than one class name, but each class can only appear in the childTypes setting for a single container. The values here also affect the searches done across the containers. Searches are optimized where possible to take only the classes that can exist under the container into account.

connectorMapToSame

Specifies that connectorMapTo values do not have to be provided for the class and its naming attribute (that is, their LDAP names are used). Note that these values are not important in a connector's implementation because the container is virtual and therefore does not exist on the endpoint.

Value: *true*

eTDYNContainer001Name

Specifies the container's name used by UI clients.

Value: SDK Groups

Note: For more information on real container examples involving ambiguous mappings, review some mappings generated for a JNDI endpoint by Connector Xpress.

Important Metadata Settings for a Virtual Container Class Example

The following is an example of the important metadata settings for a virtual container class taken from the metadata for the SDKDYN sample, sdkdyn_metadata.xml:

```
<class name="eTDYNContainer001">
    <metadata name="groupMappings">
      <value>
        <mapValue>
          <keyType>
            <enumValue def="SDKDYN_Groups"></enumValue>
          </keyType>
          <valueType>
            <sequenceValue/>
          </valueType>
          <mapEntry>
            <key>
              <enumValue def="SDKDYN_Groups">ROOT</enumValue>
            </key>
            <value>
              <sequenceValue>
                <baseType>
                  <enumValue def="SDKDYN_Groups"></enumValue>
                </baseType>
                <val>
                  <enumValue def="SDKDYN_Groups">GROUP_CONT_MAIN_GROUP</enumValue>
                </val>
              </sequenceValue>
            </value>
          </mapEntry>
        </mapValue>
      </value>
    </metadata>
    <metadata name="groupContents">
      <value>
        <mapValue>
          <keyType>
            <enumValue def="SDKDYN_Groups"></enumValue>
```

```
        </keyType>
        <valueType>
          <sequenceValue/>
        </valueType>
        <mapEntry>
          <key>
            <enumValue def="SDKDYN_Groups">GROUP_CONT_MAIN_GROUP</enumValue>
          </key>
          <value>
            <sequenceValue>
              <baseType>
                <strValue></strValue>
              </baseType>
              <val>
                <strValue>eTDYNContainer001Name</strValue>
              </val>
            </sequenceValue>
          </value>
        </mapEntry>
      </mapValue>
    </value>
  </metadata>
  <metadata name="connectorMapToSame">
    <value>
      <boolValue>true</boolValue>
    </value>
  </metadata>
  <metadata name="isVirtual">
    <value default="false">
      <boolValue>true</boolValue>
    </value>
  </metadata>
  <metadata name="rdnAttribute">
    <value>
      <strValue>eTDYNContainer001Name</strValue>
    </value>
  </metadata>
  <metadata name="childTypes">
    <value>
      <setValue>
        <baseType>
          <strValue></strValue>
        </baseType>
        <val>
          <strValue>eTDYNObject001</strValue>
        </val>
      </setValue>
    </value>
  </metadata>
```

```
<property name="eTDYNContainer001Name">
  <value default="true">
    <strValue>SDK Groups</strValue>
  </value>
  <metadata name="isNaming">
    <value>
      <boolValue>true</boolValue>
    </value>
  </metadata>
  <metadata name="displayName">
    <value>
      <strValue>Name</strValue>
    </value>
  </metadata>
  <metadata name="isRequired">
    <value>
      <boolValue>true</boolValue>
    </value>
  </metadata>
</property>
```

The settings for the groupMappings, groupContents, and displayName are required by CA Identity Manager and the CA Identity Manager Provisioning Manager user interfaces. See the code example for the values for these settings.

The following settings in the above example are important to consider:

isVirtual

Distinguishes virtual containers from real ones.

Value: *true*

childTypes

Specifies all classes in the datamodel that can be contained under this class (the class on which childTypes appears).

This setting is used the same way as for real containers. You can specify more than one class name, but each class can only appear in the childTypes setting for a single container. The values here also affect the searches done across the containers. Searches are optimized where possible to take only the classes that can exist under the container into account.

connectorMapToSame

Specifies that connectorMapTo values do not have to be provided for the class and its naming attribute (that is, their LDAP names are used).

Note that these values are not important in a connector's implementation because the container is virtual and therefore does not exist on the endpoint.

Value: *true*

eTDYNContainer001Name

Specifies the container's name used by UI clients.

Value: SDK Groups

Association Metadata

In addition to describing objectclasses in metadata, describe the association relationships between objects (for example, the list of groups to which a user belongs). The following are the types of associations you can use:

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

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

Direct Associations

In Direct Associations, references are persisted directly into a multivalued attribute on the endpoint. For example, in LDAP, a group's member attribute directly stores reference to the accounts it contains. Metadata of the following represents the group's member attribute:

```
<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>
  <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>eTDYNAccount</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 MetaDataDefs.java matching each metadata property's name.

If a single associative attribute can contain references to multiple objectclasses, then the "DNLdapObjectClasses" attribute should be used instead of "DNLdapObjectClass". In either case the assocRefObjectClass setting is required and needs to have the same value (or any one of the values) of the DNLdapObjectClass(es) setting.

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="assocRefObjectClass">
  <value>
    <strValue>eTDYNGroup</strValue>
  </value>
</metadata>
<metadata name="assocAttr">
  <value>
    <strValue>eTDYNMember</strValue>
  </value>
</metadata>
<metadata name="isDNAbsolute">
  <value>
    <boolValue>false</boolValue>
  </value>
</metadata>
<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. The JDBC connector expects this scheme as it matches the way relational databases model 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.

Typically a connector has one style of associations or the other, however it is possible to have both. For example, a JDBC connector uses indirect associations but supports direct associations to represent compound values.

How Metadata Is Used

The Java CS framework reads the metadata provided for an endpoint type. The framework learns the data model from the metadata for the connector and configures how each object class and attribute are processed. Object classes and attributes which are mentioned in requests to a connector instance, for which there are no *connectorMapTo** settings, are logged and discarded.

This 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 not relevant to the Java CS. You can configure the *acceptedUnknownAttrIds* property in connector.xml to list the names of attributes which are not relevant to the connector implementation, and are not logged when received. Also, the setting *acceptedUnknownAttrIds* in server_jcs.xml specifies 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 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 lets you cut and paste 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.

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 or implementing AssocIndirectAttributeOpProcessor.

The DefaultAssocDirectAttributeOpProcessor can provide reverse association logic to you for free. 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.

The AssocAttributeOpProcessorProxy can be used to provide default reverse association handling for connectors which use direct associations. The connector triggers the proxy when the connector returns *true* from its isAutoDirectAssocRequired() method. This defaults to ! isIndirectAssociations()), where the getAutoDirectAssocExlusions() method can also be overridden to name operations you want to exempt from this processing, if necessary.

Its full source file is bundled with the SDK and provides a reference for the appropriate preprocessing and postprocessing required to handle associations around each type of operation and LDAP request.

The reverse association handling service offered by the JCS covers all LDAP operations, not just computing values when querying. For example, if group.members is the value persistently stored on the endpoint, then the account.memberOf value can be automatically calculated for you by the JCS framework. In this case, creating a new account and providing memberOf values will have the side-effect of adding the new account to all the named groups automatically. Renaming and account will cause the member list for all groups referencing to be automatically updated to its new name.

If you want to use this service, but want to exclude certain LDAP operations, (for example, because they are performed asynchronously by the endpoint) then you need to override the com.ca.jcs.BaseConnector.getAutoDirectAssocExlusions() to specify exclusions for the methods listed below.

Then either leave it to endpoint logic to tidy up dangling references as required, or customize the matching association methods in your attribute style processor to handle the special requirements of your endpoint for the following example:

