Novell exteNd Composer

5.2

www.novell.com

LDAP CONNECT USER'S GUIDE

Novell®

Legal Notices

Copyright © 2004 Novell, Inc. All rights reserved. No part of this publication may be reproduced, photocopied, stored on a retrieval system, or transmitted without the express written consent of the publisher. This manual, and any portion thereof, may not be copied without the express written permission of Novell, Inc.

Novell, Inc. makes no representations or warranties with respect to the contents or use of this documentation, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Further, Novell, Inc. makes no representations or warranties with respect to any software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to makes changes to any and all parts of Novell software, at any time, without any obligation to notify any person or entity of such changes.

This product may require export authorization from the U.S. Department of Commerce prior to exporting from the U.S. or Canada.

Copyright ©1997, 1998, 1999, 2000, 2001, 2002, 2003 SilverStream Software, LLC. All rights reserved.

SilverStream software products are copyrighted and all rights are reserved by SilverStream Software, LLC

Title to the Software and its documentation, and patents, copyrights and all other property rights applicable thereto, shall at all times remain solely and exclusively with SilverStream and its licensors, and you shall not take any action inconsistent with such title. The Software is protected by copyright laws and international treaty provisions. You shall not remove any copyright notices or other proprietary notices from the Software or its documentation, and you must reproduce such notices on all copies or extracts of the Software or its documentation. You do not acquire any rights of ownership in the Software.

Patent pending.

Novell, Inc.
404 Wyman Street, Suite 500
Waltham, MA 02451
U.S.A.

www.novell.com

exteNd Composer *LDAP Connect User's Guide*
June 2004

Online Documentation: To access the online documemntation for this and other Novell products, and to get updates, see www.novell.com/documentation.

Novell Trademarks

ConsoleOne is a registered trademark of Novell, Inc.
eDirectory is a trademark of Novell, Inc.
GroupWise is a registered trademark of Novell, Inc.
exteNd is a trademark of Novell, Inc.
exteNd Composer is a trademark of Novell, Inc.
exteNd Director is a trademark of Novell, Inc.
iChain is a registered trademark of Novell, Inc.
jBroker is a trademark of Novell, Inc.
NetWare is a registered trademark of Novell, Inc.
Novell is a registered trademark of Novell, Inc.
Novell eGuide is a trademark of Novell, Inc.

SilverStream Trademarks

SilverStream is a registered trademark of SilverStream Software, LLC.

Third-Party Trademarks

All third-party trademarks are the property of their respective owners.

Third-Party Software Legal Notices

The Apache Software License, Version 1.1

Copyright (c) 2000 The Apache Software Foundation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. 4. The names "Apache" and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org. 5. Products derived from this software may not be called "Apache", nor may "Apache" appear in their name, without prior written permission of the Apache Software Foundation.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

JDOM.JAR

Copyright (C) 2000-2002 Brett McLaughlin & Jason Hunter. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution. 3. The name "JDOM" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@jdom.org. 4. Products derived from this software may not be called "JDOM", nor may "JDOM" appear in their name, without prior written permission from the JDOM Project Management (pm@jdom.org).

In addition, we request (but do not require) that you include in the end-user documentation provided with the redistribution and/or in the software itself an acknowledgement equivalent to the following: "This product includes software developed by the JDOM Project (http://www.jdom.org/)." Alternatively, the acknowledgment may be graphical using the logos available at http://www.jdom.org/images/logos.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JDOM AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Sun

Sun Microsystems, Inc. Sun, Sun Microsystems, the Sun Logo Sun, the Sun logo, Sun Microsystems, JavaBeans, Enterprise JavaBeans, JavaServer

Pages, Java Naming and Directory Interface, JDK, JDBC, Java, HotJava, HotJava Views, Visual Java, Solaris, NEO, Joe, Netra, NFS, ONC, ONC+, OpenWindows, PC-NFS, SNM, SunNet Manager, Solaris sunburst design, Solstice, SunCore, SolarNet, SunWeb, Sun Workstation, The Network Is The Computer, ToolTalk, Ultra, Ultracomputing, Ultraserver, Where The Network Is Going, SunWorkShop, XView, Java WorkShop, the Java Coffee Cup logo, Visual Java, and NetBeans are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Indiana University Extreme! Lab Software License

Version 1.1.1

Copyright (c) 2002 Extreme! Lab, Indiana University. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Indiana University Extreme! Lab (http://www.extreme.indiana.edu/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. 4. The names "Indiana University" and "Indiana University Extreme! Lab" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact http://www.extreme.indiana.edu/. 5. Products derived from this software may not use "Indiana University" name nor may "Indiana University" appear in their name, without prior written permission of the Indiana University.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS, COPYRIGHT HOLDERS OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Phaos

This Software is derived in part from the SSLavaTM Toolkit, which is Copyright ©1996-1998 by Phaos Technology Corporation. All Rights Reserved. Customer is prohibited from accessing the functionality of the Phaos software.

W3C

W3C® SOFTWARE NOTICE AND LICENSE

This work (and included software, documentation such as READMEs, or other related items) is being provided by the copyright holders under the following license. By obtaining, using and/or copying this work, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions.

Permission to copy, modify, and distribute this software and its documentation, with or without modification, for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the software and documentation or portions thereof, including modifications: 1.The full text of this NOTICE in a location viewable to users of the redistributed or derivative work. 2.Any pre-existing intellectual property disclaimers, notices, or terms and conditions. If none exist, the W3C Software Short Notice should be included (hypertext is preferred, text is permitted) within the body of any redistributed or derivative code. 3. Notice of any changes or modifications to the files, including the date changes were made. (We recommend you provide URIs to the location from which the code is derived.)

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SOFTWARE OR DOCUMENTATION.

The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to the software without specific, written prior permission. Title to copyright in this software and any associated documentation will at all times remain with copyright holders.

Contents

About This Book

Purpose

This guide describes how to use the LDAP Connect.

Audience

This book is for developers and systems integrators who are planning to use exteNd Composer to develop directory-aware services (including Web Services) and components.

Prerequisites

This book assumes prior familiarity with exteNd Composer's work environment and deployment options. Some familiarity with the Lightweight Directory Access Protocol is also assumed.

Additional documentation

For the complete set of Novell exteNd documentation, see the Novell Documentation Web Site (**http://www.novell.com/documentation-index/index.jsp**).

1 Welcome to LDAP Connect for exteNd Composer

Welcome to the *Novell exteNd LDAP Connect User's Guide*. This Guide is a companion to the *exteNd Composer User's Guide*, which details how to use all the standard features of Composer *except* for those specific to the Connect Component Editors. So, if you haven't looked at the *Composer User's Guide* yet, you should familiarize yourself with it before using this Guide.

Novell exteNd Composer provides separate Component Editors for each Connect. The special features of each component editor are described in separate Guides like this one.

If you have been using exteNd Composer, and are familiar with the core editors (such as the XML Map Component Editor), then this Guide will help you become productive with the LDAP Component Editor in minimal time.

About exteNd Composer and the Connect Architecture

Novell exteNd Composer is a tool for building (and deploying) your own XML-enabled integration applications.

Composer comes in two parts. The design-side part (which is what we mean when we say "Composer" in this guide) is an integrated development environment (IDE) for creating your own custom applications. The runtime part (called Composer Enterprise Server, or "Composer Server" for short) is the server-resident execution engine for your apps. (You do not need access to Composer Server in order to design, test, and debug your services, however.)

The services you build and deploy using Composer can be triggered (invoked) in any number of ways. A common option is to have a service invoked by a servlet that "listens" on a URL. (For more information on invocation and deployment options, see the *exteNd Composer User's Guide* as well as the *Composer Server Guide* for your particular app server).

This guide will show you how to build LDAP Components that you can use in your Composer-built services. Typically, you will:

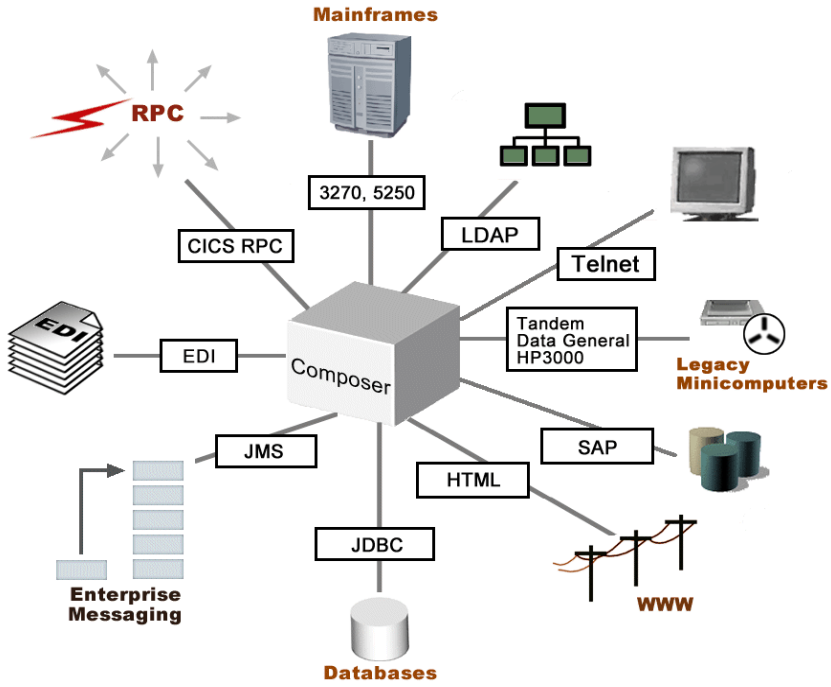
- Create a project (a Composer or Workbench**.spf** file).
- Create an LDAP Component and its associated resources (such as a Connection Resource).
- Optionally create other components (XML Map, JDBC, etc.) that carry out additional "business logic" operations.
- Create a *service* (or "Service Component") that calls your other components. The service, in other words, "wrappers" your lower-level components.
- Package and deploy your service in JAR, WAR, or EAR form as appropriate.

The *service* provides the invocation layer for a component (or group of components). It can be exposed as a WSDL-described Web Service, or it can run locally, with no public-facing interface, on the app server.

Hub and Spoke Architecture

Novell exteNd Composer is built on a simple hub-and-spoke architecture. The hub is a robust XML transformation engine that accepts XML documents, processes the documents, and returns an XML document created according to the particular requirements of the business process in question. The spokes, or *Connects*, are add-in modules that "XML-enable" sources of data that are not XML-aware. That is, Connects make it possible for non-XML data—whether from legacy COBOL-VSAM managed information systems, Telnet or other terminal streams, message queues, EDI, etc.—to be captured in XML form (or the reverse: repackage XML data so the endpoint system can understand it). Composer Connects are natively able to read, write, and transform XML using industry-standard parsers and transcoders.

All of this is done in a non-intrusive/non-invasive manner, such that there is no impact on existing systems or infrastructure. (For example, there is no need to load any new software on existing systems, aside from the app server itself.)



Composer comes with two core Connects, for JDBC and LDAP. (JDBC is the subject of a separate *User's Guide*.)

Other Connects are available for CICS RPC, Telnet, 3270, 5250, HP3000, Tandem, Data General, JMS, SAP, EDI, and HTML data sources. These additional Connects are not part of the core Composer installation but represent value-adds that can be purchased separately.

About exteNd's LDAP Connect

Just as the JDBC Connect lets you build and deploy XML integration applications that are *database*-aware, the LDAP Connect lets you build components and services that are *directory*-aware. Your component or service acquires the power to act as an LDAP client. It can make queries against (or even update the contents of) any directory—regardless of vendor—that supports the LDAP protocol.

The key to the power and flexibility of Composer's LDAP Connect is its ability to work with DSML (Directory Services Markup Language), which is an industry-standard XML grammar for encoding directory requests and responses. (See the more detailed discussion further below.) Since DSML is just a dialect of XML, it shares all of XML's advantages in terms of being human-readable, machine-parsable, transportable, firewall-friendly, etc. The data in a DSML document is easily accessed, transformed, and repurposed.

NOTE: You will not need to create, or keep on hand, actual DSML documents in order to work with the LDAP Connect. Composer will create the necessary DSML for you, on the fly.

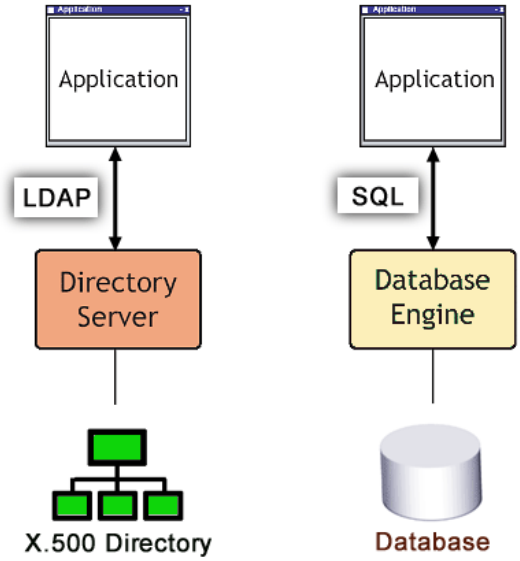
We will have more to say about DSML later on.

What Are Directories?

A directory is a structured data store. As such, it has many of the characteristics of a relational database (which is perhaps the best known type of structured data store). But a directory differs from a traditional database in a number of key ways:

- Whereas a database is designed to accommodate large chunks of potentially volatile data (data that may be subject to frequent updates), a directory is more suited to handling small, highly granularized bits of data that don't need frequent updating.
- Directories allow administrators to aggregate data in vertical object hierarchies built on flexible naming and containment rules. (Traditional databases follow a less vertical approach to organizing data.)
- Every data element (or entry) in a directory is *directly addressable* via a unique ID (known as its *distinguished name*). The address of a directory entry is constructed (and deconstructed) piecewise, much the same way a URL is, so that there can be relative versus fully qualified names. In a database, data elements are not directly addressable.

As with a database, data in a directory can be queried as well as written (updated) or removed. Databases can be queried via one flavor or another of SQL (Structured Query Language). With directories, queries follow a syntax described in RFCs 2251 and 2254. The query "language" for LDAP is well standardized.



The diagram above shows how applications communicate with the two basic types of structured data stores (directories and databases). Applications go through LDAP to query and/or update directories. They go through SQL and appropriate drivers to access data in a database.

How Is Information Stored in a Directory?

Directories store data as *entries*. Collections of entries are usually called *objects*. Objects can contain other objects as entries.

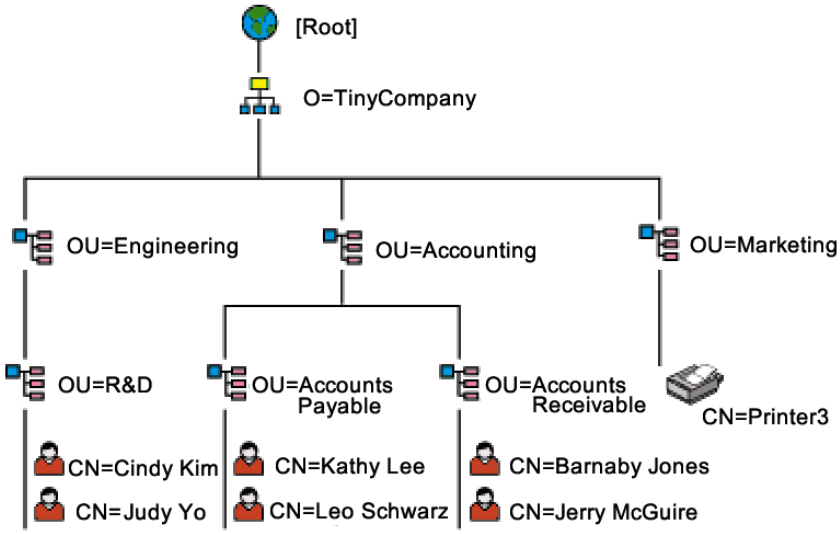
The data within directory objects consists of attribute-value pairs. Some attributes are said to be *single-valued*; others are *multi-valued*. For example, a person might have more than one phone number: In this case, the attribute under which phone numbers are stored for this person would probably be a multi-valued attribute. This would be specified in the directory's *schema*. (A *directory* schema is not to be confused with an XML schema. The two are completely distinct.)

When entries contain other entries, the containing entry is called a *container object*.

A contained entry is said to be the *subordinate entry* of the container. The container, on the other hand, is said to be the *superior* of the contained entry.

NOTE: An entry can be both a container (superior) as well as a subordinate to another entry, much the same way that a DOM node can be both a parent and a child.

Nesting of entries is a key characteristic of directories and gives them a treelike structure. (In fact, you will often hear the term Directory Information Tree, or DIT, in reference to a directory.) The tree structure lends itself, in turn, to XML representation.



The above graphic shows how the directory structure for a small company might look. In this example, the top-level node in `TinyCompany` has an Organization (O) attribute of `TinyCompany`. Under `TinyCompany` are containers for the Engineering, Accounting, and Marketing departments. (These are labelled as such in terms of OU or Organization Unit attributes.) Accounting has two organizational units under it (Accounts Payable and Accounts Receivable.) So far, all of these entries have been containers. At the leaf-node level, we reach the CN Common Name) entries for the various personnel under the containers. Note that Marketing has no employees at the moment. But they do have a printer.