```
import com.ca.jcs.processor.OpProcessor.MethodName

…

importcom.ca.jcs.processor.OpProcessor.MethodName

…

publicHashSet<MethodName>getAutoDirectAssocExlusions()

{

        final HashSet<MethodName>exclusions=newHashSet<MethodName>(1);

        exclusions.add(MethodName.doDelete);

        exclsions.add(MethodName.doModifyRn);

        exclusions.add(MethodName.doMove);
        return exclusions;
```

These methods for the previous example would be:

- doDeleteAssocs()
- doModifyRnAssocs()
- doMoveAssocs()

Association Modeling

As well as representing indirect associations using DNSs, we have added the following extensions to association modeling:

- Simple key associations

  An association where the membership is expressed through an additional attribute (key). For example, members of a posixGroup are identified by a value of their uid which is an attribute of an account, rather than using account names.

- Complex key associations (nisNetgroup)

  An association where the membership is expressed through a filter expression evaluated for membership test. For example, account members of a nisNetgroup are expressed through a nisNetgroupTriple value (host, user, and domain). However, matching in the reverse direction from account to group requires using a complex filter expression.

Chapter 8: Implementing Connectors

This section contains the following topics:

How to Implement a Connector

To implement a connector, do the following.

1. Determine which values are required to be passed to the endpoint system to establish a connection.
2. Decide which LDAP attributes are used to pass these values (on the connector level of the DIT, or connector objectclass).
3. Write connector metadata, paying special attention to connection-related attributes on the connector's objectclass.
4. Incrementally write and test the related connector logic while defining the metadata.

   Note: For more information, see Create New Metadata (see page 80).
5. Decide whether connection pooling support (between connector and endpoint) is required. If using the default support built-in to the Java CS, then it is only necessary to write a class extending org.apache.commons.pool.BasePoolableObjectFactory.
6. 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 approach is the same as the approach used for the Java CS core framework where generic problems are encountered, such as adding support for *connectorMapToAmbigious=metadata* mappings for the JNDI connector.
- Write code that takes 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 approach 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

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 flexible metadata rather than static Java code drives most of the logic.

- com.ca.jcs.meta.MetaConnector—A connector translates 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 (nonhierarchical) 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 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: Defining virtual containers more dynamically in metadata is considered best practice. 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 or JNDI directory), and you want to represent this information in your connector, this class (or one of its derived classes) we recommended that you start with this class. If the endpoint system search semantics map clearly to LDAP search filters that can match multiple objectclasses, then write your connector so that it does not define isBehaviourSearchSingleClass(), as it defaults 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 strictly represent associations between objects by using a table external to both of the objects? The default value returned by this method is null, which means that metadata will be consulted it order to determine the style of association for each association (for instance the JDBC connector supports both indirect and direct styles of associations). Where the style of associations is strictly defined by the technology of the endpoint system either true or false should be returned as appropriate. For instance the JNDI connector returns "false" from this method as LDAP technology does not support indirect associations.

- **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 causes the Java CS framework to use 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.

Implementing Validator and Converter Plug-ins

The com.ca.jcs.PluginNotRequiredException exception is now deprecated in favor of the plugin class implementing the com.ca.jcs.cfg.Vetoable interface and uses the getVetoed() method to return false when this exception would otherwise be thrown.

The concrete class converter implementation com.ca.jcs.converter.meta.NullValueClassConverter (used by the JDBC connector to prune attributes which effectively have no value) is included with the SDK. You can use this class as a reference If you need to implement a converter that has to consider all attribute values for an object at once.

Representing Connector-Speak DNs

As the conventions of attribute names and values on the native endpoint may not match those of LDAP, use the com.ca.jcs.util.SimpleRdn and com.ca.jcs.util.SimpleLdapName classes to represent native DNs and their components. For example, a native DN may permit underscores or multibyte characters in attribute names. For example, database column names in the JDBC connector case, which LDAP does not support.

Exceptions

To simplify resiliency configuration, it is important to chain exceptions in your connector implementation using namingException.initCause(origEx) when wrapping exceptions thrown by native APIs. This allows you to configure retrying with a minimal set of base error messages configured (for example, chained cause may be a socket error).

For example, consider the following code snippet:

```
try
{
…native API calls…
}
catch (NativeException e)
{
final LdapNamingException   ne;
ne = new LdapNamingException(msg,
ResultCodeEnum.INVALID_CREDENTIALS);
ne.initCause(e);
throw ne;
}
```

Representing Target Objects

Each LDAP operation results in a matching method in your connector's attribute style processor being called. These methods are passed an instance of the ObjectInfo class which passes on information about the target object such as its DN (in both connector-speak and LDAP) and a class map that allows easy reference to the target object's class.

Connectors that have special requirements for handling connector-speak Distinguished Names may need to make use of the following extension points:

- Connectors can implement the ObjectInfo createObjectInfo() method in the com.ca.jcs.Connector interface, which allows a connector to add extra annotations to be added via an extension to the ObjectInfo class passed in to each of the do*() methods if desired.
- Specialized ObjectInfo extensions can implement the String getConnectorNativeName() method to return native connector-speak hierarchical names which are not in the standard comma separated format (for example, Lotus Notes Domino uses a '/' separator between naming elements)

  When this method is implemented it may also be useful to implement the ObjectInfo getObjectInfo(LdapDN ldapDn, **boolean** mapDN) method if you need to do an extra lookup on the endpoint to cache some extra state on your specialized ObjectInfo value.

The following methods also in the com.ca.jcs.Connector interface are also likely to be useful in such cases, so that JCS framework services such as reverse associations are accessible:

- convertAttributesFromConnector
- convertDNFromConnector

- convertDNToConnector
- postProcessLdapSearchResult

Also, it may be useful to implement an extension to the com.ca.jcs.converter.connector.DNPropertyConverter class and to register it for use with DN typed attributes in the connector's conf/connector.xml file with XML similar to the code shown next. Note that a plugin will displace an already registered plugin from which it extends, so the following settings cause the LNDDNPropertyConverter to displace the standard DNPropertyConverter (which it extends) registered in server_jcs.xml:

```
<property name="converters">
        <bean class="com.ca.jcs.cfg.MetaPluginConfigSuite">
                <property name="propertyPluginConfigs">
                        <list>
                        <bean class="com.ca.jcs.cfg.MetaPluginConfig">
                                <property name="pluginClass">
                                <value>com.ca.jcs.lnd.LNDDNPropertyConverter</value>
                                </property>
                                <property name="metadataPropNames">
        <list>
                                <value>DNLdapObjectClass</value>
                                <value>DNLdapObjectClasses</value>
                                <value>isDNAbsolute</value>
                                <value>DNTestExists</value>
                                <value>DNNameOnly</value>
        </list>
                                </property>
                        </bean>
```

If you need to implement such a converter then pay careful attention to the role of the DNConverterFactory which allows specialized DNConverters to extend the basic converters which are part of the JCS framework, and hence reuse much of their implementation.

Note: For more information, see Non-homogenous Association Collections. (see page 103)

Non-homogeneous Association Collections

It is possible to define a single association attribute that contains DNs for more than one object class, in which case use the MetaDataDefs.MD_DN_LDAP_OBJECTCLASSES ("DNLdapObjectClasses") metadata setting, rather then the singular "DNLdapObjectClass" setting.

When this setting is used, a DN converter implementation needs to determine the object class for each contained value. The base DN converter in the Java CS framework (com.ca.jcs.converter.connector.DNPropertyConverter) first attempts to distinguish the object class for each DN based on their connector-speak naming attributes. If there is no overlap in these for all of the classes allowed to appear in the collection, the connector developer does not need to provide any special handling. However, if there is an overlap, then the connector developer must override the com.ca.jcs.meta.MetaConnector.resolveObjectClass() method and use custom logic to return the appropriate object class for each DN. In the most difficult case (there is no syntactic clue in the connector-speak native names) it may be necessary to actually lookup the referenced object on the endpoint itself.

Style Processors

The following are the Java CS style processor types for implementing connectors:

- Attribute Style Processor (see page 106)
- Method Style Processor (see page 103)
- Scripting Style Processor (see page 105)

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. Therefore much of the content for opbindings of this style is concerned with formatting the parameters 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 following:

- The name of a native method to execute, that is, its method attribute.
- A number of bindings for each parameter the method expects.

Each parameter of type ParameterBindingType specifies:

- An attribute name
- The native method parameter name bound to it when the method is called
- The Boolean flags that specify whether the parameter is used to pass a value into the native method (input), or out of the native method (output), or both.
- The handling of multivalued attributes is addressed through the multiValuedFlattenStyle and multiValuedModifyMode attributes, where multiple values can be *flattened.* Or the collection of all attribute values could be encoded using <jcs-sdk-home>/conf/xsd/opattributes.xsd into a single string literal.

  As 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, the Java CS framework performs flattening and mapping simplifications of parameters on their behalf.

The following table shows the special values that can also be used as the value for the attribute value of any method payload parameter, in addition to the attributes mapped in the datamodel metadata for the object class targeted by a method opbinding. Using special attribute names allows 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 the native method.

Contextual Attribute	Direction	Description	Applicable LDAP Operations
NAME	IN	Target object's most nested RDN value by itself	All
DN	OUT	Target object's full distinguished name	All
ErrorStatus	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. POST delete method opbindings are special as it can be necessary to cache attribute values before the target object is deleted. However, ccaching can impose too 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.

The Scripting Style Processoris similar to method-style processing except that the opbindings are tied to executed scripts. 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 through the Java CS framework and does not require any special support from your connector. A script-style processor is only required to support opbindings for which executedDirectly=false, in which case the script generates a string of native code that is executed. For example, scripts bound to the JDBC connector can produce text containing SQL commands that are executed com.ca.jcs.jdbc.JDBCScriptStyleOpProcessor later.

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. The relative power of scripting languages allows the arguments to Java methods to be passed in as-is to the scripts, rather than defining mappings for them in the metadata, as is required for method-style processing. Currently, only script-style opbindings can be bound to the 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	createAttributeStyleOpProcessor ()
Method Style Processor	createMethodStyleOpProcessor()
Scripting Style Processor	createScriptStyleOpProcessor()

Connectors can implement one or more of the preceding styles. Where multiple styles are implemented, multiple processors being applied to a single LDAP request (as dictated by data model and opbindings metadata content) can result. 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.

The SDK connector works by persisting data to local files. Therefore 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 are written. The SDK attribute-style processor's methods have been wrapped in try and catch blocks, to demonstrate how code which that accesses a connection manager is 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, write ConnectionManager implementations as pools to provide scalability benefits. The utility class com.ca.jcs.cfg.GenericObjectPoolConnectionManager* is useful in case the endpoint system does not have connection pooling built-in. If this approach is used, 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, does most of the work.

In most cases, it can be necessary to:

- Write a small class which extends BasePoolableObjectFactory and gets and releases connections in its makeObject() and destroyObject() methods. The pool uses this method to manage the raw connections it contains.
- Write another small stub class extending GenericObjectPoolConnectionManager which contains any additional custom logic you require.

  Note: For an example of how you to code such a manager and factory, see sdk.com.ca.jcs.jndi.JNDIConnectionFactor, 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.

You can specify a custom connection/pooling management class through the connectionManagerClass property in connector.xml which names such class. The JCS framework loads this class and creates an instance of the class provided a constructor with the signature Attributes, GenericObjectPool.Config, and Logger exists. The custom connection manager constructor is given all the connection-related attributes and is responsible for initializing itself into a state where it can then create connections using those attributes.

The activate() method is invoked when your connector's LDAP interface receives its first LDAP request after the Java CS is started. The method is also invoked when the client modifies any attribute you have flagged with the isConnection metadata. When the method is invoked, it does the following:

- Looks up the attribute values using the getAttributes() method
- Establishes a connection to the endpoint system (or preferably a connection pool).

  If the endpoint system cannot be contacted or the supplied credentials are invalid, the method throws appropriate LDAP exceptions.

When the connector receives a message from the Java CS that it is about to shut down, override the deactivate() method of your connector to perform any cleanup routines that you want to perform inside your connector. For example:

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

Connector Opbinding Support

Any connector allows you to define opbindings for all top-level operations implemented by its AttrbuteStyleProcessor. Defining opbindings allows you to customize connector behavior through JavaScript payloads, or if your connector supports a method-style processor, such as JDBC stored procedures.

However, to call opbindings in all circumstances, it is important that any calls your attribute-style processor makes to its own methods are invoked through the following:

- proxiedSelf.method(...)

  For methods defined on the com.ca.jcs.processor.AttributeStyleOpProcessor interface. This attribute is defined in the com.ca.jcs.processor.AbstractAttributeStyleOpProcessor abstract base class.

- proxiedAssocSelf.method(…). For methods defined on:
  - com.ca.jcs.processor.AssocAttributeOpProcessor interface.

    In this case, this attribute is defined in the AbstractAttributeStyleOpProcessorAssocDirect base class.
  - com.ca.jcs.processor.AssocIndirectAttributeOpProcessor interface.

    In this case, this attribute is defined in the com.ca.jcs.processor.AbstractAttributeStyleOpProcessorAssocIndirect base class.

For example, a call directly to doAdd(…) in your attribute-style processor, (such a call can occur in the implementation of the doModifyRn() method) bypasses any registered opbindings, whereas a call to proxiedSelf.doAdd() with the same arguments executes any registered opbindings.

In addition to writing code that defines all possible opbindings, be aware of relevant configuration such as *allowMetadataModify* settings.

Note: For more information, see Connector.xml Files (see page 65)

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 has validated and stored the metadata you provided for the endpoint type. For static endpoint types, the metadata is read from within the connector's *.jar* as configured through the contained *connector.xml* file. For dynamic cases, the metadata is 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 created, that contains all 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. For example, 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 69)

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

This section contains the following topics:

Implementing in Java or JavaScript Considerations

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

- Almost anything that can be implemented in Java can be implemented in JavaScript, therefore, the relative power of each approach is not alarge consideration. In particular, note that the Java CS scripting support allows streaming of search results through the use of the *searchResultsBlockingQueue* scripting variable. The SDKSCRIPT connector demonstrates this.
- 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 for larger production connectors, consider Java.

- As the Rhino 1.7R1 JavaScript engine used by the JCS does not support embedded debugging, the primitive approach of using trace messages is the only option available to debug your scripts at this time. This approach can prove prohibitive if your scripts become too long and complicated, unless they are composed of sections of script that have already been independently tested and verified.
- Where minor customer-specific customizations of an existing connector are the focus, scripted opbindings are a good option.
- If most of the logic for a connector depends on fairly simple textual manipulation, consider using scripts. For example, preparing specially formatted arguments to be passed as command-line arguments to existing native endpoint system executables.

- 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. Writing a few specialized classes and referencing their names in a new connector.xml can be all that is required. In such circumstances, implementation code can be shared even if a different LDAP schema is 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. However the cost is minimal as the Java CS helps ensure that scripts are only compiled once and maintained in a pool for fast reuse. Of more concern is that the loose type-checking in most scripting languages (including JavaScript) can mean that problems picked up by the compiler in Java are only discovered later during execution.
- You can start a connector as a scripted solution and later migrate it to Java. For example, if the connectors code grows beyond initial expectations and becomes hard to maintain and or nonperformant.

Important! 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. To avoid script corruption, use CDATA sections as demonstrated in sdkscript_opbindings.xml. If for some reason you cannot use CDATA sections, then use the correct quoted characters in your script text instead. For example, 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. The version used by JCS 1.7R1 is later than the bundled version (1.6R2), as the bundled version is deficient in regards to exceptions thrown from your JavaScript scripts.

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. You can define one or more global scripts 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 are passed an ObjectInfo instance as its first argument, and an Attributes object as its second argument. This is because these are the arguments to com.ca.jcs.processor.OpProcessor.doAdd(ObjectInfo, Attributes). Use this approach for all but the simplest scripts, as it allows reuse of utility functions between multiple scripts.

   Note: For more information about the arguments passed to other methods, see the JavaDocs for the com.ca.jcs.processor.OpProcessor interface. For an example of a 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) are called with two scripting variables defined:

   - objInfo is bound to a Java object of type ObjectInfo
   - attrs are 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 are then passed back to the client. If strictCompletion is true, the LDAP operation to fail. In JavaScript, it is better to throw an exception than to use this variable, as there is better control. For example, an LDAP error code can be assigned to the exception. The SDKSCRIPT connector has a number of examples where LDAP exceptions are thrown, for example, where a LdapServiceUnavailableException is thrown in sdkscript_opbindings.xml.

- 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 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 required. For example, a script can verify that a single-valued attribute currently has no values before adding a new value for it.

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

Only script opbindings that have 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. The Java CS framework invokes all other opbindings without any special support being required from the target connector.

Exception Handling In Scripts

The JCS framework intercepts any JavaScript exceptions which are thrown that include line numbers, and helps ensure that the line number relative to the start of the opbindings document. That is, the value of the namespace.eTOpBindingsMetaData attribute are also included in the exception text.

A useful technique when encountering errors in a JavaScript script is to put an exception breakpoint on the org.mozilla.javascript.JavaScriptException base class used by Rhino. Walking up the stack of such an exception often provides some context about where and why the script is failing.

Scripted Opbindings Debugging

As the tools for debugging embedded JavaScript (whether individual opbindings or as part of a completely scripted connector) are not currently offered as part of the Rhino project, a number of methods like ScriptStyleOpProxyHandler.debug(String message, Object obj, boolean dumpStack, int pos) have been added.

These methods let you can invoke them from your script and therefore allow you to put breakpoints on these methods and analyze state during script execution. In addition to acting as potential breakpoint targets, these methods also output tracing output.

The "pos" parameter is intended to allow you to flag related calls with the same value, and allow triggering of conditional breakpoints so that calls which are not interesting during a debugging run are easily skipped.

You can try this approach for setting breakpoints out against the SDKSCRIPT connector which includes example debug calls.

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 through 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 are executed asynchronously, by setting a*synchronousSearch=true* on their guards. When the property is set to true, the triggered script must queue search results by adding them to the queue bound to the searchResultsBlockingQueue scripting variable, and 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 process the query, rather than the target connector being wrapped. 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*. (see page 129)

- 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 can 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 Map is used, they must be compatible with the signature Map<String, Object>.

  Note: BaseConnector.CONN_NAME ("!nameId!") 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.
  - Process search results 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 percent scripted connector.

If you do not want to use a ready-made connector, you can create your own pure scripted connector using a templates provided by Connector Xpress. You can use the following two templates to create a pure scripted connector:

- SDK DYN Script
- SDK DYN UPO Script

Using this template as a starting point, you can invoke your own JavaScript functions for each mandatory operation. This functionality relies on the 'instead of' operation binding. Before the Java CS performs any operation, it checks to see whether there are any operation bindings that tell it to invoke some logic before, after, or instead of the operation. Use the 'instead of' operation binding to invoke a JavaScript operation for your own pure scripted connector.

You must create an 'instead of' operation binding for each of the mandatory operations; Add, Delete, Modify, Search, and Lookup.

You can 'hot deploy' a pure scripted connector, which means that you do not have to change the Java Connector Server for the new connector to become operational.

A hot deployed connector specifies its connector.xml content as part of its metadata 'connectorXML' at the namespace level (accounting for proper XML encoding of this value). This means that a scripted connector can be created on the fly on a running Java CS without needing to restart it, or adding a static connector.xml on the Java CS host. Also, any connector configuration changes that are typically specified in connector.xml can be made active without a Java CS restart such as the connection pool settings.

Scripted Logic Update Considerations

Consider commenting out the *staticMethodScriptStyleMetaDataFile* property from your connector's connector.xml file while writing or enhancing scripts. 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 the following:

- 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, consider reinstating the *staticMethodScriptStyleMetaDataFile* property setting in connector.xml, therefore stopping changes to the endpoint type's *eTOpBindingsMetaData* attribute which can subvert your connector's implementation. If you are using a hot-deployed connector that has its connector.xml settings in metadata rather than in a file residing on the JCS, this procedure is not recommended.

Hot-deploying Connectors

You can create and hot-deploy a new pure scripted connector from Connector Xpress. For example, you can deploy the SDKSCRIPT to any Java CS using its associated Connector Xpress template instead of installing the connector on the Java CS through the Java CS installer. These types of connectors require configuration, however you do not need to configure the connector on the Java CS side through connector.xml. Typically, the Java CS is told about a connector type being implemented by registering it in connector.xml and restarting the Java CS.

A hot deployed connector specifies its connector.xml content as part of its metadata 'connectorXML' at the namespace level (accounting for proper XML encoding of this value). This means that a scripted connector can be created on the fly on a running Java CS without needing to restart it, or adding a static connector.xml on the Java CS host. Also, any connector configuration changes that are typically specified in connector.xml can be made active without a Java CS restart such as the connection pool settings.

Example: Sample hot-deployed connector.xml