Any item in the tree can be addresses by a concatenation (or federation) of namespaces: Judy Yo can be addressed unambiguously by means of:

```
cn=Judy Yo,ou=R&D,ou=Engineering,o=TinyCompany
```

This "fully qualified" name is called a *distinguished name* (usually abbreviated "DN" or "dn"). It is analogous to a fully qualified path or URI.

NOTE: Order is important in a DN. Unlike URL or file-system naming schemes, the ordering inside a DN (going left to right) flows from *lowest* level to *highest* level of organization (or from child to parent)—always upwards along the ancestor chain.

Because order is important, it would be wrong to rewrite the above DN as

```
cn=Judy Yo,ou=Engineering,ou=R&D,o=TinyCompany
```

This DN states the hierarchy of relationships incorrectly, since R&D belongs under Engineering. The DN shown here would not resolve correctly given the object hierarchy shown in the foregoing graphic.

We will talk more about naming and other conventions in later chapters.

What is LDAP?

LDAP (Lightweight Directory Access Protocol) is a messaging protocol for communicating with directories. It implements a compact subset of the functionality specified in the more elaborate *Directory Access Protocol*. Full DAP is a feature-rich, far-reaching, rigorous ("heavyweight") protocol for communicating with X.500 directory servers. By comparison, LDAP is a *lightweight* protocol with a reduced feature set and relatively modest learning curve for those wishing to use it. You can think of LDAP as a kind of complexity-reduced version of DAP based on ordinary TCP/IP for messaging as opposed to DAP's use of the less-common OSI network protocol stack.

In many circles, LDAP has come to mean more than just the network protocol described in RFC 3377. When people talk about LDAP, they may be talking about:

- The LDAP *functional model*: that is to say, the kinds of operations you can perform on a directory
- The LDAP *namespace model*: The ways in which data groupings can be distinguished or identified by name
- The LDAP *data model*, which defines the kinds of information that can be stored in a directory, and rules for organizing that data

Many of these concepts stem from International Telephone Union (ITU) Technical Recommendation X.500: "The Directory: Overview of Concepts, Models and Service"(1993). LDAP remains, at its core, a *network protocol* (like HTTP), but its semantics are tightly bound to the conceptual framework defined by ITU's X.500, X.501, and X.511 standards.

With LDAP, you can establish communications with (or "bind to") a directory, access (read) information in the directory, and/or update (write information to) the directory.

The LDAP v3 specification is at **http://www.ietf.org/rfc/rfc2251.txt**.

For additional information on LDAP, consult **http://www.openldap.org.**

What Does LDAP Do?

LDAP is a protocol for communicating with directories. Like any network protocol, LDAP has its own "handshake" conventions and a vocabulary of keywords with implied semantics. There are 20 verbs, total (see the section following this one), describing various types of operations:

- **Bind**—The *bind* operation establishes an LDAP session between a client application and a directory server. (This operation also allows a client application to pass authentication information to the server.)
- **Unbind**—The *unbind* operation terminates an LDAP session and signals to the server that the "connection" can be discarded.
- **Search**—The *search* operation allows an LDAP-enabled client application to exploit the lookup services offered by a directory server. Depending on the parameters in the client application's search request, the server will return information from a single entry, from all of the entries below a particular entry on the directory tree, or from an entire *branch* of the directory tree.

NOTE: In addition to allowing client applications to define the scope of a particular search, LDAP 3 parameters allow a client application to define search *filters*; a size limit for the number of entries returned; and a time limit within which the search should be performed.

- **Modify**—The *modify* operation allows a client to modify the value of an attribute for a particular entry. Parameters of the modify operation allow a client application to add values for attributes, to delete values for attributes, and to change values for attributes. For example, a client application might request an add, delete, or replace operation.
  - *Add* a telephone number to the `telephoneNumber` attribute of a particular user entry.
  - *Delete* a telephone number from the `telephoneNumber` attribute.
  - *Replace* the existing telephone number with a new telephone number.
  - *Modify Distinguished Name* (DN). The modify-DN operation allows a client application to change an entry's distinguished name by changing the leftmost element of that name. (Distinguished names are discussed further below.)
- **Compare**—The *compare* operation allows a client application to compare a stated attribute value with the value of an attribute in a particular entry. For example, a client application might verify a particular user's password using the compare operation.
- **Abandon**—The abandon operation allows a client application to abandon an operation that has not yet been completed.

LDAP 3 also defines an *indirection* capability—that is, a means by which a directory can *refer* a client application to another LDAP 3 directory. For example, suppose a client application were to request information about a particular entry from LDAP Directory A, but Directory A could not find the requested entry. If Directory A knew that Directory B that might contain the desired info, A could refer the client to B, and the search could continue (possibly along an extended series of referrals).

When the host silently implements its own referral-following scheme, it's called *chaining*. In LDAP, the client has no knowledge of nor (generally speaking) any control over chaining, since it's totally under the server's control.

Referral-following, on the other hand, is under the client's control: The client must decide whether to act on each referral as it is received, and what kind of security to use on each "hop."

NOTE: The current version of the LDAP Connect for exteNd Composer does not provide native support for referral-following. You can, however, build your own action-model logic to accomplish this, using ordinary looping constructs and dynamic connection-resource parameter values.

LDAP Verbs

The above discussion presented LDAP operations in "non-programmer" terms. At a lower level, LDAP operations are specified by LDAP *verbs.*

The current list of LDAP verbs (including those that are unique to Version 3 of the spec) looks like this:

- `BindRequest`
- `BindResponse`
- `UnbindRequest`
- `SearchRequest`
- SearchResponse (v2 only; not in v3)
- `SearchResultEntry (v3)`
- `SearchResultDone (v3)`
- `SearchResultReference (v3)`
- `ModifyRequest`
- `ModifyResponse`
- `AddRequest`
- `AddResponse`
- `DelRequest`
- `DelResponse`
- `ModifyDNRequest`

- ModifyDNResponse
- CompareRequest
- CompareResponse
- AbandonRequest
- ExtendedRequest (v3)
- ExtendedResponse (v3)

These verbs are presented here merely to give you an idea of the kinds of operations that are possible in LDAP. You do not need to understand how to work with these verbs directly in order to use the Composer LDAP Connect.

For more information on the uses and meanings of these commands, refer to the developer documentation at **http://developer.novell.com/ndk/**.

What is DSML?

Directory Services Markup Language, or DSML, allows directory information and/or directory queries to be represented as an XML document.

NOTE: If you are already experienced with LDAP, you can think of DSML as the XML analog of an LDIF file. (LDIF is the LDAP Data Interchange Format, a text format for specifying directory entries and LDAP queries. LDIF is described in RFC 2849)

The DSML specification (see **http://www.oasis-open.org**) was created in order to make it easy for XML-based enterprise applications to leverage resource information stored in directories, using firewall-friendly mechanisms like SOAP (Simple Object Access Protocol).

DSML allows XML and directories to work together. It provides a generalized mechanism by which XML-based applications can access directory-based information. The goal of DSML is simply to make it possible for an ever-growing number of XML-based enterprise applications to be directory-aware.

NOTE: You do *not* need to understand DSML in order to use the Composer LDAP Connect. Many of the LDAP Connect's features are wizard-driven. Based on your input to the wizard dialogs, Composer's LDAP Connect will build any necessary DSML documents or DOMs for you, dynamically.

What Kinds of Applications Can You Build Using the LDAP Connect?

With the aid of the LDAP Connect and Composer, you can build "directory awareness" into your XML integration applications (whether they're Web Services or private apps running in a local context). Your LDAP-aware app can push data into or pull data from any LDAP-accessible data store, using XML as the interchange format. (DSML is the XML dialect that is actually used.) And you can do this without having to know anything about DSML. For example, you can write a component (perhaps part of a larger web service) that retrieves the phone number, e-mail address, and title of a company employee from a company directory. If the information your app needs resides in two or more directories, you can merge the information from separate directories before displaying it to the user or passing it to another component in your application.

LDAP and Security

LDAP applications typically use SSL and TLS (Transport Layer Security) for authentication of LDAP endpoints and encryption of LDAP session data.

TLS is a generic wrapper for various kinds of transport-layer security. The participants in a TLS session agree, at the beginning of a session, to use one of several available security mechanisms (a typical choice being Secure Socket Layer technology); then the participants use that mechanism to conduct a "secure session."

When TLS is enabled, all communications are encrypted; no passwords (and no data) are ever sent in the clear.

Host authentication is also a component of TLS. By default, the host sends its X.509 certificate information to the client and the client verifies (authenticates) the host, confirming that the host in question is indeed the LDAP server that the client was expecting.

NOTE: There are other possible authentication handshakes, including authentication of the client by the host *without* authentication of the host by the client, as well as two-way mutual authentication; but the LDAP Connect for exteNd Composer currently supports only the most common handshake scenario, in which the server provides its certificate info to the client.

Additional discussion of LDAP Connect security will come later (in *"Getting Started with the LDAP Component Editor"*).

For more information on Transport Layer Security, see RFC 2246 at **http://www.faqs.org/rfcs/rfc2246.html**.

Access Control

Access control is different from "security" of the kind discussed above: Rather than addressing authentication and encryption, access control has to do with authorization, operational privilege levels (read-only vs. read-write), inheritance rules for privileges, object and attribute visibilities, etc., and assignment, management, and enforcement of these properties on a per-user and per-group basis.

ITU X.501 addresses (in a characteristically abstract way) directory access control concepts, but does not specify access control mechanisms in enough architectural detail to enable directory vendors to implement access control in a standard way. Therefore, access control features have necessarily tended to be implemented in vendor-specific ways. There is no one "standard" way to implement access control.

Novell's eDirectory offers a rich, flexible, robust access control architecture centered on an attribute called ACL (for "Access Control List"). The ACL attribute is an optional, multivalued attribute that (in eDirectory and NDS) is defined on the Top object class. Since all objects derive from Top, all object classes can avail themselves of the ACL attribute.

The ACL attribute is an attribute on *the object that is being accessed*. Each ACL value specifies who can access the object; what type of rights the accessor has been granted; and whether children of the accessor inherit those rights. When an object has been granted rights to another object, the accessor is called a *trustee* of the target object.

Since the ACL attribute is multivalued, any object that uses an ACL attribute can store any number of values in it. Typically, there is one value per trustee. Therefore, an ACL might contain a great many values. But some trustees might actually be container objects representing large groups. The rights of an entire group of directory objects can be controlled through a single ACL entry.

ACL-based access control is based on very simple principles, but the ramifications of those principles are far-reaching. For a technical overview of ACL concepts, you should consult the **NDS Technical Overview** documentation (in particular, the chapter on eDirectory Security) available online at **http://developer.novell.com/ndk.** ACL is also discussed later in this guide, in the discussion of "Access Control List (ACL) Methods."

For More Information

The web has many good LDAP and directory resources. A good place to start is **http://developer.novell.com/edirectory/ndsldap.htm**, which has links to many LDAP-related articles, specifications, and developer resources.

Links to the RFCs covering LDAP, along with other resources, can be found at **http://nldap.com/nldap/**. Some of the RFCs include the following:

- RFC 2251 - LDAP (v3)
- RFC 2252 - LDAP (v3): Attribute Syntax Definitions
- RFC 2253 - LDAP (v3): UTF-8 String Representation of Distinguished Names
- RFC 2254 - The String Representation of LDAP Search Filters
- RFC 2255 - The LDAP URL Format
- RFC 2256 - A Summary of the X.500(96) User Schema for use with LDAPv3

A public LDAP test directory is maintained by Novell at **http://nldap.com**. You can use it to set up a private container for test purposes, if you don't have a local LDAP server against which to test your LDAP Connect components and services.

Information on the X.500 directory standard (and DAP) can be found at the web site of the International Telecommunication Union, **http://www.itu.int**. (Note: ITU's standards must be purchased before they can be downloaded.)

2 Getting Started with the LDAP Component Editor

The steps involved in creating a Composer component using the LDAP Connect are fundamentally no different from those involved in creating any other kind of component. The steps are:

1. Decide on any XML Templates you may need for your component.
2. Create a Connection Resource to allow your component to bind to an LDAP host. (The procedure for this is explained in detail below.)
3. Create a new Component.
4. Create Actions specific to the Component.
5. Test the action model by running the Component in animation mode.
6. Fix any problem discovered during animation.
7. Save your work.
8. Create a deployable Service that calls the Component. (Components cannot be deployed directly; the unit of deployment in a Composer project is the *Service*.)
9. Deploy your service to a staging area or to an app server environment (such as Novell exteNd App server, IBM WebSphere, or BEA Weblogic).

In the sections that follow, we'll look closely at Step 2: creating a Connection Resource. See the later discussion under "Creating an LDAP Component" for a detailed explanation of how to use LDAP-specific Actions.

About Connection Resources

Before you can create a working LDAP Component, you need to create a Connection Resource for it. (If you try to create an LDAP Component and there are no LDAP Connection Resources in your resource list, you will be prompted with a dialog that offers to take you to the Connection Resource wizard.)

The LDAP Connection Resource contains the information needed to allow a component to connect to an LDAP host (or directory server). You will generally create a separate resource for every different host you want to establish a connection with, *or* for every unique set of credentials (or timeout settings) that you want to use with a given host.

NOTE: The various parameters associated with a given connection resource can be late-binding if you specify *expression-driven connection parameters* in your resource, as described below.

About Expression-Driven Connection Parameters

You can specify Connection parameter values in one of two ways: as *constants* or as *expressions*.

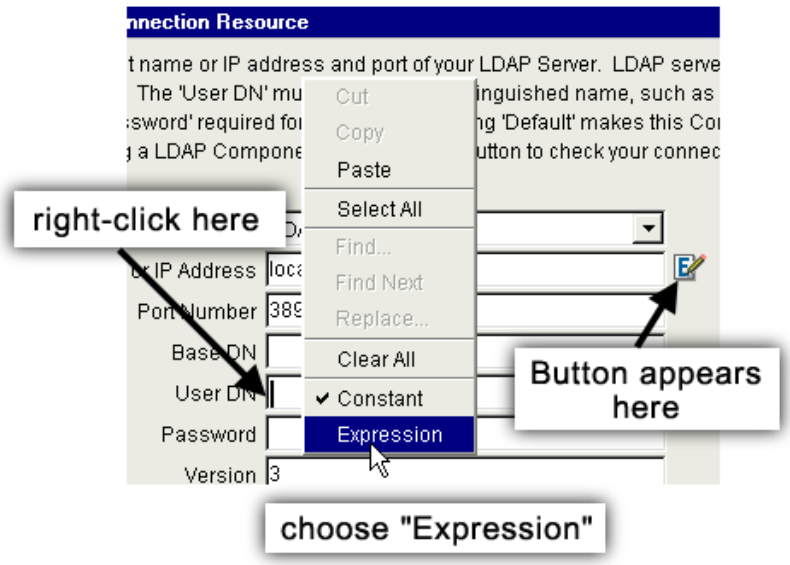
A constant-based parameter uses the literal value you supplied in the Connection wizard every time the Connection is used. An *expression*-based parameter means that you set the value using a programmatic (ECMAScript) expression, which is evaluated at runtime. The “late binding” of parameters afforded by the latter method makes it possible for you to supply a different User DN or password each time the connection resource is used. That is, the connection params can be chosen programmatically, at runtime.

Suppose you want to pull login credentials from a file or database at runtime. You can set up your connection resource to use expressions that look up the necessary info from files or databases via ECMAScript File I/O extensions or via Java directly. The use of expressions allows a Connection’s behavior to be flexible and vary dynamically in accordance with runtime conditions.

➢ **To switch a parameter from Constant to Expression driven:**

1 Click the right mouse button inside the parameter field you are interested in changing.

2 Select **Expression** from the context menu that appears. A small editor button (icon) will become visible to the right of the parameter field.



3 Click on the button and use the Expression Builder dialog to create an ECMAScript expression that evaluates to a valid parameter value at runtime.

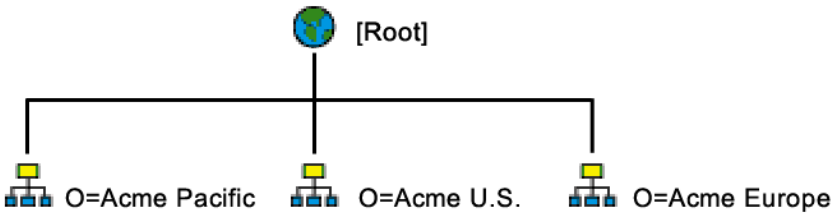
LDAP Connection Parameters

To create an LDAP connection, you will need to supply, at a minimum, the *server IP address* and the *port number* on which the session will occur. These two items are all that is needed to bind anonymously to a host that permits anonymous binds. (An anonymous bind in LDAP is analogous to an anonymous FTP session.) The IP address might be something like 127.0.0.1 or “localhost” at design-time (but presumably something else at runtime; perhaps switched by an expression-driven IP address parameter). The Port Number is typically 389 for a non-secure session and 636 for an SSL session, because this is how most directory servers are configured, but in fact you can supply any value here that might apply.