```
<metadata name="connectorXML">
  <value>
            <!—
Contains contents of connector.xml that can be dynamically applied
to the connectorType and all live connectors via namespace metadata update
-->
      <strValue>
        <![CDATA[
        <?xml version="1.0" encoding="UTF-8"?>
        <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
            "http://www.springframework.org/dtd/spring-beans.dtd">
```

```
<beans>
  <bean class="com.ca.jcs.ImplBundle" id="sdkscript">
    <property name="name">
      <value>SDKSCRIPT</value>
    </property>
                                        . . .
    <property name="connectorTypeName">
      <value>SDK Script DYN Namespace</value>
    </property>
                                        . . .
    <property name="defaultConnectorConfig">
      <bean class="com.ca.jcs.stub.StubMetaConnectorConfig">
                                                . . .
      </bean>
    </property>
  </bean>
</beans>
]]>
            </strValue>
  </value>
</metadata>
```

Chapter 10: Endpoint Objects

This section contains the following topics:

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 is not routed through this method, but handled instead by the Java CS framework, which calls your connector's constructor and activate() methods.

In some cases, it is necessary to define and register a specialized com.ca.jcs.processor.ConnectorAttributesProcessor to handle the attributes stored for the connector. For example, if you want to support a virtual attribute which has a value calculated by the code.

If your method creates an object that exists, throw an LdapNameAlreadyBoundException.

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 verifying that an object with the same name does not exist before trying to add the object.

Update the endpoint system to record the object's creation, given a reference to it (objInfo) and the attributes are 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 can 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.

It is necessary to persist the provided attributes on the endpoint system. For example, a JDBC connector would translate this list into the column names and values in an SQL INSERT clause 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 handles associations then get the list of associations and handle adding them. The following code snippet shows an example:

```
// 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, issue separate creation statements or a single statement listing multiple parameters.

Add Operation Testing

At this stage, verify that you can 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 step 1 and 2 and then performs multiple instances of steps 3 and 4. For your initial development work, drive simple creations from an LDAP client. However, once your add operation is complete you 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 exists, access permissions are insufficient, or other appropriate endpoint failures cases.

Use the scripted tests for creating connectors and directories you developed earlier to construct a working environment for these test cases.

Chapter 11: Removing Endpoint Objects

This section contains the following topics:

How you Delete an Object

Removing an endpoint object can 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)
```

Verify that the object you want to delete exists. To verify that the object exists, 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, 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 used by the connector contain insufficient privileges to perform a deletion.

Note: For more information, see, org.apache.directory.shared.ldap.exception.LdapNoPermissionException.

Write your code to account for a possible transient condition, such as a communication exception. In this case, throw the following exception: org.apache.directory.shared.ldap.exception.LdapServiceUnavailableException.

If necessary, perform subsequent cleanup on any other objects that contain references to the object that you deleted. For example, membership references to this account can exist in other group objects.

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

If possible, use the AttributeStyleProcessor doSearch(ObjectInfo info) and doDelete(ObjectInfo info) methods as a basis for your own custom logic if the association handling logic built into the JCS is not sufficient. However if your connector uses any structural converters, we recommend that you carefully examine the format of the search results returned by calling doSearch(). In particular, the search results have relative (rather than absolute DNs) and are in connector-speak. However, if structural converters are used, use com.ca.jcs.meta.MetaObjectClassMapping.unflatten() before changing attribute values and flatten() after changing the values.

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 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 method following 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):

com.ca.jcs.assoc.DefaultAssocDirectAttributeOpProcessor.doAssocUpdateReferencesTo()

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, verify that you can 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, we recommend that you drive simple creations from an LDAP client. However, once the delete operation is complete, 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.

  Use the scripted tests for creating connectors and directories you developed earlier to construct a working environment for these tests cases

Chapter 12: Querying Connector Objects

This section contains the following topics:

How You Search for an Object

The Java CS framework provides an abstract class AbstractAttributeStyleProcessor which your connector can implement. 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;
```

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 are mapped to the connector by the time these methods are called. If you provided *connectorMapTo=values* in your connector metadata file, all values are in connector terminology.

The Java CS Framework runs any required validators or converters before you invoke this method, that is, all data is in connector terminology.

Note: The childTypes metadata setting for the top-level eTDYNDirectory class and all containers classes is critical in driving the behavior of the lookup and search operations – if a class name is missing then objects of this class will not be found.

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: The ApacheDS SchemaService calls this method on your connector to sanity test MODIFY and other operations. Therefore, implement it as one of the first operations for your connector, before everything except, perhaps, the ADD operation. Also, if 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 yields 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 are 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

   Write your connector to 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 necessary, 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).

For an example, see LDAPFilterToFileSearchVisitor in the SDK sample.

The com.ca.jcs.filter.SimpleLDAPFilterToMapVisitor provides a useful utility visitor for connectors that only implement basic filter semantics, and utility methods like com.ca.jcs.filter.FilterUtil.toSimpleMap(FilterInfo) use this visitor under the covers to convert a FilterInfo into a simple map of attribute values mentioned in the filter. Connectors making use of such visitors could probably also benefit from using the isConnectorFilterable Boolean metadata setting on any attributes that can appear in customer-provided filters but which the connector cannot process the filtering for. When it is used, the Java CS framework automatically revaluates search results returned by the connector against the provided filter, and throws away any that do not match it before they are returned to the client. This post-filtering has no negative impact on search result streaming (if implemented by the connector), but does incur a slight performance impact.

1. 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. 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. The Java CS also does not return attributes 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, typically set using your connector's conf/connector.xml file.

2. 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. AppendingNamingEnumeration can be used to wrap a prepared collection of results or multiple subenumerations are stepped through in order.

3. Streaming search support

   The ApacheDS and Java CS frameworks support a streaming search mechanism, which is marginally harder to implement but has 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 stream search, implement your own NamingNumeration which processes and returns each SearchResult object one at a time, and return this NamingNumeration from doSearch(), rather than caching all search results in memory before 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). For an example, see com.ca.jcs.sdk.SDKSearchEnumeration.

In some cases, it can 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 can 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 nonstreaming search for possibly better runtime performance. Use of this threshold approach assumes there is a way to determine efficiently the number of results a search can possibly return or did actually return, or somehow tracking the total number of objects of a type that can exist based on a rough calculation during connector activate() and keeping running totals.

The streaming search can 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 nonstreaming implementation working first. When debugging search problems, the following setting in *jcs-home* /conf/log4j.properties can 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 it is necessary 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

Your connector and directory object load correctly after implementing the doSearch method. 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 uses them), by providing filters of the form (*objectclass=<class1>*) for each object type. One level searches are 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 you want to return.

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

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 specifies a target attribute and new attribute values. For multivalued attributes, a specified mode indicates exactly how the target attribute is modified. For example, whether new values are 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[]) 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 modified. getModificationOp() can be used to indicate what operation is performed (like ADD, REPLACE, or REMOVE) and getAttribute() provides the attribute details.

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

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.

The Java CS framework runs any required validators or converters before you invoke this method.

The forceModificationMode metadata setting can significantly simplify coding of modification logic for multivalued attributes in many cases. The setting normalizes all modifications to either of the following, according to the requirements of the endpoint system with which your connector interacts:

- 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").

For example, 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. As a consequence, its code is simplified considerably.

Note: See the JCS Javadoc for information about the MetaDataDefs.MD_FORCE_MOD_MODE constant.

Modify the provided attributes 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 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, verify that you can do the following:

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

For your initial development work, drive simple creations from an LDAP client. However, when your add operation is complete 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 does not exist, access permissions are insufficient or other appropriate endpoint failures cases

Use the scripted tests for creating connectors, directories and endpoint objects you developed earlier to construct a working environment for these test cases.

Chapter 14: Implementing Associations

This section contains the following topics:

Associations

There can be a requirement to create associations between the managed objects in your endpoint system. It can be necessary 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, implement com.ca.jcs.assoc.AssocAttributeOpProcessor which has the following methods:

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

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

Defining Associations

Define associations between objects 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, update the the group members attribute in all Account objects which are the members of the group 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 doModfyRn. 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

Be aware of the reverse association support offered to connectors which use 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 have in connector terminology. To help you understand how representation is configured and implemented, the source code for com.ca.jcs.converter.connector.DNPropertyConverter (which handles these conversions) is bundled with SDK.

The following metadata settings and their impact on the DN conversion are important to understand:

- processMetaDataDefs.MD_DN_NAME_ONLY ("DNNameOnly")
- MetaDataDefs.MD_DN_TEST_EXISTS ("DNTestExists")
- MetaDataDefs.MD_DN_LDAP_OBJECTCLASS ("DNLdapObjectClass")
- MetaDataDefs.MD_DN_IS_ABSOLUTE ("isDNAbsolute")

The following settings can be used for ambiguous properties where multiple objectclasses can apply:

- MetaDataDefs.MD_DN_LDAP_OBJECT_CLASSES (“DNLdapObjectClasses”)
- MetaDataDefs.MD_DN_NAME_ONLY_LDAP_OBJECTCLASS (“DNNameOnlyLdapObjectClass”)

Chapter 15: Renaming Endpoint Objects

This section contains the following topics:

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. ObjInfo.getObjectClassMapping().getConnectorClassName() specifies the type of object being renamed. The old object name is objInfo.getName() and the new object name is newRn.

To perform the equivalent of a rename operation, modify the appropriate endpoint objects. If your endpoint does not support a rename operation, simulate the behavior by doing the following:

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

If your endpoint does not support renaming objects, write you connector so that it raises an LdapNotImplementedException.

The JDBC connector supports both the direct rename operation and the simulated rename which involves delete and add operations. In this case the MD_IS_RENAME_VIA_DELETE_ADD (“isRenameViaDeleteAdd”) metadata is used to signify which rename style is requested per class.

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 that is 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 can require 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 all objects referencing the target object renamed, through associations. For example, Groups referencing an Account which is being renamed. The method 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 verify that you can rename all managed object types. Write test cases that cover renaming all managed object types, and failure scenarios such as renaming nonexistent objects, or renaming an object using the name of an existing object.

Chapter 16: Moving Endpoint Objects

This section contains the following topics:

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. indicated by

The objInfo.getObjectClassMapping().getConnectorClassName() method indicates the type of object being renamed. 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.

Modify the appropriate endpoint objects to perform the equivalent of the move operation. If your endpoint does not support a move operation, simulate the behavior by following these steps:

1. Call doLookup to retrieve the existing attributes.
2. Call doAdd to create an 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, write your connector so that it raises 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 can require 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 moved, through 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. Write you test cases to cover moving all managed object types and testing failure scenarios like moving nonexistent 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

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. For example:

```
jcs-connector-*.zip
```

Match the short name of the implementation .jar containing the new connector, to the short name of the connector, for example:

```
lib/jcs-connector-abc.jar
```

Important! The zip file contents are extracted directly into the installation folder during the install process. 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 be registered with the server at install time with no coding effort.

The installer reads _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.

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 sdkscript.

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. Verify that that the jcs-connector-abc.jar and any dependent jars are in the /lib folder, deployed documentation files are in the /doc folder, and static configuration files are 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.* This deployment lets you test against a development Java CS before deploying using a .zip to a more formal Java CS, as described previously.

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

Hot-deployment

You can hot-deploy a custom connector through a client of the Java CS. With hot deployment, a connector does not need to specify a connector.xml or pop ldif file, as it is created purely from the metadata (and any operation bindings or scripts).

Hot deployment is only applicable to bundles known to Java CS before hand such as a scripted connector using DYN namespace.

Note: For more information, see Hot-deploying Connectors (see page 118).

Appendix A: Testing with JMeter

This section contains the following topics:

JMeter

Apache JMeter is an open source Java component testing application designed to load test functional behavior and measure performance. Apache JMeter 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/, 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 web page 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 test cases 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 about 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 file (like *sdk_core_basic*). When saving a file, click 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 can spawn multiple threads.)

Next, the View Results Tree and View Results in Table listeners record output from each test step with timings, followed by an Include Controller, and a standard JMeter controller which allows one .jmx file to import another.

In this case, the jcs_global_vars.jmx file is imported. This file is used to pass on a number of variable assignments from the top-level build.xml, used by all component tests (which port to use / the absolute ${DIST} path and similar). When 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 use 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. Controllers are used widely in the component tests to read in data model and opbindings metadata from local files, making it possible to share them between multiple tests and edit them more easily.

The controllers assign these variables in a prepass 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 example, to skip a node deleting all objects after the test case has run, to allow some post-mortem examination of their exact state. The first nodes typically create the parent endpoint type (where the metadata is one of the attribute values that nonstatic connectors require) 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 it expects.

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

The simplest response assertions simply verify 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, in cases where you are verifying the result of a test step which you expect to fail, select the Ignore Status check box. 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 simply containing it. The Not modifier inverts the check too *not Contains* or *not Matches.*

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

Using an Equals test makes it easier to write comprehensive test plans quickly, however using the test also means that the cause of a failure can be harder to determine. Use more specific test cases earlier in your test plan to gain confidence in determining correct behavior. Use coarser tests such as an Equals to verify a whole subtree search result later.