For non-anonymous binds, you will need to supply a User DN (user distinguished name), and a password. The User DN represents (in the words of RFC 2251) “the name of the directory object that the client wishes to bind as.” It is typically the distinguished name of a person or entity in the directory, such as “cn=John Doe, ou=Mailroom,o=Rising Star Industries.”

When a given server exposes more than one tree, the server needs to know which tree (more accurately, which *subtree*) the client wants to bind to. This information is given in the form of a *Base DN*.

For example, consider a directory server operated by the fictitious international conglomerate Acme, Inc. The Acme directory server might have three top-level container objects—three "trees"—representing complete directory structures for Acme Pacific, Acme U.S., and Acme Europe, as shown below.



If an LDAP client wants to bind to the Acme directory, the server will need to know which tree (Pacific., U.S., or Europe) the client wishes to bind to. In this example, if an Acme human resources manager in Spain wants to bind to the tree for Acme Europe, he would specify "o=Acme Europe" as the Base DN.

NOTE: In LDAP, the Root object is not really an entry in the tree and is not addressable directly. The Root is more formally known as the root DSE, or DSA Specific Entry. (A DSA is a *directory system agent*, which is an X.500 term for a directory server.) The DSE can be queried for "meta" information about the directory server (in LDAP version 3).

Security Settings

The LDAP Connect for Composer supports encrypted sessions (and server authentication using X.509 digital certificate technology) through the use of SSL.

To enable encryption and authentication, you must check the TLS (Transport Layer Security) checkbox on the connection setup dialog. In most cases, you will also want to set the Port Number (in the same dialog) to 636, because LDAP servers generally expect to carry out encrypted sessions on that port.

NOTE: The Port Number shown in the dialog does *not* automatically change to 636 when you set the TLS checkbox. You must enter the Port Number manually. (Likewise, merely entering a port value of 636 does not cause the TLS checkbox to come on.)

When the TLS box is checked, all information moving across the connection will be encrypted; no information will be sent in the clear. Also, the host (or directory server) will be presented a certificate challenge. The host will respond with its X.509 certificate info. Composer will check that certificate info against the Certificate Authority data stored in the `agrootca.jar` file (which is a Java archive that ships with Composer, containing Certificate Authority info for numerous industry standard certificate issuers; see the *Composer User's Guide* for details).

NOTE: You can find **agrootca.jar** in your Composer-installation **\lib** folder. Use WinZip to open, inspect, and/or add new certificates to this file. *Be sure to consult your app server's documentation to learn how to add X.509 certificates, and any other security resources that your applications might need, to your app server's runtime environment.*

The SSL3 security mechanisms supported by Composer's LDAP Connect are "all or nothing" (not dynamically switchable). In other words: the ability to begin an LDAP session in clear-text mode, then drop into "secure" mode on the fly (as part of the same session, using the same Connection Resource), is not supported in this version of the LDAP Connect.

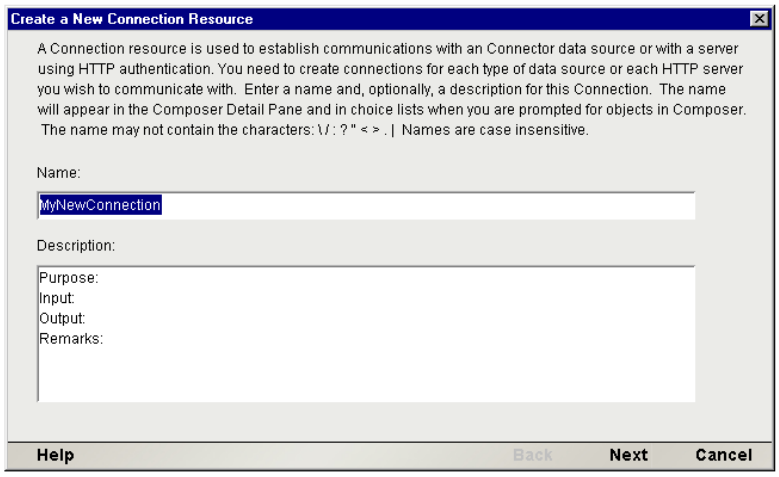
Creating an LDAP Connection Resource

The process of creating an LDAP Connection Resource is straightforward.

➢ **To create a LDAP connection resource:**

1 From Composer's main menubar, select **File > New > xObject**, then open the **Resource** tab and select **Connection**.

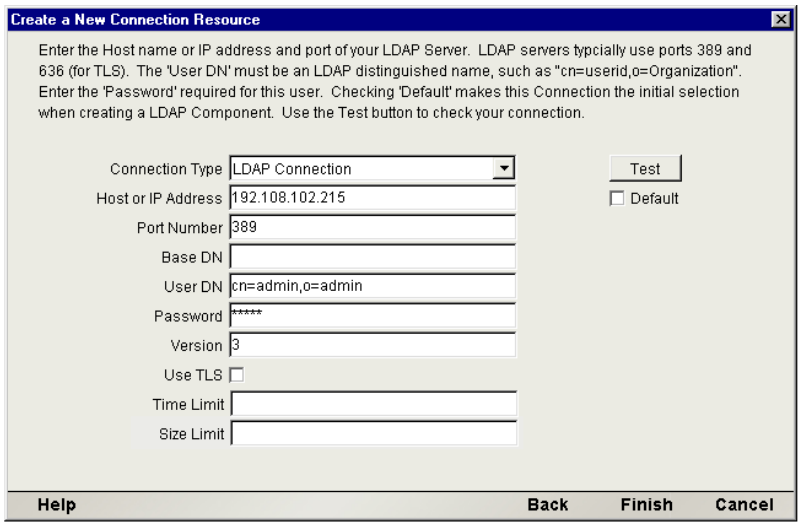
The "Create a New Connection Resource" Wizard appears.



2 Type a **Name** for the connection object. (This is the name that will show up later in the Navigator pane of Composer's main window when you are browsing resources in the Connection category.)

3 (Optional) Enter text in the **Description** area of the dialog.

4 Click **Next**. A new dialog appears.



5 Select **LDAP Connection** from the **Connection Type** pulldown menu.

6 Next to **Host or IP Address**, enter the name or IP address of the directory server with which you intend to connect. (For testing, this might be `localhost`.)

NOTE: This parameter, and all subsequent text fields in this dialog, can be set dynamically using ECMAScript expressions. See "About Expression-Driven Connection Parameters" earlier in this chapter.

7 If your connection will be using a port other than the default of **389**, enter the appropriate value next to **Port Number.** (If the connection will be a TLS connection using SSL, and you are not sure what value to enter here, enter **636**, which is the standard port for LDAP-over-SSL.)

TIP: You can specify a port as part of the IP address (previous step) using colon notation, such as `localhost:389`. If you do decide to use the all-in-one *IP-address:port* notation, enter 0 (zero) in the Port Number field.

8 Enter a valid **Base DN** (tree name), if applicable. This value might look something like `t=DEVNET-TREE`.

9 Enter a valid **User DN** (user distinguished name). See earlier discussion for information about distinguished names.

NOTE: If you are attempting an anonymous (no-password) bind, you can enter any value here.

10 No password is required for an anonymous bind. In all other cases, enter a valid **Password** for the given User DN.

CAUTION: *This password will be sent in the clear if the connection is not TLS-enabled.*

11 Unless your LDAP host requires a different value, accept the default **Version** value (LDAP version) of **3**.

12 Check the **TLS** checkbox if this is to be a secure connection (SSL3). Note that for X.509 certificate-based authentication to work, you must have the appropriate Certificate Authority entries in your **agrootca.jar** file (in Composer's **\lib** directory). Also, the appropriate certificate setups must exist on the server. Consult your app server documentation for details.

13 Optionally enter a **Time Limit** value representing the maximum number of milliseconds your component is willing to wait for establishment of a connection with the server. The default is no limit.

14 Optionally enter an integer value in **Size Limit**, representing the maximum number of tree entries (nodes) you are willing to accept for purposes of rendering tree-views of the directory at design time. The default value is 1000, which means that Composer can display a maximum of 1000 child nodes under any given parent node (entry) in the tree browser.

Two things could happen if you enter a very large number here:

- Performance could suffer as Composer tries to pull down and render all the requested entries
- The remote host might reach a preset limit on how many entries it can serve, and return an error (in which case Composer will not be able to display entries under the node in question)

It is recommended that you accept the default value of 1000, unless you know for sure that you will need to browse a tree node that contains a particularly large number of child nodes.

NOTE: This is a design-time consideration *only*. At runtime, on the server, this setting does nothing.

15 Click **Test** to see if the connection defined in your resource really works. A success or failure alert dialog will appear after a few seconds. (The test can take 30 seconds or more to "time out" if no connection can be made. To lessen the wait, enter a Time Limit value as described in the previous step.)

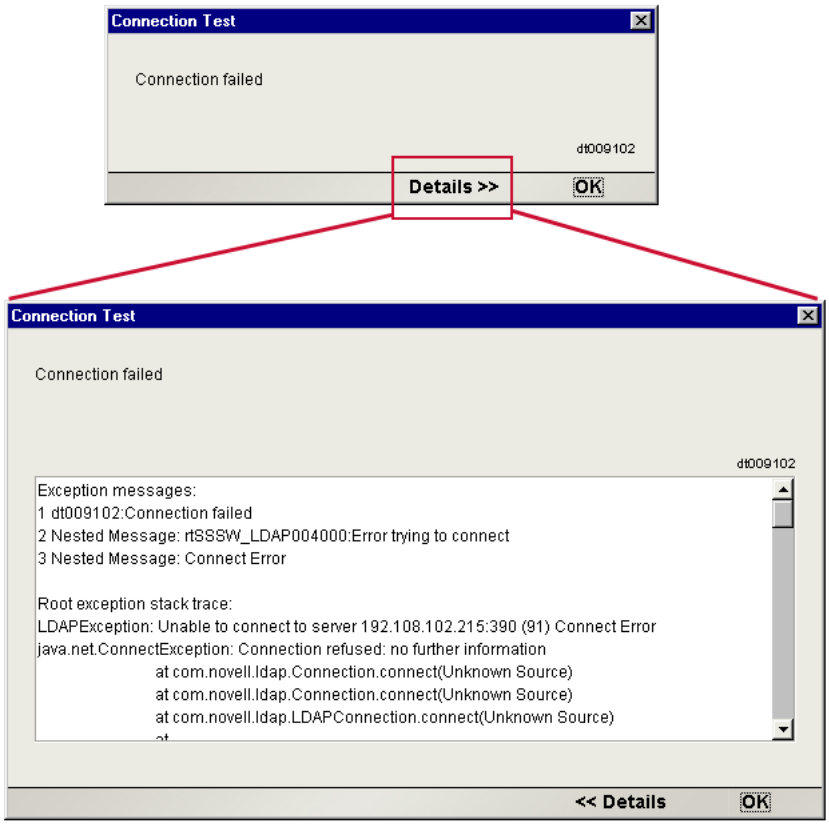
NOTE: You can continue editing the resource if your connection fails. You can also Save the resource and edit it later.

16 Optionally check the **Default** checkbox (underneath the Test button) if you want this connection to be used as the default connection each time you create a new LDAP Component. (You can override this behavior at any time.)

17 Click **Finish**. The newly-created resource connection object appears in the Composer Connection Resource detail pane.

Connection Troubleshooting

If you see an error dialog (instead of an alert with the message “Connected Successfully”) when you press the Test button, click the Details button of the dialog to see a full stack trace for the exception. See illustration below.



The stack trace message often provides useful clues for isolating the cause of the problem. In the example shown above, the connection was refused. On closer examination, it can be seen that the IP address specifies (at the end, after a colon) a port number of 390. The correct port number for the server was 389. Hence, no connection could be obtained and an `LDAPException` was thrown.

Here are some issues to bear in mind if you encounter trouble:

- Anonymous binds: If you can’t establish an anonymous bind, remember that the host server *might not be set up to allow anonymous binds*—the same way that an FTP server may not be set up to allow “anonymous FTP” sessions. If the server doesn’t allow anonymous binding, you *must* authenticate to the server in some fashion (whether by simple password and User DN over port 389, or full SSL on port 636, or some other way).
- **ECMAScript expressions:** When expression-driven parameters are used (see “About Expression-Driven Connection Parameters” further above), literal string values must be wrapped in quotation marks. Conversely, *if you are not using expression-driven parameters, do not wrap param values in quotation marks.*
- **TLS:** Be sure to uncheck this checkbox when a secure connection is not needed; and when it is checked, verify that you’ve specified a “secure port” (such as 636). Also verify, when using a secure connection, that your Certificate Authority **.jar** file contains the correct entries and is installed properly. (Look for a file called **agrootca.jar** in your Composer installation. The fully qualified path to this file must be specified in the `<XCCERTFILE>` element of your **xconfig.xml** file, not only in the design-time installation but also the runtime, or server, installation. Also, you may have to take additional steps to be sure this file can be found by your app server at runtime. See your app server documentation for details on how to set up X.509 certificates and certificate authority files.)

Exception After Successful Connection

It is possible to see a “Connected Successfully” alert, followed by an error dialog, when testing a connection. The success message means the host IP address and port number that you supplied were valid and a connection was established with the host. The subsequent exception message means a problem was encountered after connecting (such as a timeout, or an unknown Base DN).

NOTE: LDAP makes a subtle distinction between *connecting* and *binding*. Establishing a connection merely means that a the host and client were able to begin an LDAP session over TCP/IP, without reference to any particular target objects in a directory. A *bind* occurs when the host sees fit to associate a given combination of User DN and credentials with a given object in the server, in the context of session access.

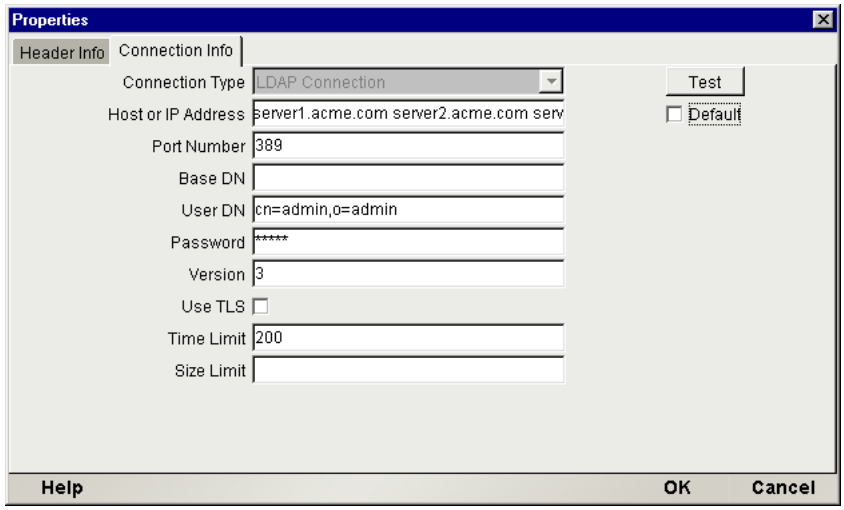
By way of analogy: Suppose 10-year-old Danny knocks on the door of the next-door-neighbor’s house. An adult opens the door. The child asks the adult: “Can Tommy come out and play?” In this example, Danny is analogous to an LDAP client and the adult is the LDAP host. When the door opens, a connection is established. The adult (server) then has to decide whether the child (client) should have access to Tommy (the target object). If the adult knows Danny (i.e., Danny can be authenticated in some way), and Danny has access rights to Tommy (i.e., it’s the right time of day, Tommy is done with his homework, etc.), then the two children might be allowed to play (to “bind”). But if the adult has never seen Danny before or doesn’t trust him, or if access rules preclude Tommy’s availability, then Danny will be turned away, even though the front door did open and a conversation took place.

Silent Failover

Directories quite often form the heart of mission-critical systems, and as a result, things like system outages (no matter how brief) and normal maintenance-related takedowns tend to be extremely disruptive. For that reason, directory trees are often *replicated* across servers, so that if a client tries to connect with Server A, but Server A is down, it can try Server B.

If you know the IP address of one or more backup servers that can be used if the primary server you’re interested in is either physically down or simply bogged down due to high demand, you can construct your LDAP Connection Resource in such a way as to provide silent failover capability.

Suppose your primary server is at **server1.acme.com**, but there are replicas of the directory also on **server2.acme.com** and **server3.acme.com**. When you create your connection resource, enter all three addresses (separating each with a space character) in the “Host or IP Address” field:



In the example shown, you'll see that not only have three addresses been entered in "Host or IP Address," but a value of 200 milliseconds has been specified under Time Limit (at the bottom of the dialog). When a component uses this resource, it will first try to connect to **server1.acme.com** (on the specified port of 389). If that server is down, or takes longer than 200 milliseconds to respond, a new connection attempt will be tried on the next address: **server2.acme.com**. If *that* server doesn't respond within 200 milliseconds, **server3.acme.com** will be tried. Finally, if *none* of the servers can be connected to, an exception is thrown.

NOTE: All three servers, in this example, will be tried on port 389, because that's the value that was entered in the Port Number field of the dialog. If the servers are *not* all using the same port, it is still possible to build a failover connection by using colon notation in the IP addresses: **server1.acme.com:389, server2.acme.com:636**, etc. If you do this, however, you should enter 0 (zero) in the Port Number field of the dialog.