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

Extensions to JMeter

We have made the following custom extensions to JMeter:

- Support for "Equals" response assertion. This support also required an extension to make search results stable (that is, ordered alphabetically by search DN and attribute id) so that the Equals tests make sense. However, when more search results are returned than the limit configured using ldapsampler.max_sorted_results=2000 in jmeter-home/bin/jmeter.properties, sorting is disabled.
- Variable From File Controller was added, so that the contents of a named file can now be read into a specified JMeter variable for use in later test steps.
- Jmeter was modified to allow multiple attribute values to be specified in a modify request through the following XML semantics.

  A Jmeter modify request can take the following as a modification value:

  - <list><value>value1</value><value>value2</value><value>value3</value></list>

The following symbolic strings were introduced so that they can be used instead of their cryptic numeric forms:

LDAP Extended Request: Search Test: "Scope" Field (scope for LDAP search operation)		
String in JMeter Test	Numeric Equivalent	Java Constant (in javax.naming.directory.SearchControls)
object	0	OBJECT_SCOPE
onelevel	1	ONELEVEL_SCOPE

LDAP Extended Request: Search Test: "Scope" Field (scope for LDAP search operation)		
subtree	2	SUBTREE_SCOPE

LDAP Extended Request: Modification Test: "opCode" Column (mode for LDAP MODIFY operation)		
String in JMeter Test	Numeric Equivalent	Java Constant (in javax.naming.directory.DirContext)
add	1	ADD_ATTRIBUTE
replace	2	REPLACE_ATTRIBUTE
remove	3	REMOVE_ATTRIBUTE

Support for single modification items with multivalued attributes have been added where multiple values can be provided as a single string for a multi valued attribute, as shown in the following:

- <list><value></value></list> syntax:
- <list><value>value1</value><value>value2</value></list>

Run a JMeter Test Case

To run a JMeter test case

1. Verify 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*) do not need external endpoints. 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 the node named View Results Tree and then select Run.

   Each test step is listed. To display results, click the result.

   If the test failed it, appears in red.

If an Equals tests on large test responses fails, cut-and-paste the expected and received lines of text to two lines of a temporary file. For example, a failure that occurs when comparing the entire results of a wide-ranging search request. Then, working backwards from the end, insert spaces in the shorter string until the later text realigns 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 you cannot find a reason, then debug the connector until the strings agree. Otherwise, cut-and-paste the response from the View Results Tree to the matching response assertion to record that this behaviour is now correct.

CA extensions to JMeter LDAP Extended Request Sampler, such as sort search results and the order of attributes within them, 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, only perform a final verification of response assertions for a test after the previous attempt has completed, so that all created objects are cleaned up.

Editing Test Files

When editing test files, we recommend that you save them outside of *jcs-sdk-home* /build/. Save the files 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.

When you have completed your edits and verified that everything is working, do the following:

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: 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 after saving. 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: For more information, see *jcs-sdk-home*/README.txt

Appendix B: Frequently Asked Questions

This section contains the following topics:

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 named 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. Configure it automatically using Spring XML by changing the class name specified to the matching <bean class=" construct in connector.xml. Then, and add extra XML to set the values of the fields you added to your JavaBean.

Note: For more information, see Connector.xml Files.

What is connectorTypeClass?

This field specifies the ConnectorType class for storing data model and opbindings XML metadata documents, and to act as the parental container for all connector instances which are driven them. Connector implementations rarely (if ever) 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). 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?

You can find information about the attributes in the Java CS Javadoc for com.ca.commons.datamodel.MetaDataDefs for the Java CS and com.ca.commons.datamodel.DataModelDefs for JIAM, where constants are defined for each standardized setting. You can also add extra per-connector metadata settings, after you have 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.commons.datamodel.MetaDataDefs 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 likely to need additional settings from the Java CS Javadoc for com.ca.commons.datamodel.MetaDataDefs 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.commons.datamodel.MetaDataDefs.html#MD_CONN_MAP_TO_AMBIGUOUS com/ca/commons/datamodel/MetaDataDefs.html#MD_CONN_MAP_TO_AMBIGUOUS and com.ca.commons.datamodel.MetaDataDefs.html#MD_CONN_MAP_TO_AMBIGUOUS_CHOICE_ATTR http://lod1218.ca.com/identitymanager/r12.5/IM_HTML/javadoc-jcs/com/ca/commons/datamodel/MetaDataDefs.html#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.

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 configured using conf/connector.xml - for example the SDKDYN sample connector sets it to true as follows:

```
<property name="caseSensitive">
    <value>true</value>
</property>
```

Also note that the SDKFS connector demonstrates the case-insensitive case, which means that all native class and attribute names are converted to lowercase before performing lookups of associated metadata. In the case sensitive case class and attribute names returned by the endpoint system must exactly match the names provided in the metadata document.

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 are executed. Never set validateToConnector=false outside of development because a false setting turns off all validation of LDAP information being passed to your connector.

validateFromConnector defaults to false. If you suspect bad data either preexists or is being written to the endpoint system by another interface, in which validation is performed on query results before they are returned to the client, you can set validateFromConnecto to true.

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

The Plug-In Classes (see page 43), 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. You can then 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 111).

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? The setting must match, otherwise the Java CS does not know which ImplBundle to use to create an appropriate connector instance. As a result there are 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 objectClasss (skipped 0)

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

      If your connector does not appear in this list and you are running within a Java IDE like Eclipse or IDEA, then add your connector's jar file to the Java CS's classpath.

c. 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/. Verify that it contains a valid /conf/connector.xml file. For more information refer to the structure of the SDK connector's jar file.

Why does the Java CS appear to execute without triggering break-points in the debugger?

In some circumstances, the Java CS has trouble shutting down. 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. 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?

The Java CS use of Java proxies in its implementation complicates setting exception breakpoints in your IDE.

If you observe an exception but then find a matching exception breakpoint that is not triggered as you expect, try setting a breakpoint on InvocationTargetException, which can wrap 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) 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 access your extra custom metadata settings, you create 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 the following circumstances:

- When attributes are passed from doLookup() or doSearch() with no values, the ApacheDS LDAP codec is affected. In this case, do not include the attribute names should not be included, 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 singles 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) 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 these cases, the ApacheDS filter parsing code does not fail gracefully but displays an error message to stdout (and may therefore not be noticed). ApacheDS throws an exception which causes the thread executing the current LDAP request to never return to the client.

  Note: If you observe this behavior, contact CA technical support.

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

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 is sufficient to override the convertAttributesFromConnector() method in your connector. Verify that you call super.convertAttributesFromConnector() !) to handle what ever logic in required, as this method is called by com.ca.jcs.meta.MapSearchResultsFromConnectorEnum for each search result to map them to LDAP.

We recommended that you add you virtual attributes first and then call super.convertAttributesFromConnector() afterwards, so that you can keep your logic free from referencing 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 do not want my code to refer to LDAP objectclass names.