Editing Connection Resources After They're Created

You can go back and change a connection resource at any time. Simply locate the resource by name in the Navigator pane (at the Composer window's left edge) and doubleclick it to bring up a tabbed dialog containing the information you typed into the wizard when you created the resource originally. (See previous illustration.) Choose the tab of interest and type new information into the appropriate fields. Click OK to keep your changes, or Cancel to revert back to the resource's original settings.

3 Creating an LDAP Component

The LDAP Connect for Novell exteNd Composer allows you to build XML integration applications that are directory-aware. In practical terms, this means you can leverage well-established LDAP APIs for:

- Searching and retrieving entries from a directory
- Adding new entries to the directory
- Updating entries in the directory
- Deleting entries from the directory
- Renaming entries in the directory
- Binding operations
- Abandoning operations

LDAP's search function emulates the more complex X.500 operations such as *list* and *read*. The metaphor is the same: You specify a base object to be searched and the portion (or scope) of the tree to search. A *filter* specifies the conditions that must be met in order for the search to capture a particular entry. (The LDAP search operation offers identical functionality to DAP's, but is encoded in a simpler form.)

NOTE: While in theory the LDAP API allows applications to perform operations either *synchronously* or *asynchronously* depending on whether the client wants to wait for an operation to complete before receiving the results of a previous operation, in actuality all operations in the LDAP Connect for exteNd Composer are synchronous.

In the LDAP Connect, you will issue LDAP queries in DSML (Directory Services Markup Language) form, using an industry-standard query-response syntax, rather than writing your own custom Java code to package and unpackage queries using an LDAP SDK. What makes DSML particularly attractive is that it is XML: You can map data into or out of it easily using ordinary Composer actions. What makes the LDAP Connect particularly powerful is its ability to autogenerate DSML for you, on the fly, so that you don't have to know low-level DSML internals.

The LDAP Application Model

LDAP applications typically perform five steps:

1. **Open a connection to the LDAP server.** This step involves initializing the session, setting session preferences and binding to the server. Usually the session preferences involve defining things like:
   - maximum number of entries returned in a search
   - maximum number of seconds spent on a search
   - how referrals are handled
   - security preferences
2. **Authenticate to the server.** The client can authenticate either anonymously with public rights, or through simple (clear-text) password authentication, or with full encryption. Novell's LDAP Services v3 uses SSL3 for full encryption and authentication.
3. **Perform the operations and obtain the results.** This step usually involves searching the directory, but may also involve modifying the directory data as well.

4 **Process the results.** This step involves making use of the returned information in some fashion, via custom business logic.

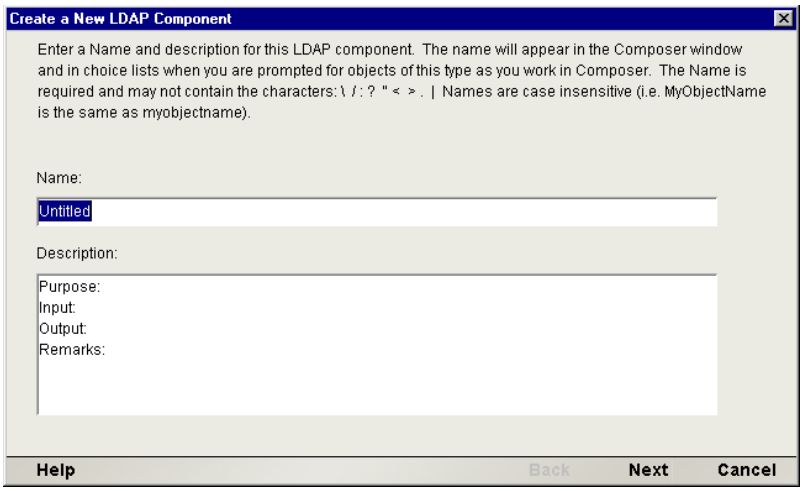
5 **Close the session.** This means unbinding from the server and disposing of the session handle. (In Composer, this step occurs automatically when your component goes out of scope.)

Before Creating an LDAP Component

As with all exteNd components, the first step in creating a new LDAP component is to determine whether you need any XML templates. (For more information, see *Creating a New XML Template* in the separate *Composer User's Guide*.) Providing you've created a Connection Resource in advance (discussed earlier), you can then create an LDAP Component, using your templates' sample documents to represent the inputs and outputs processed by your component.

➢ **To create a new LDAP component:**

1 In Composer's main menubar, go to the **File** menu and select **New > xObject**, then open the **Component** tab and select **LDAP**. The "Create a New LDAP Component" Wizard appears.

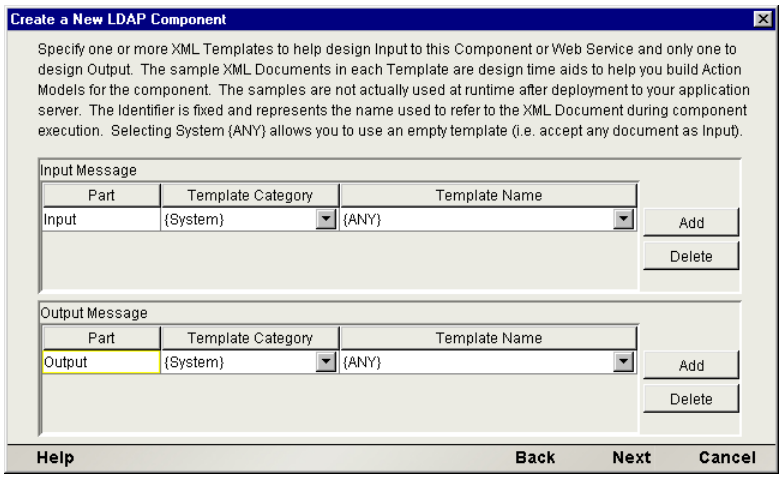


NOTE: If your project currently contains no LDAP Connection Resources, you will be prompted to create one at this point, and you will go through the Connection Resource creation process before you reach the above dialog.

2 Enter a **Name** for the new LDAP Component.

3 Optionally, add your own **Description** text.

4 Click **Next**. A new dialog appears.



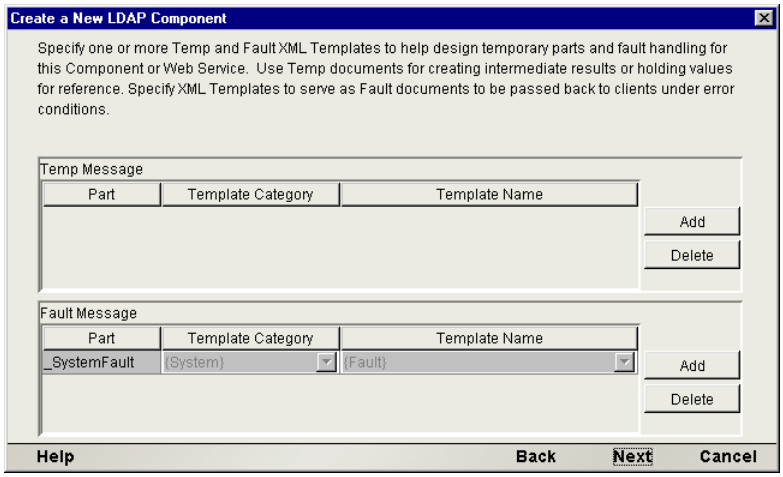
5 Specify one or more Input templates as follows:

- Select a **Template Category** if yours will be different than the default category.
- Using the dropdown menu immediately to the right., select a **Template Name** from the list of XML templates in the selected **Template Category**.
- To add additional input XML templates, click **Add** and repeat these steps.
- To *remove* an input XML template, click inside an entry and click **Delete**.

6 Select an XML template for Output.

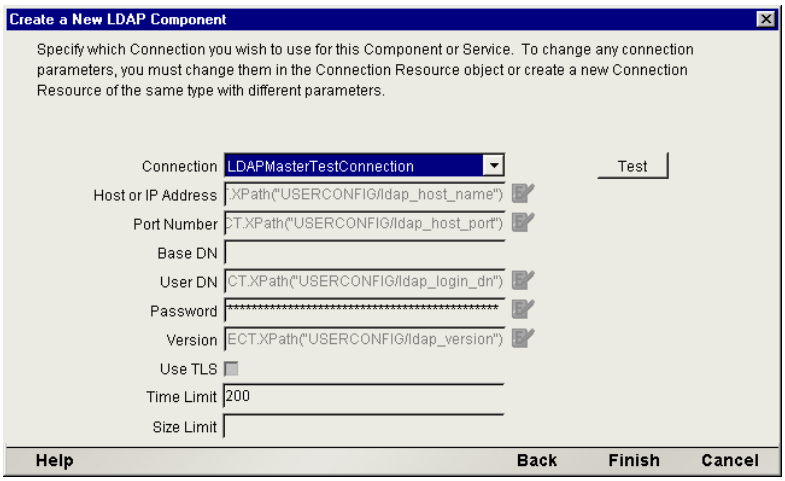
NOTE: You can specify an empty XML template by selecting **{System}{ANY}** as the Output template. You might do this if you wanted to generate a custom Output DOM dynamically using ECMAScript or XPath inside a Map Action. (For more information, see the *Composer User's Guide*, particularly the chapter on creating an XML Map Component.)

7 Click **Next.** A new dialog appears.



8 In this dialog, you may (optionally) add Temp documents for use as "scratchpad DOMs" in your component. You may also add Fault documents as needed. Use the **Add** and **Delete** buttons to add or remove documents as desired.

9 Click **Next**. The connection panel of the wizard appears.

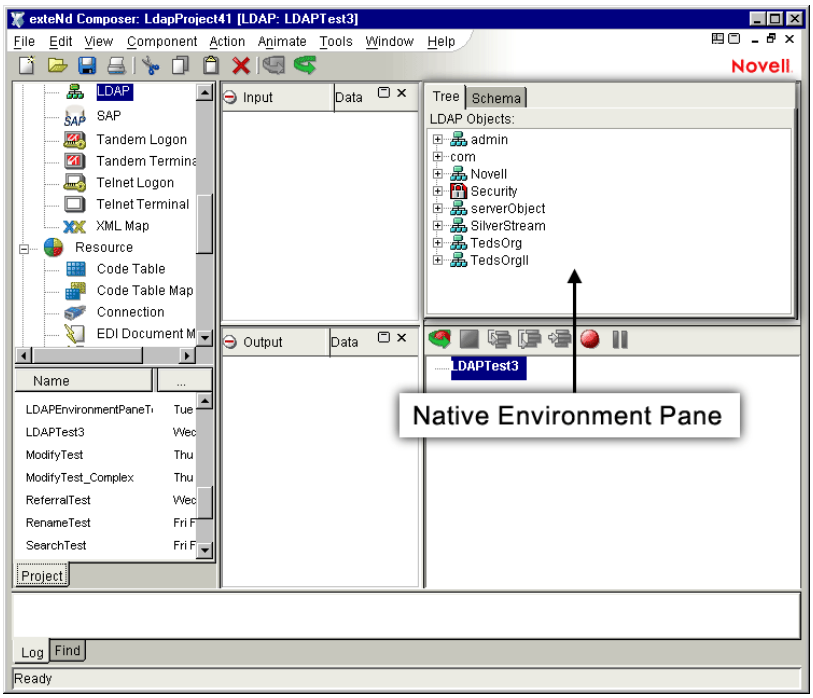


10 Select a **Connection** from the pull-down list. (The list will be prepopulated with the names of existing LDAP Connection Resources.)

NOTE: You can later change this selection and use a different connection resource, if you want. You are not “locked into” using this connection.

TIP: The fields of the connection resource will be greyed out in this dialog. If you need to edit the connection resource, open it separately after you are done creating the component.

11 Click **Finish**. The Composer main window appears, with a blank action model pane. The Native Environment Pane in the upper right should have a Tree tab as well as a Schema tab. (*Schema* here refers to an LDAP directory schema, not an XML schema.) When the Tree tab is selected, you should see a tree view of the target directory. See below.



At this point, you can begin creating actions in your action model, or you can simply **Save** your work and come back to the component later.

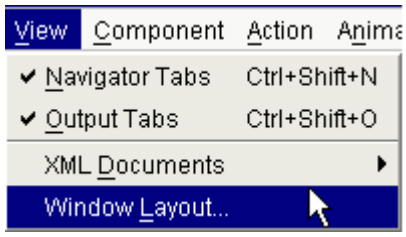
Special Features of the LDAP Component Editor

The LDAP Component Editor includes all the functionality of the XML Map Component Editor. It contains mapping panes for Input and Output XML documents as well as an Action pane. All of the regular Composer actions (such as XML Map, Function, Log, Decision, Send Mail, etc.) are available to you along with the two LDAP-specific actions, Create DSML and Execute DSML. (These actions will be discussed in detail in the next chapter.)

LDAP Native Environment Pane

The LDAP Component Editor includes a Native Environment Pane (see graphic above) with a Tree tab offering a browseable tree view of the directory to which your component's Connection Resource points, as well as a Schema tab (discussed in more detail below). If you are familiar with Novell's ConsoleOne tree browser, the Tree tab's browser operates in much the same way.

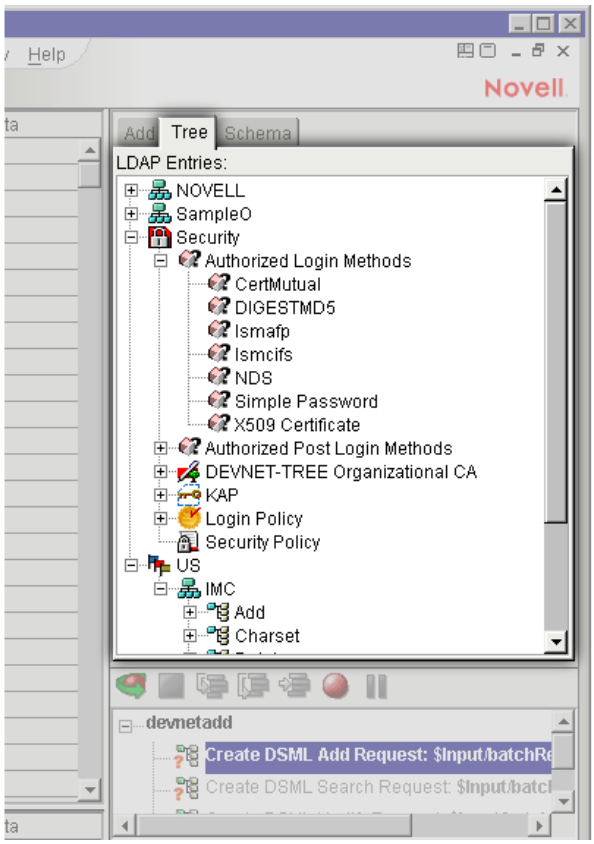
The Native Environment Pane's *default* location is in the upper right corner of the main Composer window. You can change its location, however, by using the **View > Window Layout** menu command:



This command will bring up a dialog in which you can specify North, West, East, and South positionings of the Native Environment Pane, XML document windows, and Action Model pane.

Tree Tab

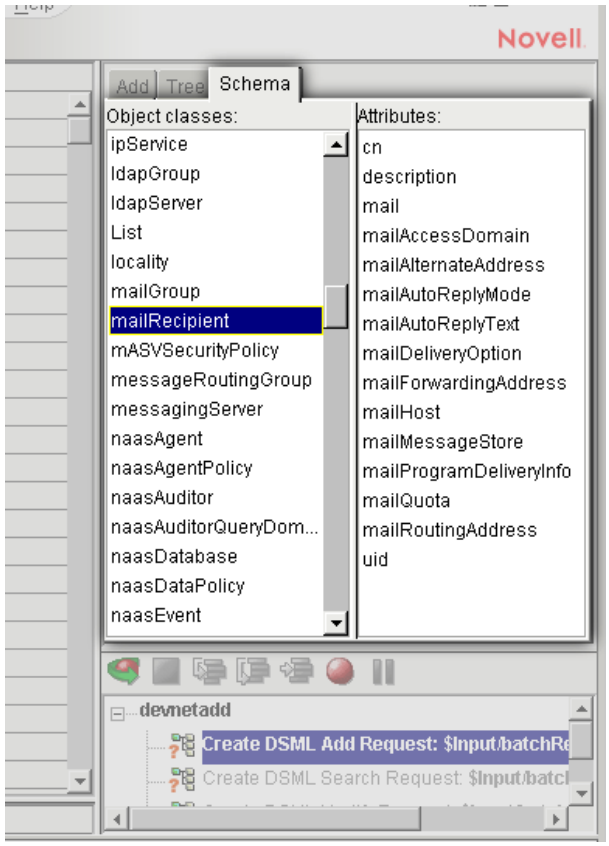
The Tree tab of the Native Environment Pane (see illustration below) offers an interactive design-time visual aid for finding and verifying the location of objects in the target directory. This view will populate automatically when you open a component. Its contents are based on the host specified in the component's Connection Resource.



NOTE: You cannot drag or drop items into nor out of the directory tree view.

Schema Tab

The Schema tab (see illustration below) is a read-only design-time aid for inspecting object/attribute relationships as defined by the schema for this particular tree. If your component will contain actions that modify the schema, you can verify the changes visually simply by inspecting this tab. (“Schema” here refers to the directory schema, not an XML schema.)

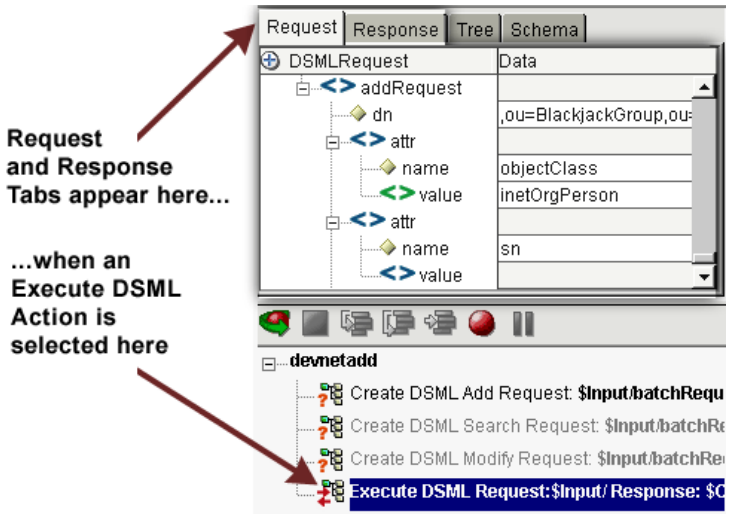


To see all of the attributes defined on a given object, simply click any object name in the top scrollpane of the tab; the list of attribute names in the bottom scrollpane updates accordingly in real time.

For additional information on the Composer work environment, consult the separate *Composer User’s Guide*.

Request and Response Tabs

Two special tabs—the Request and Response tabs—appear in the Native Environment Pane when an Execute DSML Action is selected (highlighted) in the action-model pane. These tabs (see below) present a tree view of the DSML request/response DOMs that were used by the Execute DSML action during execution.



NOTE: The tabs are initially empty. To populate the tab with the appropriate DOMs, run your action model in animation mode (or by using the Execute All button in Composer's main toolbar), then select/highlight the Execute DSML action.

The request and response DSML DOMs are volatile (transitory) and cannot serve as drag sources nor drop targets. To map to or from these DOM elements, use a Create DSML action to map a `batchRequest` element to *Input* (as described in the next chapter), or use an Execute DSML action to map a `batchResponse` element to *Temp*, *Output*, etc.—then drag-and-drop between DOMs (or map DOM contents in whatever way you choose).

Working with DSML DOMs will be described in more detail in the next chapter.

Contextual Tabs

Additional tabs (with names Add, Attributes, Compare, Filter, Modify, Rename, and Search) will appear dynamically when you add Create DSML actions of various "flavors" to your action model. The appearance and usage of each of these will be discussed later.

Drag-and-Drop Operations

Composer's UI allows you to drag and drop DOM nodes from one DOM tree to another, or in some cases, from a DOM tree to a field in the Native Environment Pane. This is a convenient, quick way to specify data-mapping rules that would otherwise require hand-coded XPath or ECMAScript expressions.

DOM-to-DOM

DOM-to-DOM drag-and-drop allows you to specify data mappings across documents. (For example, you can specify the transfer of data from a particular spot in a DSML response document to a special location in a custom DOM.)

When you click on a DOM element in (for instance) the Input DOM, in tree view, and drag it into another DOM window, then release the mouse when it is over a particular element, Composer maps a copy of the source's data, attributes, and children (including their attributes, etc.) into the target element (the "drop target"). At the same time, Composer automatically adds a Map Action, corresponding to the mapping that just occurred, to your component's action model.

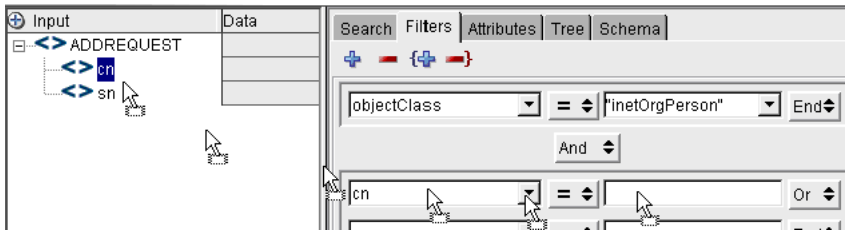
In many cases, the map-all-descendants behavior just described is what you will want. But in some cases, you may need different mapping behavior. For more precise control over data mappings, you will want to create Map Actions manually and use the Map Action dialog (and its Advanced button) to achieve the desired mapping(s).

NOTE: These techniques are described in detail in the *Composer User's Guide*. Consult that guide for more information.

DOM-to-NEP (Native Environment Pane)

In many instances, you can click on a DOM element in (for instance) the Input DOM, then drag that element to a text field in the Native Environment Pane and let go, thereby populating the target field with a reference to a particular XML element.

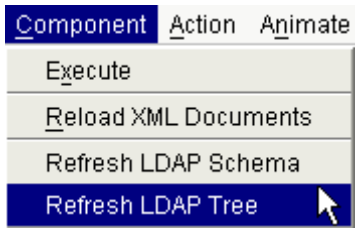
Consider the following example:



In this example, the user is constructing a search filter and is taking advantage of drag-and-drop to bind a node in the Input document to a parameter in the filter expression. The user has clicked the `cn` node under `ADDREQUEST` in Input, and dragged it to the empty field in the Native Environment Pane. When the mouse is released, the appropriate ECMAScript expression, namely `Input.XPath("ADDREQUEST/sn")`, will appear in the target field of the search expression. Thus, the filter (as constructed) will match entries where `objectClass` equals `inetOrgPerson` and `sn` equals the value in `Input/ADDREQUEST/sn`.

Special Menu Commands

Menu commands specific to the LDAP Connect include two new actions (Create DSML and Execute DSML, both under **Action > New Action**) as well as two commands under the Component menu:



The **Refresh LDAP Schema** command updates the contents of the Schema tab (discussed above). This command has no runtime significance. It is potentially useful at design time in cases where you've modified the target directory's schema in some way and want to see the change(s) reflected through to the Schema tab contents in Composer.

Likewise, the **Refresh LDAP Tree** command (which has no runtime effect) is useful when you have modified the contents of a directory at design time by executing DSML requests—such as Add or Rename—and want to see the changes reflected in the Tree tab in the Native Environment Pane.

4 DSML Actions

An *action* is similar to a programming statement in that it takes input in the form of parameters and performs specific operations. It's an atomic unit of execution in a Composer component, the same way an *expression* is in Java or ECMAScript.

The list of actions you create in your component is called an *Action Model*. The Action Model is the logical core of your integration app: It is where data mapping, data transformation, and data transfer between directories and XML documents occurs. It's the ordered list of statements that comprise your app.

Some of the actions available in Composer are data-specific; others involve control flow constructs like looping, conditional branching, exception vectoring, etc. The LDAP Connect provides two main LDAP-related actions: *Create DSML* and *Execute DSML*. The usage of these actions will be described in detail in the sections to follow.

Working with DSML

DSML (Directory Services Markup Language) is an XML grammar for presenting LDAP queries and storing responses to queries. In some respects, it takes the place of LDIF, which is a text-based file interchange format for LDAP queries and objects.

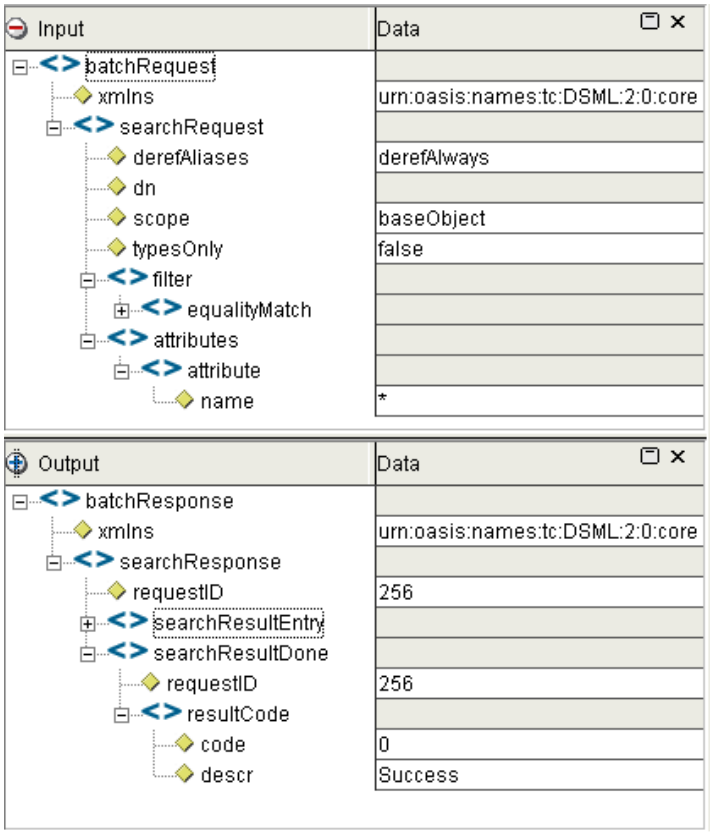
In Composer, when you want to query a directory using DSML, you have two choices:

- If you have a ready-to-go DSML file (perhaps received as input from the servlet that triggered your service), you can use it in an Execute DSML action. Composer will use the DSML to form the appropriate LDAP query, and execute that query against the target directory server.
- If you do not have a readymade DSML query document, use a Create DSML action. Composer will create the necessary DSML for you.

In the latter case, you will use point-and-click UI tools to choose the parameters for your query; Composer will then build a DSML DOM that uses your param values and display it in a DOM window. You can drag or drop data into or out of that DOM window just as with any other Composer DOM window.

When a query result comes back from the server, it comes back to your component as DSML. You can specify the target DOM for this information (Output, Temp, etc.) and you can manipulate this DOM, once again, just as you would any other DOM.

The following illustration will give some idea of the overall structure of a DSML request and response.



In this case, the response document has been mapped to Output, but it could just as easily have been mapped to Temp or some other DOM.

DSML mimics the mnemonics and operational linguistics of LDAP, so that operations specified in DSML map very closely to LDAP SDK method signatures. So for example, every response contains an element called `resultCode`, with XML attributes of `code` and `desc`. The `code` attribute value is zero on success, or else (on failure) is one of the result code values shown in Appendix B of this documentation (i.e., which in turn come from Novell's JLDAP SDK). The desc value is a string representing the plaintext explanation of what caused the error (such as "Invalid DN Syntax," which is associated with a result code of 34).

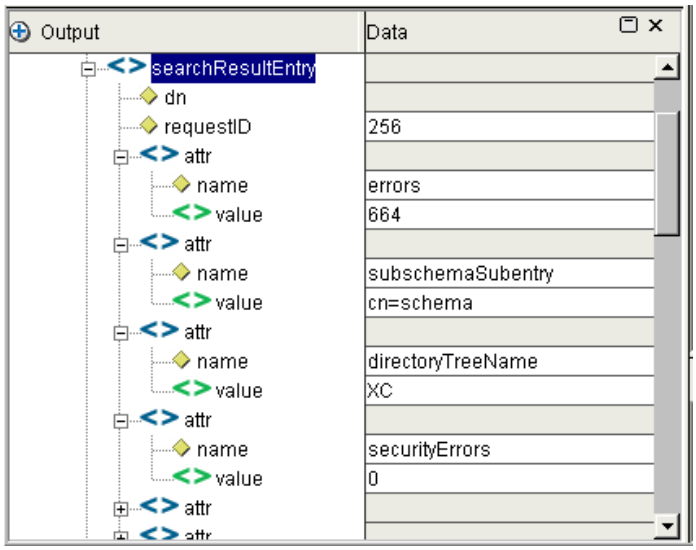
Multiple Requests in a Single DSML Document

It is possible to accumulate multiple requests within a single DSML document so that on a single server query, the server can be told to perform a series of Add operations, or Add and Delete operations, etc., in batchwise fashion.

The way this is done is to construct multiple back-to-back Create DSML actions, all of them mapping to the same root node (`batchRequest`) in the same document (typical `Input`). Each time a Create DSML action executes, it appends a new request to the specified root node. The DOM will continue to grow as necessary. Finally, this DOM is submitted as a single query when an Execute DSML action occur (presumably at the end of the list of Create DSML actions). The request executes as a batch; and a batch response is produced.

When a query response contains more than one search result, the results are accumulated under a `searchResultEntry` element node in the response document.

It is possible for a response to contain multiple results even if the request involved only one Create DSML action, since Search requests often use filters that contain wildcards, resulting in multiple "hits."



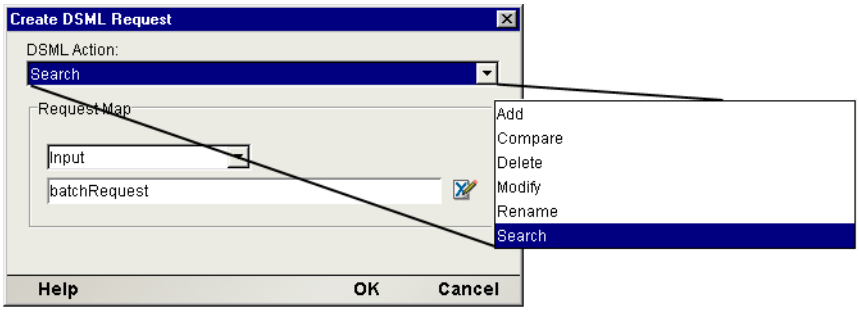
In the example shown above, there are numerous `attr` elements under the `searchResultEntry` element. The original query, which used wildcards, asked for an array of attributes. That's what was returned. Notice the many name-value pairs.

The Create DSML Action

The Create DSML Action supports the following LDAP operations:

- **Add**—adds an entry to the directory
- **Compare**—produces a boolean value based on a comparison of values (often useful for verifying the existence of a particular object or object type in a directory, so as to facilitate logical operations like "if this container doesn't already have an attribute called 'paymentHistoryRef', modify the schema to add such an attribute")
- **Delete**—removes an entry from the directory
- **Modify**—changes (edits) a value in the directory
- **Rename**—renames an entry or moves it
- Search—searches for entries based on filter criteria

These operations are shown in the Create DSML Request dialog's pull-down menu, under DSML Action:

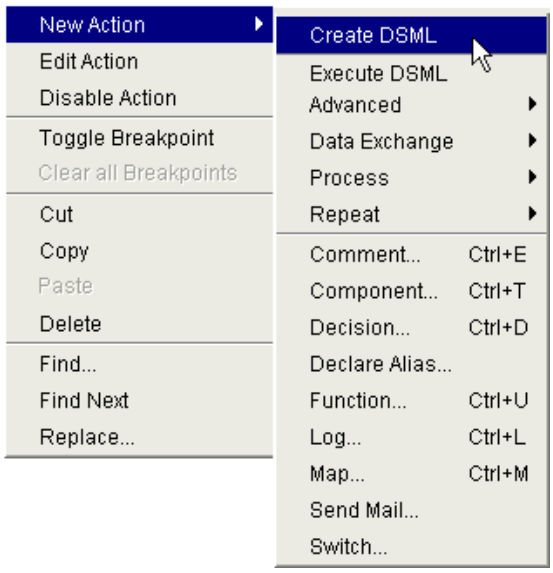


Each of these options will be discussed in turn.

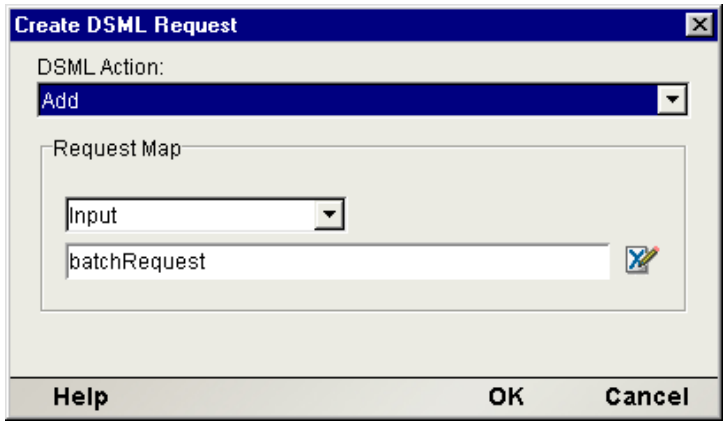
Add

➢ **To construct a Create DSML Action that performs an Add request**

1 Right-mouse-click inside the action-model pane (or use Composer's Action menu, in the main menubar) and select **New Action** > **Create DSML** from the menu, as shown here:



2 In the dialog that appears, choose **Add** from the pull-down menu.

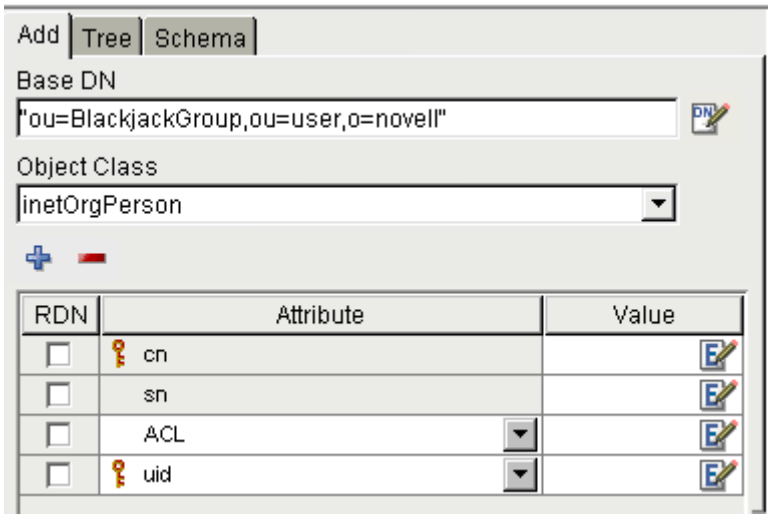


3 Use the pull-down menu under **Request Map** to specify a target DOM (or target Message). The menu will be prepopulated with the names of DOMs (Messages) you specified when you originally created the component.