Use the metadata setting com.ca.commons.datamodel.MetaDataDefs.html#MD_CONN_MAP_ALIAS http://lod1218.ca.com/identitymanager/r12.5/IM_HTML/javadoc-jcs/com/ca/commons/datamodel/MetaDataDefs.html#MD_CONN_MAP_ALIAS to specify a short alias for your objectclass (say connectorMapToAlias=account), and use the metadata setting as the discriminator in your code. We recommend that you use your own utility method to look up the value of this metadata setting where required (see previous question for tips).

What does an exception 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 LDAP schema (in this case, connectors/lnd/conf/eta_lnd_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.

Why is the Provisioning Server not behaving as expected?

If you enter non-valid XML code directly into the Provisioning Directory, the Provisioning Server or Provisioning Manager may no longer work as expected.

What's the impact of using integer types in java connectors?

When you use an attribute mapped as an integer, for example, when you add an integer typed field to an account in Connector Xpress, the integer values that your connector receives may end up padded. This can effect minimum and maximum field length validation in the CA Identity Manager Provisioning Server.

This is because the CA Identity ManagerProvisioning Server pads the value. For example, for a client sending 22, JCS receives the following:

Type : 'eTDYN-int-c-01'

Val[0] : 0000000022

For a client sending 022, JCS receives the following:

```
Type : 'eTDYN-int-c-01'

Val[0] : 0000000022
```

Although the Java CS unpads the values, in this example the Java CS cannot determine if 022 or 22 is sent. The unpadding algorithm unpads the value up to the minimum length you specified, so 0000000022 becomes 022 for a specified minimum length of 3.

This occurs for all connectors when you map an *int* datatype. We recommend that you do not map to *-int* and use *–str*, unless, for example you use a capability attribute.

Appendix C: Debugging Tips

The following are suggested breakpoint locations for debugging custom connectors. These breakpoints are listed in order, starting with locations that are nearest your custom connector to locations 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 your connector-specific code uses them.
- LDAP methods within MetaConnector, for instance MetaConnector.add(). Although the arguments to these methods have been normalized, they are still in LDAP terminology and can provide clues if you are having problems during the name mapping, validation, and conversion phases.
- 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 PartititionLoaderService (say modify()), but it does not make it to the corresponding call on MetaConnector (say modify()) then put 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())
- The search() method returns NamingEnumerations. Some streaming varieties mean that results are not retrieved at the time that the search() method returns. Instead they are retireved some later time when the ApacheDS framework steps through the entries in the returned enumeration. When problems occur while stepping through search results, the most interesting breakpoint 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 ApacheDS processes them. This class acts as the front end to the Java CS. It triggers the activation of new connector instances when DNs 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.

- When porting a C++ connector to the Java CS, you can compare the objects and attributes found in each. To do compare the objects, explore the same endpoint in each connector, and then connect to the ETADB. Delete everything under the connector level in ETADB and restart your Provisioning Server. Also, remove systemdb in the Java CS before starting it.
- If you want to clean up 'systemdb' on a regular basis, during development, use the secret C++ Connector password in both the Provisioning Manager and the Java CS. Using the password saves time in resetting the password. To reset the password manually, bind to a running Java CS with the secret password and set the userPassword on the uid=admin,ou=system object, where it is saved to systemdb.
- For scripted connectors methods invokeFunction and invokeScript are places suitable for debugging execution of scripts and serve as the Java to script language boundary. The exception message generated by Rhino usually includes the file and the line number where the problem occurred.
- Setting exception breakpoint org.mozilla.javascript.JavaScriptException for debugging scripted connectors is a useful way to catch Rhino script execution failures.

Appendix D: Connector Review Checklist

This section contains the following topics:

Checklists

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 (CCS). The Java CS is architecturally and functionally similar to the CCS, except that it is implemented in Java rather than C++. Consequently this allows you to write your connectors in Java. In addition, to the extent to which it is possible the Java CS is data-driven rather than code-driven, which allows the container (i.e. Java CS) to do much of the connector's work for it.

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

Note: For the most current technical information, see the JavaDoc included with the JCS SDK install. It may be slightly more up to date then the JavaDoc integrated with this Guide.

Holistic Design Considerations

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, it is necessary to answer *yes* to one of the following two questions:
      - Is the Java connector completely deprecating the C++ connector? In this case the Provisioning Manager plug-in does not 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 changes, has C++ connector been updated to match the changes?
   c. Have you listed any migration steps (including migration steps as a result of step b) required to migrate C++ connector customers and confirmed where the steps 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 selects the function 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 use 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 the third-party files and how to install them? Installing third-party files usually involves copying jars to *jcs-home*/lib and recycling the Java CS.
   - Does the connector depend on JNI (Java Native Interface) support, directly or indirectly?
8. Does the connector depend on any special configuration on the Java CS server that you cannot work around using a URL scheme for connection details? Have the details of any environmental preconditions regarding third-party software installation been documented?
9. Does the connector impose any operating system requirements on its host Java CS?
10. Is there a requirement that the connector supports a notion of custom attributes, which are mapped to native attributes by the customer after deployment? If so, we recommend that they are 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 ofinformation? 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 use a plug-in to the Provisioning Manager, outside of the plug-in to the CCS for the connector? Have the details about 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 requirements?
   - Have class header comments (especially on the core classes) explaining requirements and gotchas been included?
   - Have method comments, especially important for methods that are confusing, been included?

     Note: Some leeway in documenting only one of get or set method for a property is acceptable.
   - Are package.html files on subpackages 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, synchronization of activate and deactivate calls, and such)?
8. Has some testing been performed where multiple threads access the connector concurrently?
9. 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. For static C++ options that have been ported, there is typically, an 80-90 percent code reduction. 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 verified 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?

5. Do all attributes requiring secure handling such directory and account passwords have secure metadata settings?

   isWriteOnly=true means that the attribute value can only be written and not read back and should be used on attributes containing sensitive data, unless there a requirement that they can be queried.

6. Is metadata and opbindings modification through LDAP MODIFY requests allowed for this connector? If not, then allowMetadataModifyGlobally (server_jcs.properties) and allowMetadataModify in connector.xml can be used to lock down the connector with respect to metadata changes.

   - allowMetadataModifyGlobally can be set in the server_jcs.xml and can disable all metadata modifications server wide.
   - allowMetadataModify is set on a per connector basis and can override the server setting.
   - Enabling metadata modifications means that metadata can be updated from time to time, when connector is up and running. However, it can be beneficial to keep it locked down which means no metadata changes are allowed until the flag is reset again.

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. Include the reason that the same connection is used for the whole search. For example, this can 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 use them?
7. Are all attributes that can be returned from an object scope search also supported for one-level and subtree cases?

   Note: This approach is highly recommended.

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 retriable 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) grow in strict tandem with the functionality of the connector. A useful maxim is that any connector functionality not covered by automated component tests cannot be considered to exist. Even if manual testing proves the connector works today, this may not be adequate when the connector is 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 multivalued 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, modifyrn on nonexistent object, adding an existing object)?
4. If the connector requires multiple flavors of connection, is there enough coverage of different supported connection schemes?

Index