4 In the text field underneath the DOM-name pull-down menu, enter an XPath expression representing the target node for the creation of the request. (In most cases, you can simply accept the default value of `batchRequest`.)

IMPORTANT: If you are going to pass this DOM straight through as a DSML query, *do not change the root node's name.* The root *must* be called `batchRequest` in order for this to remain a valid DSML document.

5 Dismiss the dialog by clicking **OK**. You should see the tabs in the Native Environment Pane update. Also, a new action with the label “Create DSML Add Request” will appear in the action model.



6 In order for this action to create a meaningful DSML request, it needs to know the Base DN of the Add request (representing the target node of the directory tree) as well as the Object Class of the new entity being added. You will supply this information in the Add tab of the Native Environment Pane (as shown above). Select the **Add** tab if it is not already selected.

7 Enter a quoted string next to **Base DN**, representing the name of the container object in which the Add will take place.

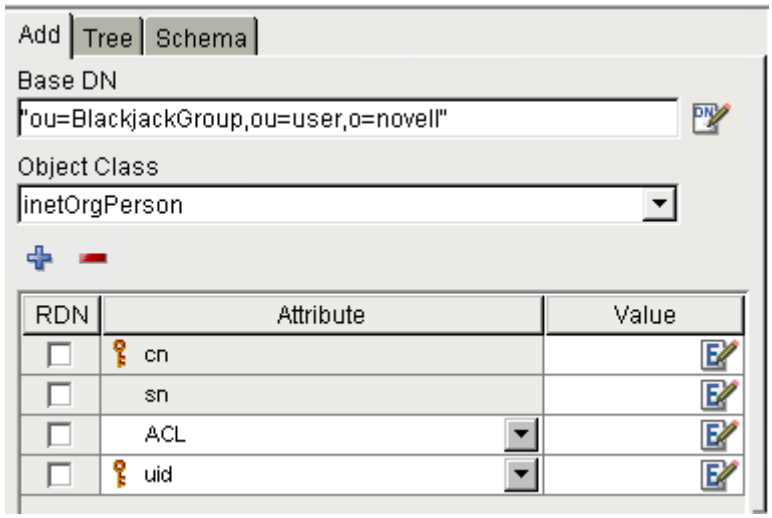
NOTE: Composer will help you build this reference: Click the DN icon to the right of the text-entry field to bring up the Expression Builder dialog. This dialog contains a tree representation of the directory like that available in the Tree tab of the Native Environment Pane. As you doubleclick items in the picktree, Composer automatically generates an appropriately formatted DN string for the target tree node.

8 Using the (prepopulated) pull-down menu under the label **Object Class,** choose the *object type* that applies to the object you will be adding to the tree. When you do this, the table of Attributes in the bottom part of the tab automatically updates to show the required naming attributes of the object-class in question.

9 The bottom portion of the panel will be prepopulated with Attributes (but not values) appropriate to the container or Object Class in question. Some attributes are *required* for the object class in question; these are shown with a solid-colored background. Some attributes (whether they’re required or merely optional) are eligible for use as naming attributes; these are shown with a small vertical key icon.

NOTE: The key iconology and background shading are available only when the LDAP server in question is powered by Novell eDirectory. In all other cases, items in the Attribute list are shown in plain-text form.

All possible combinations of visual hints are shown in the illustration below:

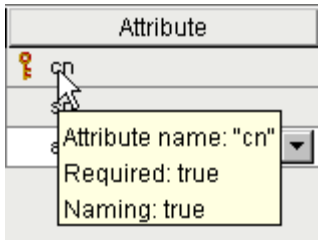


The `cn` and `sn` attributes are *required* for the `inetOrgPerson` object. This is indicated by the solid background shading in the `cn` and `sn` cells (as well as by the fact that these two attributes cannot be deleted from the Attribute column).

Two additional attributes, in this example, have been added by the user (via the blue "plus sign" button): Those attributes are `ACL` and `uid`. Note that they have *white* backgrounds, indicating that they are *not* required attributes for the object class in question.

The `uid` attribute (as well as the required `cn` attribute) has a key icon next to it, indicating that `uid` can be used as a *naming attribute* for an instance of `inetOrgPerson`. (That is, the `uid` or User ID field can be part of the person's Distinguished Name and Relative Distinguished Name.)

NOTE: A hover-help tip explains the status of any item if you let the mouse linger over the item:



10 To designate an attribute for use as part of the item's Relative Distinguished Name, check the **RDN** checkbox to the left of the attribute in question. *The mere presence of a key doesn't make the attribute part of the new entry's RDN: You must check the checkbox.*

11 If you wish to add members to the list of attributes, click the **plus sign (+)** icon to add Attribute cells. (Then choose an attribute from the pulldown menu that appears in the new cell.) To *remove* any entry, just click into (apply focus to) the entry in question and click the **minus sign (-)** to remove it.

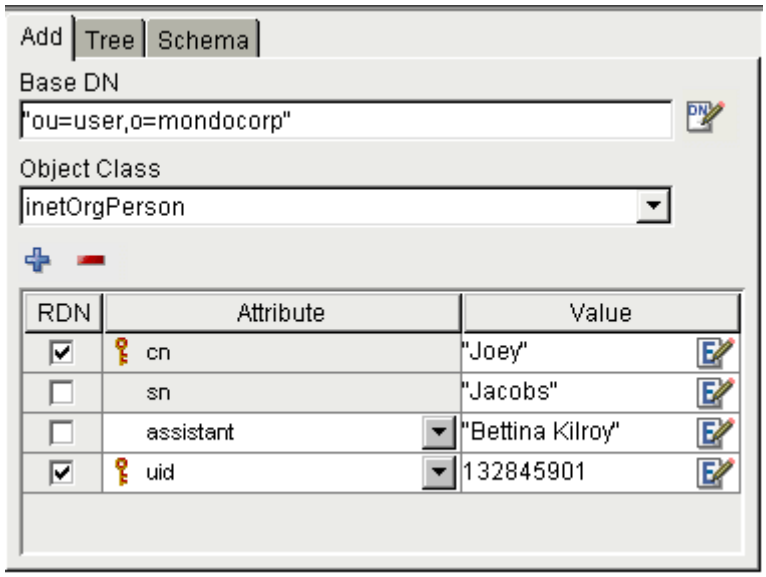
12 Be sure to associate a Value with each Attribute. The value you enter must be a valid ECMAScript string, or an expression that will evaluate to a string. If you are entering literal data, you should wrap the value in quotation marks (i.e., enter **"John"** rather than just **John**).

NOTE: You may optionally click the 'E' icon (at the right) to bring up the Expression Builder dialog, and from there, you can either type or doubleclick on picktree items to build an expression interactively.

Add Request (Detailed Example)

Suppose you have set up a container (an `organizationalUnit` object) called `user` under `o=mondocorp`.

Now suppose you are interested in adding a new `inetOrgPerson` named Joey Jacobs to the `user` object. You also want to store certain additional information about Joey, including a UserID value and the name of Joey's assistant. To add the new entry to the directory, you would construct a Create DSML action (in Add mode) as described previously, setting up your Native Environment Pane to look something like this:

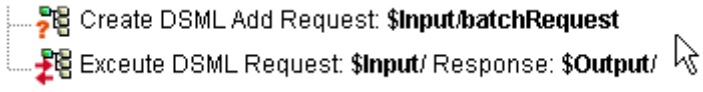


These settings say: "Add an `inetOrgPerson` with common name (cn) of Joey and surname (sn) of Jacobs, with the `assistant` and `uid` values shown, to the `user` `organizationalUnit` (ou), in the `organization` container called `mondocorp`."

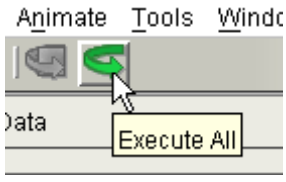
The checkboxes indicate which pieces of information to use in forming the new entry's Relative DN. In this example, the new RDN will be `cn=Joey,uid=132845901` and it will go under the container whose Base DN is `ou=user,o=mondocorp`, so the new entry's full distinguished name would be:

```
cn=Joey,uid=132845901,ou=user,o=mondocorp
```

When the Create DSML action executes, it will create DSML for this request and put it in `Input` (or whichever DOM you specified in the Create DSML Request dialog). The request still hasn't been issued to the server, however. To do that, you'll need to create an Execute DSML action. (If your request is in the `Input` DOM, just add an Execute DSML action and accept the defaults in the dialog.) Your action model will look like:

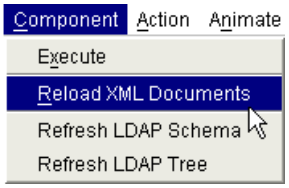


Test the actions by clicking the Execute All button in the Composer main toolbar:

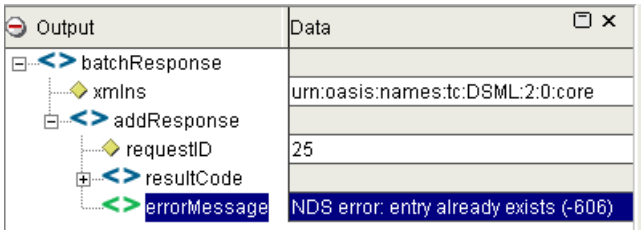


The actions will execute and you should see the Output DOM populate. A result code of 0 in the Output means that operation was a success.

To verify that the operation was, indeed, a success, go to the main menu and choose **Component > Reload XML Documents**:



Now click the Execute All toolbar icon again (re-run the component). You should see an error message in the Output DOM this time:



The `errorMessage` element contains a message to the effect that the entry you're trying to make *already exists*, indicating that the original test of the action model was successful (Joey Jacobs was added to the tree).

NOTE: This illustrates an important principle of X.500 directory architecture, which is that no two siblings (instance objects on the same level of the tree) can have the same identity. In other words, you can't add the same object twice, as we tried to do here. You will get an error on the second Add.

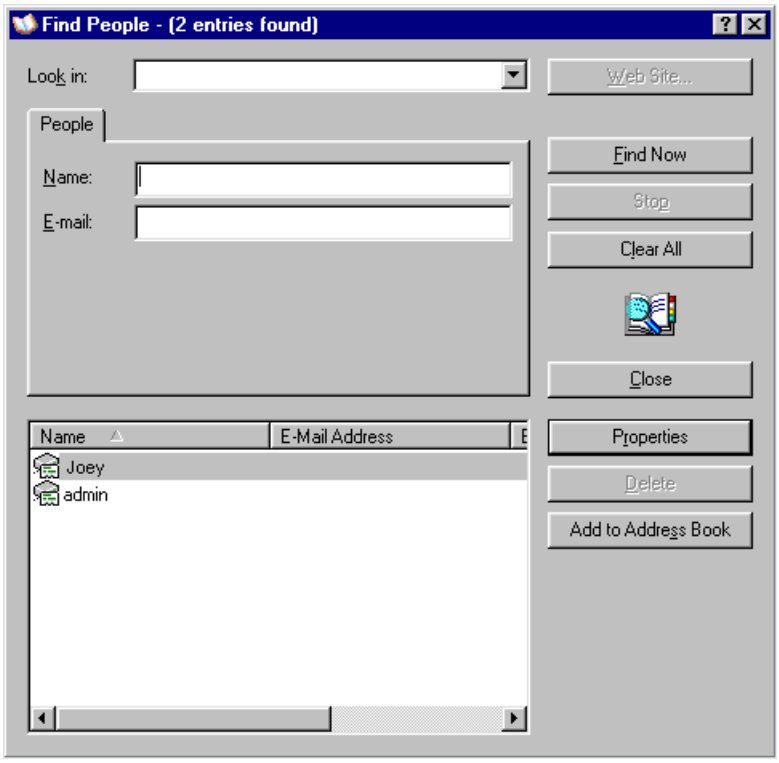
You can also verify Adds by using a third-party LDAP client—such as a web browser—to query the directory.

Most web browsers are LDAP clients and will honor the **ldap://** URL protocol. If you wanted to verify the add request that was carried out in the foregoing example, you could simply type the following URL into the address window of Microsoft Internet Explorer:

```
ldap://[server-
domain]/ou=user,o=mondocorp?cn?sub?objectClass=inetorgperson
```

The URL in effect says "Using the `ldap:` protocol, go to the server at `[server-domain]` and bind anonymously to the `user` container, then search the subtree under that container for `inetOrgPerson` objects and retrieve the attribute values corresponding to `cn` (common name)."

If you make MSIE go to this URL, an Address Book window will open, showing the search result:



Indeed, a new entry with common name John appears in the list.

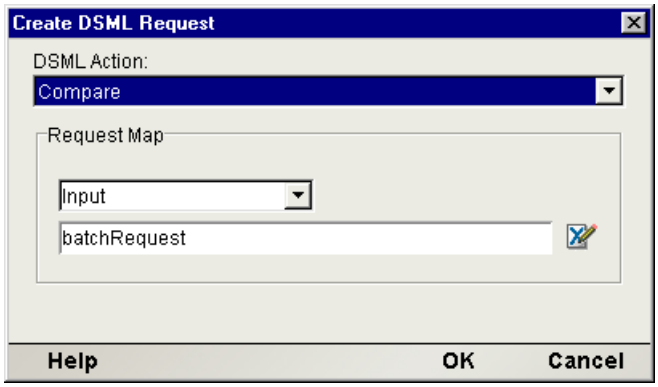
Compare

The Compare operation is valuable when you are trying to test a directory entry for existence. The procedure for setting up a Compare request is similar to that for doing an Add (as discussed above).

➢ **To construct a Create DSML Action that performs a Compare request**

1 Right-mouse-click inside the action-model pane (or use Composer's Action menu, in the main menubar) and select **New Action > Create DSML** from the menu.

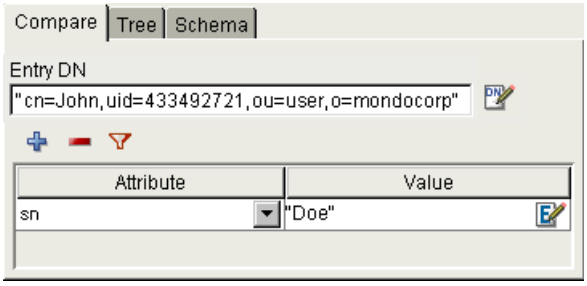
2 Use the pull-down menu control to select **Compare**.



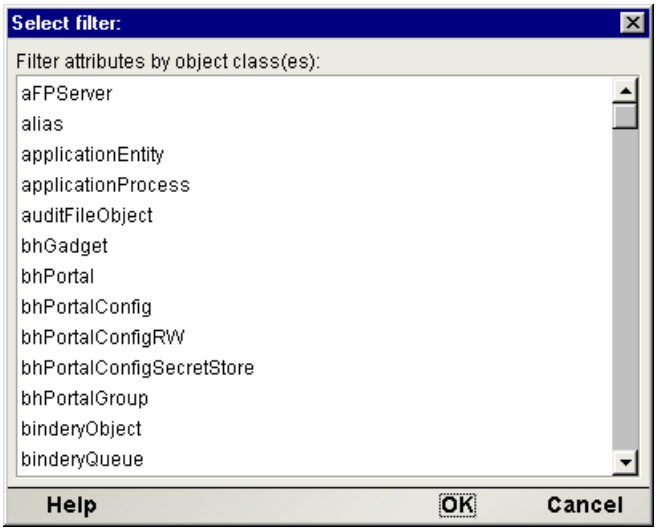
3 If you are in doubt as to what to put for Request Map values, accept the defaults shown. Click **OK**, and the action will be added to the action model.

4 Click the **Compare** tab in the Native Environment Pane if it is not already the frontmost tab.

5 Next to **Entry DN**, enter the distinguished name of the object against which you will be making comparisons. (Include quotation marks around this string as shown in the illustration below.)



6 Click the small funnel-shaped **Filter** icon (next to plus and minus icons). A dialog with a scrolling list will appear.



7 **Select** (highlight by single-clicking) the object class corresponding to the entry whose existence you want to test.

8 Dismiss the Select Filter dialog by clicking the **OK** button.

9 In the Attribute-Value table in the lower part of the Native Environment Pane, use the pull-down menu to select an attribute name. *This list of attributes corresponds to the allowable attributes for the object class you selected in the preceding step.*

10 Click the **plus-sign icon** as needed to add more entries to the Attribute-Value table. (Optional)

11 For each Attribute name, enter a corresponding value under Value.

NOTE: The Value should be wrapped in quotation marks if it is not a programmatic (ECMAScript) expression.

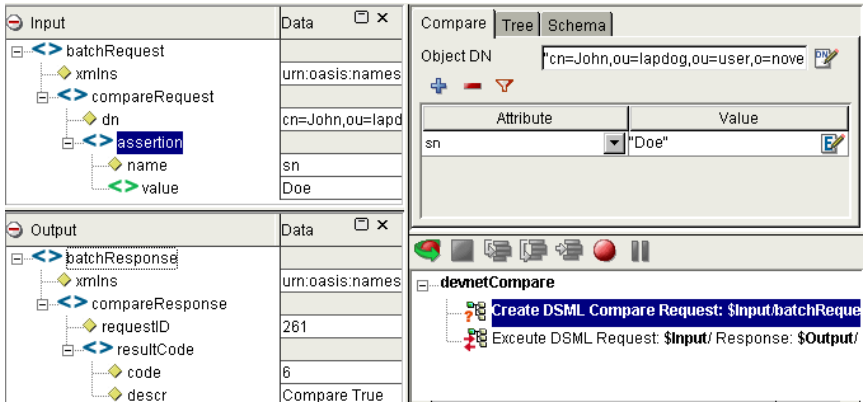
The action is now ready to test.

Compare Request Example

In the Add request example previously, we added John Doe to a container node called `lapdog` in a test directory. (Refer to that example for setup details.) To test for the existence of an entry with common name John and surname Doe, in the `organizationUnit` object called `lapdog`, one would

- Construct a Create DSML action exactly as described above
- Set up the Compare tab as shown above
- Add an Execute DSML action to the action model (accepting the defaults in the Execute DSML setup dialog)
- Run the action model either in step-through (animation) mode or by using the Execute All button in Composer's main toolbar

The following graphic shows what the component windows look like after executing the model:



Delete

The Delete operation is useful for removing entries from a directory.

The procedure for setting up a Delete request is similar to that for doing an Add (as discussed previously).

➢ **To construct a Create DSML Action that performs a Delete request**

1. Right-mouse-click inside the action-model pane (or use Composer's Action menu, in the main menubar) and select **New Action > Create DSML** from the menu.
2. Use the pull-down menu control to select **Delete**.
3. Click the **Delete** tab in the Native Environment Pane to bring it forward, if it is not already the frontmost tab.
4. In the Delete tab, next to **Entry DN**, enter the unique DN (distinguished name) of the entry you wish to delete.

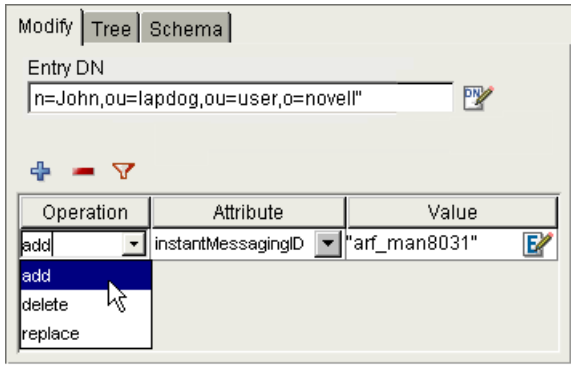
The action is now ready to use. Executing it will append an appropriate request to the DSML target document.

Modify

The Modify operation is useful for adding, deleting, or replacing values in a directory. For example, if your directory already contains an entry for John Doe but does not contain an entry for his instant messaging ID, you could use the "add" flavor of the Modify request to add his IM handle to the directory. This is illustrated in the following procedure.

➢ **To construct a Create DSML Action that performs a Modify request**

1. Right-mouse-click inside the action-model pane (or use Composer's Action menu, in the main menubar) and select **New Action > Create DSML** from the menu.
2. Use the pull-down menu control to select **Modify**.
3. Click the **Modify** tab in the Native Environment Pane to bring it forward, if it is not already the frontmost tab.
4. In the Modify tab, next to **Entry DN**, enter the unique DN (distinguished name) of the entry you wish to modify.



5. Click the small funnel-shaped **Filter** icon (next to plus and minus icons). A Select Filter dialog with a scrolling list will appear.
6. In the scrolling list, **click on** the object class corresponding to the entry you want to modify.
7. Dismiss the Select Filter dialog by clicking its **OK** button.
8. In the Modify tab of the Native Environment Pane, click the **plus-sign** icon to add an Operation to the Operation-Attribute-Value table.
9. Select the type of operation you wish to perform (**add**, **delete**, or **replace**) from the pull-down menu as shown above.
10. In the **Attribute** column, use the pull-down menu to choose the attribute that corresponds to the one whose value you intend to add, delete, or replace. (The attribute list is prepopulated with attributes that are appropriate to the object class you chose in the Select Filter dialog earlier.)
11. In the **Value** column, enter the appropriate value for the attribute.

    The action is now ready to use. Executing it will append an appropriate request to the DSML target document.

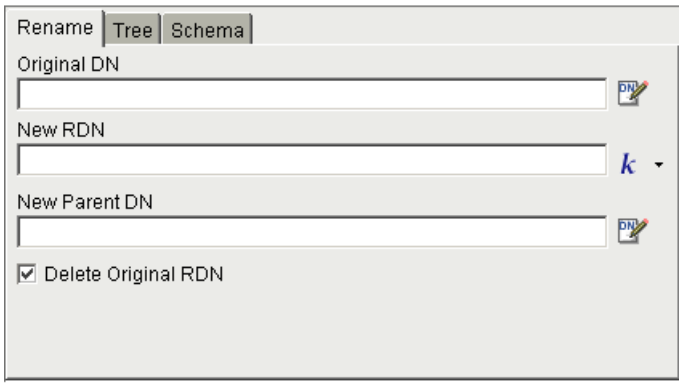
Rename

The Rename operation can either rename or move an entry in a directory, depending on how it is used.

The screenshot in the following procedure illustrates changing the common name (`cn`) for someone named John to a `cn` of Johann.

➢ **To construct a Create DSML Action that issues a Rename request**

1. Right-mouse-click inside the action-model pane (or use Composer's Action menu, in the main menubar) and select **New Action > Create DSML** from the menu.
2. Use the pull-down menu control to select **Rename**.
3. Click the **Rename** tab in the Native Environment Pane to bring it forward, if it is not already the frontmost tab.
4. In the Rename tab, next to **Old DN**, enter the unique DN (distinguished name) of the entry you wish to rename or move.
5. Next to **New RDN**, enter the new relative DN for this entry. (You do not need to use quotation marks around the value if the *k* icon is showing at the right. See the explanation of this icon in the procedure for Add, above.)



6. Optionally enter a new parent-container DN next to **New Parent DN**, if you are moving the entry.
7. Optionally check the **Delete Old RDN** checkbox if you are moving the entry to a new location and don't want to keep the old copy in the old location.

   The action is now ready to use.

Search

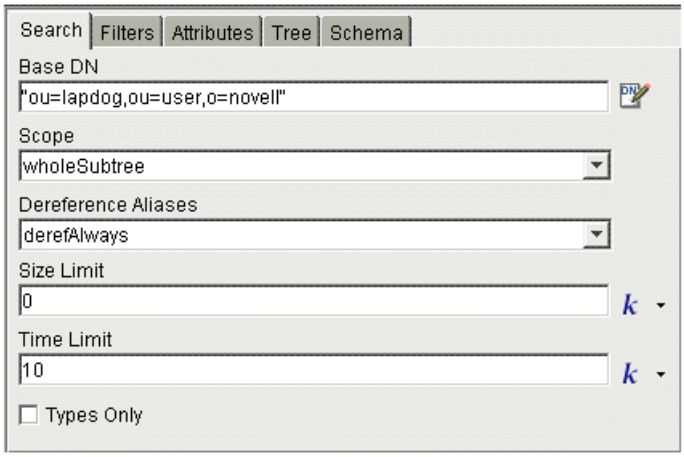
The Search operation can be used to retrieve data from a directory. LDAP defines a query filter syntax involving logical operations that can be combined to build complex conditional search criteria. The syntax is covered in RFC 2254 and won't be recapitulated here since Composer can automatically generate the correct expression syntax for you based on your use of UI controls. (See below.)

NOTE: If you are familiar with the rule-builder GUIs offered in e-mail clients for constructing mail filtering conditions, the same principles (and GUI metaphors) apply here.

➢ **To construct a Create DSML Action that issues a Search request**

1. Right-mouse-click inside the action-model pane (or use Composer's Action menu, in the main menubar) and select **New Action > Create DSML** from the menu.
2. Use the pull-down menu control to select **Search**.

3 Click the **Search** tab in the Native Environment Pane to bring it forward, if it is not already the frontmost tab.



4 In **Base DN**, enter the distinguished name of the container object where you want the search to start. (Be sure to wrap this string in quotation marks.)

5 For **Scope**, choose one of the three available values:

- **baseObject**—means to limit the search to the base object (specified in Base DN) only
- **singleLevel**—mean the search will be scoped to the base object plus its immediate (one-level-deep) children
- **wholeSubtree**—means the search will include the base object as well as all children under the Base DN object, no matter how many levels deep the children might be

6 For **Deref Alias**, choose one of the following values from the pull-down menu:

- **derefAlways**— means that aliases (entries that contain a pointer to another entry) are always dereferenced, both when finding the starting point for the search, and also when searching the entries beneath the starting entry
- **derefFindingBaseObject**— means that aliases should be dereferenced (resolved) when finding the starting point for the search, but not when searching under that starting entry
- **derefInSearching**— means aliases should be dereferenced when searching the entries *beneath* the starting point of the search, but not when finding the starting entry
- **neverDerefAliases**— means not to look at aliased entries

7 For **Size Limit**, enter a numeric value representing the maximum number of "hits" the component is willing to receive from the server. (Note that the server may also have its own response-batch-size limits, which can't be controlled from the client, generally speaking.) The default value of zero means there is no limit on the number of items that can be returned and the component will simply block until all results come in.

TIP: Use the flyout menu to select *k* for constant or ECMA Expression if you want to enter a programmatic (ECMAScript) expression that resolves to a number at runtime. See description under the "Add" procedure further above.

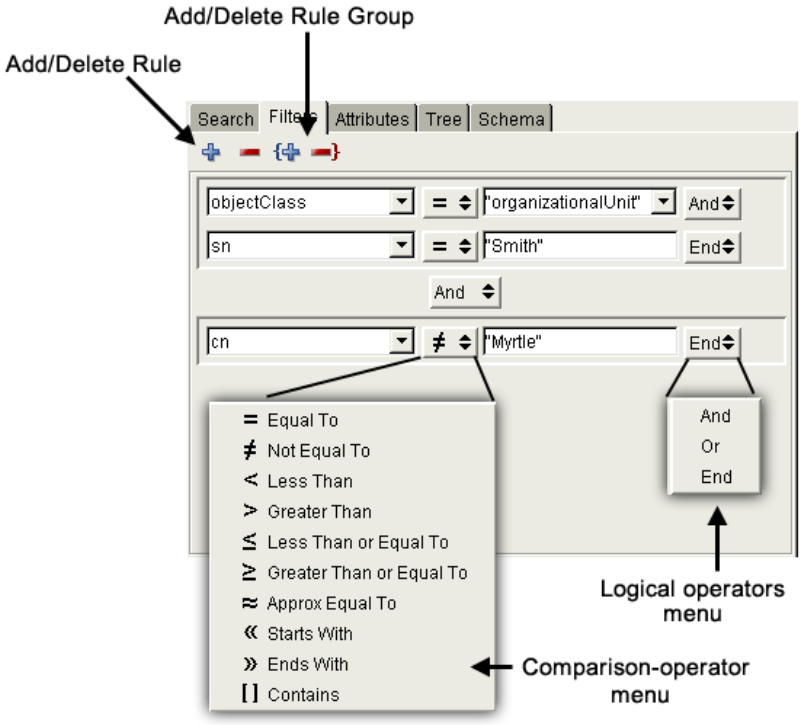
8 For **Time Limit**, enter a numeric value representing the maximum number of *seconds* (not milliseconds) the component is willing to wait while the server processes the request. This tells the server that it should stop searching and return an error Leaving this value blank (or zero) means there is no upper limit on wait time; the client (your component) is content to wait forever.

TIP: Needless to say, it is not a good idea to wait forever. Enter a prudent value here.

9 Optionally check the **Types Only** checkbox if you want the search to return the *names* of any attributes found, but not their associated values.

10 Click the **Filter** tab to bring it forward.

11 Use the various controls to build rules for filtering the search. (See below.)

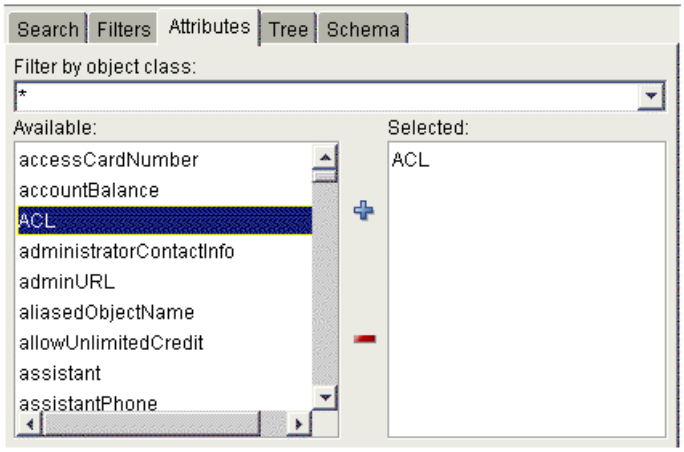


The settings in this particular graphic show a rule that says: "*objectClass* equals *organizationalUnit* AND surname equals Smith; AND common name does not equal Myrtle."

12 Click the **Attributes** tab to bring it forward.

13 Use the pull-down menu under **Filter by object class** to select an object class name, thereby exposing a list of the allowable attribute types for that object class in the Available subpane (lower left).

14 Use the plus and minus icons to move attribute names from the Attribute column to the Selected column or vice versa.



In this example, the `isManager` and `fullName` attributes have been selected from the list of allowable `organizationalPerson` attributes. What this means is that the search results will contain `isManager` and `fullName` attributes (and their values) for the "hits," if any.

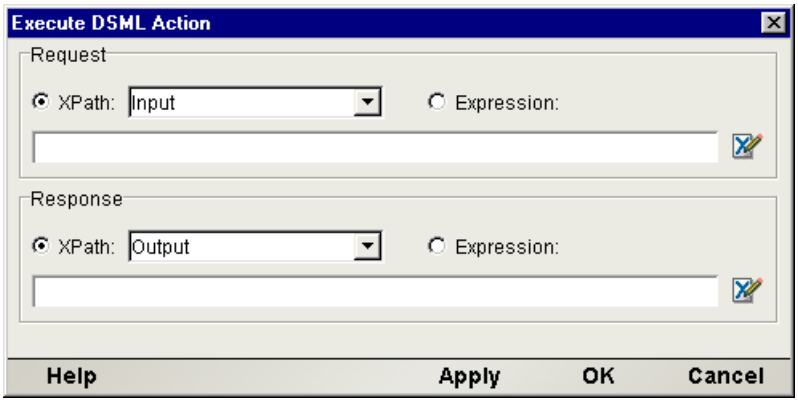
The Execute DSML Action

The Execute DSML action converts a DSML-packaged request to a low-level LDAP request, and executes it (synchronously) against the server. It retrieves any response data received from the server and maps it to the XML document of your choice (typically the Output DOM or a Temp DOM).

To set up a valid Execute DSML action, you have to tell Composer where the DSML request document (which should be an in-memory DOM) can be found, and where to map the DSML response that comes back after a request is made.

➢ **To create an Execute DSML Action**

1 Right-mouse-click inside the action-model pane (or use Composer's Action menu, in the main menubar) and select **New Action** > **Execute DSML** from the menu. A dialog appears.



2 In some cases, there is no need to alter the default settings in this dialog and you can close the dialog immediately by clicking **OK**. What this does is tell Composer: "Just use the DSML in `Input` for the request, and map the response to `Output`." If this is not what you want, specify the correct Request and Response DOMs (and target nodes) using the radio buttons, pull-down menu controls, and text fields provided. (For example, you might want to map the response to Temp, Temp1, or some other "scratchpad DOM" rather than straight to Output.)

NOTE: These controls will be familiar to you if you have created Map Actions in Composer before. Consult the *Composer User's Guide* for information on how to specify XPath-to-XPath mappings if you have not done this before.

3 Click **OK** to dismiss the dialog. The new Execute DSML action is added to your action model.

Using Other Actions in the LDAP Component Editor

In addition to the DSML-related actions described so far, you have all the standard Basic and Advanced Composer actions at your disposal as well. The complete listing of Basic Composer Actions can be found in Chapter 7 of the *Composer User's Guide*. Chapter 8 contains a listing of the more Advanced Actions available to you.

5 Working with LDAP and DSML

This chapter is designed to familiarize you with LDAP programming idioms as they apply to the LDAP Connect for Composer, and show how various advanced LDAP and Composer features can be used together, with emphasis on testing and debugging.

DSE Query Example

A useful discovery mechanism (and audit capability) afforded by LDAP is the ability to query the DSE root, or DSA-specific entry, of a directory server. (DSA means *directory system agent*, which is X.500 jargon for a directory *server*.)

If a server supports LDAP version 3, the DSE root can be queried for “meta” information about the server. The DSE root entry isn’t really an addressable object, per se, but it can be queried through a special syntax (which we’ll see in a minute).

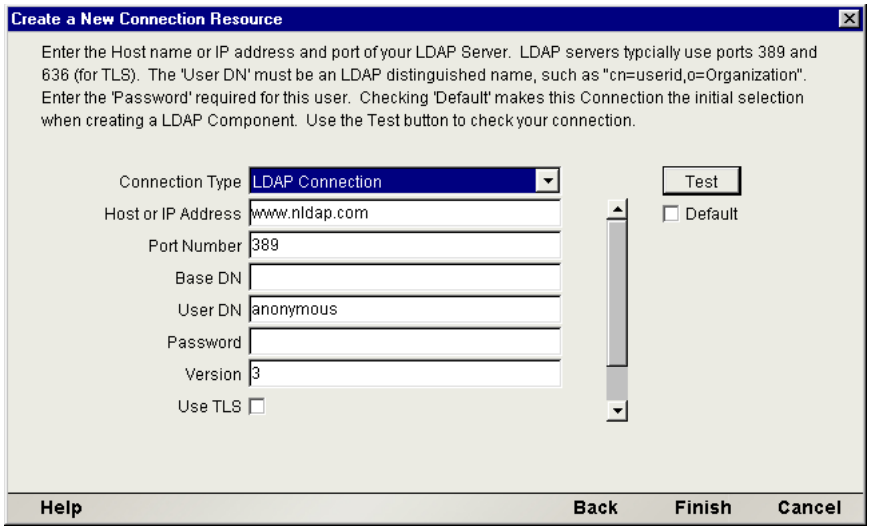
By querying the DSE entry, you can find out (among other things):

- Supported security mechanisms
- Supported extensions
- Implementation (vendor version) info
- The number of simple-authentication and strong-authentication binds that have occurred since server startup
- The location of the schema for the subtree

The following example shows the steps involved in building a component that queries the DSE entry of Novell’s public server at **www.nldap.com**. But instead of recapitulating the detailed step-by-step procedure for building each action and each resource (already covered in previous sections), we’ll concentrate on the larger workflow issues related to building an LDAP Component, such as how to test and debug the component as part of the design session.

Connection Resource for Anonymous Bind

First, we need to have a connection resource. The LDAP Connection Resource for this example will use the following settings. Notice that no password is given. The bind is thus (by definition) anonymous.



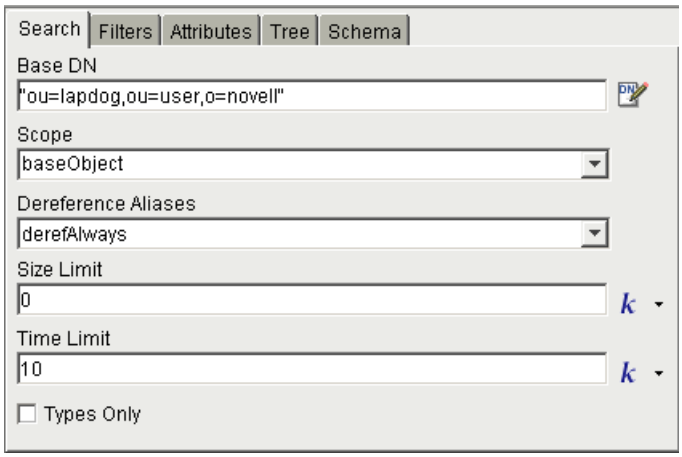
Component and Action Model

Next, we'll create a new LDAP Component called DSETest, containing two actions: a Create DSML Action (of the Search request type) and an Execute DSML Action.

The search parameters for a DSE root dump are straightforward: All that's needed is a base-level search on an empty Base DN, with a filter set to:

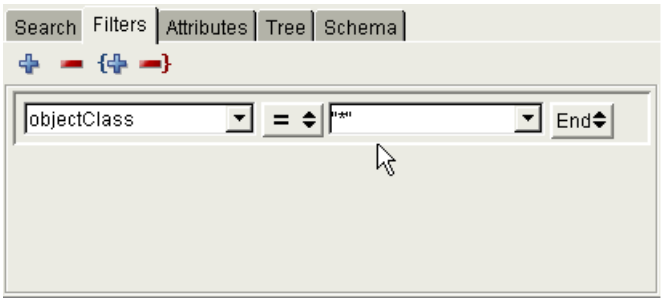
```
(objectClass=*)
```

We will set the Native Environment Pane's Search tab something like this:

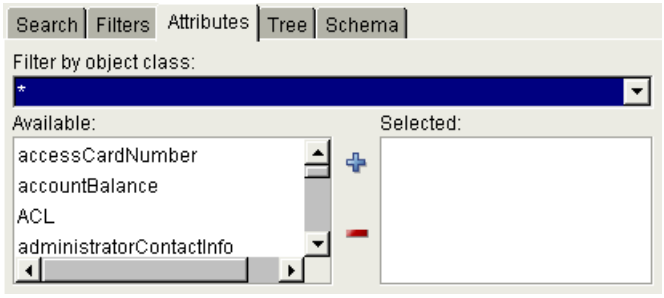


NOTE: This example purposely contains a bug. The settings shown above will cause an error. (Can you spot the problem?)

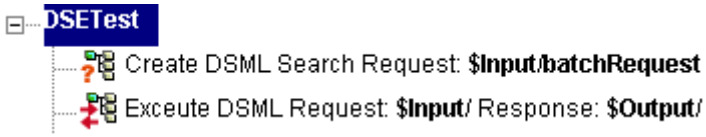
The Search tab will look like:



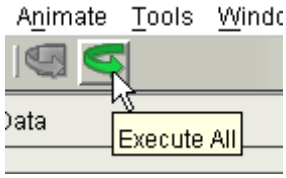
We want to receive information on *all* attributes, hence we will enter nothing in the Attributes tab. (Just accept the defaults.) Notice that the **Selected** pane (below, right) is empty. When this list is empty, the server, by default, returns *all* attributes that apply to the "hits" turned up by the query.



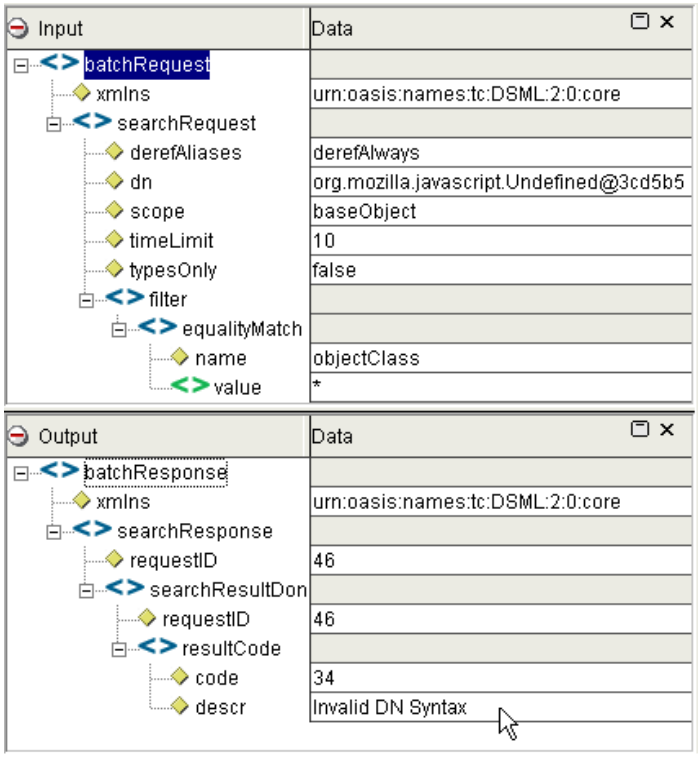
For test purposes, we will simply map DSML (from the Create action) into Input and map our response straight to Output. Therefore, the action model will initially look like this:



To test the actions, we can simply click the Execute All tool icon in the main toolbar.



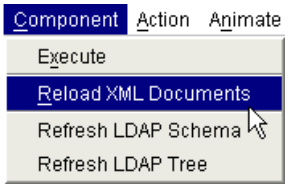
When the component executes, the Output DOM populates with data. But on close inspection, we see that there has been a problem. The Output document looks rather short and the `resultCode` is not zero:



The error description (at the bottom of the Output DOM) is “Invalid DN Syntax.” And this is reflected in the Input DOM tree, next to `dn`, under `searchRequest`, where the data value is `org.mozilla.javascript.Undefined@3cd5b5`.

The problem: The Base DN value in the Search tab has to be valid ECMAScript. We left the field blank. Instead, we should have specified an empty string: two quotation marks with nothing in between them.

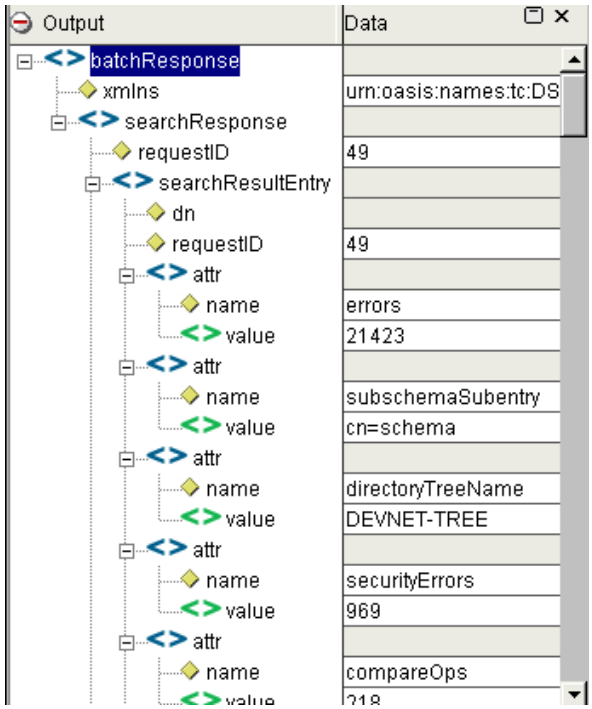
The fix is easy: Go back and put two quotation marks in the Base DN field. But before pressing the Execute All button again, we need to clean out the existing DOMs. To do this, go to the main menubar and choose **Component > Reload XML Documents**:



The DOM windows will reset to their original (empty) state.

NOTE: If you fail to reset the DOMs, the next round of execution will cause the Create DSML action to simply append more data (another request) onto the existing Input document. After the component executes, your Output document will contain the responses for two requests: the original (unsuccessful) request and the one that was appended to Input.

When we rerun the component, this time it works without error. The Output DOM is quite long and contains many `attr` elements corresponding to DSE root entries:



Dealing with Errors

As the above example shows, unsuccessful LDAP or DSML operations don't result in errors, per se, at the action-model level. If an LDAP exception occurs in the context of a query, it's merely reported back to the client in the DSML document under the `resultCode` element. Your application may want to know about the exceptional condition and take special action; or it may simply need to pass the information on through. If it needs to know about the unsuccessful nature of the query, you will need to add logic to deal with this.

As an example, suppose you want your application to log unsuccessful DSML requests. One way to implement this would be to include logic in the action model to the effect: "If the result was anything other than success, log the reason to System output [or a logfile]."

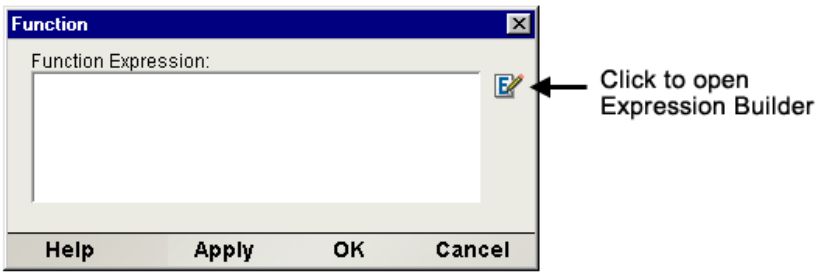
Here is how you could accomplish this:

➢ **To log query failure notices**

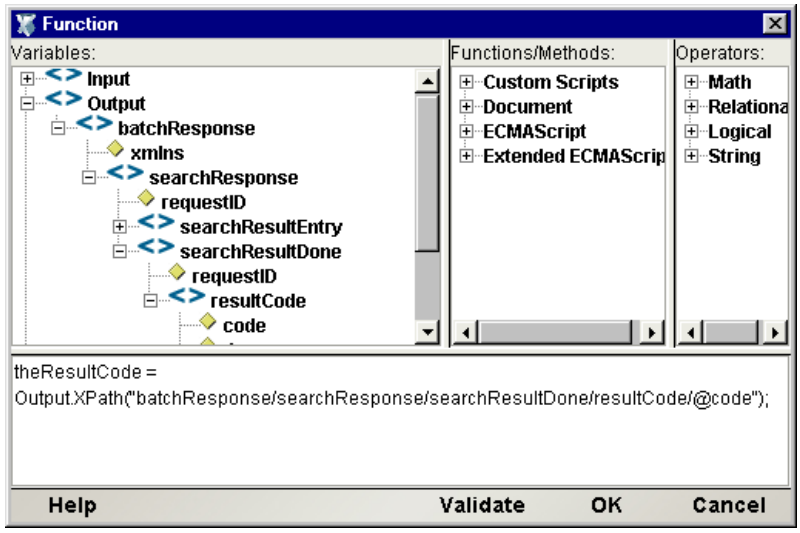
1 Position the cursor inside the action pane (action model) at the point where you want a new action to appear.

2 Use the **Action** menu in the main menu bar to select **New Action > Function**. The Function Action dialog appears.

3 Click the small 'E' icon to open the Expression Builder dialog.



4 In the Expression Builder, click into the `Output` node tree until you have exposed the `resultCode` node and its children. Doubleclick the `code` attribute node. An ECMAScript expression appears in the text-edit pane.



NOTE: This example assumes that you have a DSML document already loaded in your Output DOM. Construct and run a Create DSML and Execute DSML action, if need be, so that you will have a fully browseable Output DOM (similar to that shown) in the Expression Builder picktree.

The expression gives the XPath syntax for the resultCode, wrapped in a call to the Composer extension method `XPath( )`. The Expression Builder gives you a visual, point-and-click way of building XPath-related ECMAScript expressions so that you don't have to type them in (and debug them) manually.

5 Type "theResultCode = " (minus quotes) in front of the ECMAScript expression. (This is just a shorthand way of storing the result code value in a script variable.)

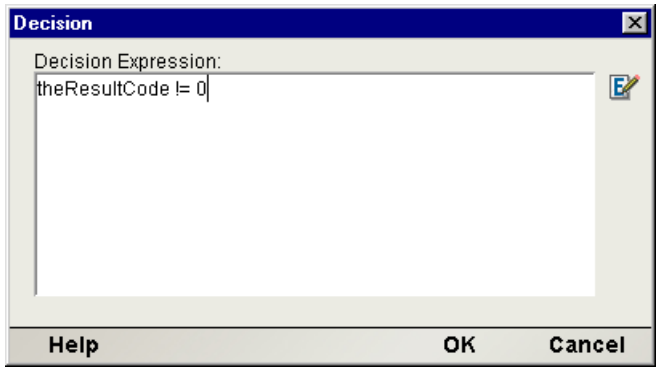
6 Click **OK** to go back to the Function Action dialog. The edit window of the dialog should contain the statement:

```
theResultCode =
Output.XPath("batchResponse/searchResponse/searchResultDone/resultCode/@code");
```

(It should be all one line. Ignore the linewrap.)

7 Click **OK** or type Enter to dismiss the Function Action dialog. A new action appears in the action model.

Use the **Action** menu in the main menu bar to select **New Action > Decision**. The Decision Action dialog appears.

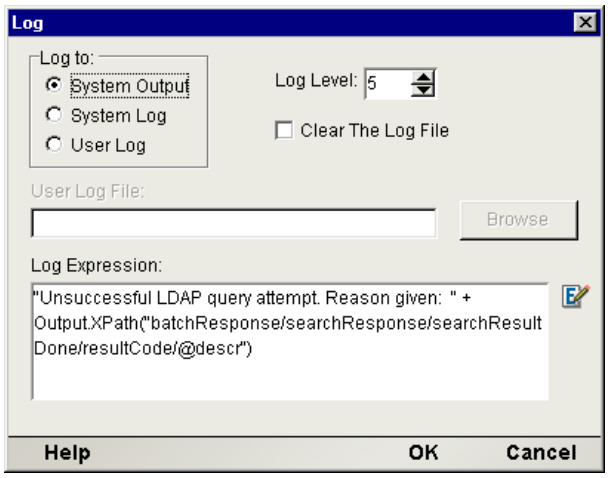


8 Type "theResultCode != 0" (as shown above) into the box. This is the truth condition for the Condition. If `theResultCode` is zero (success), the condition is false. Any value other than zero will make the condition true.

9 Click **OK** or type Enter to dismiss the dialog. A new Decision action appears in the action model.

10 TRUE and FALSE statements will appear on their own lines after the Decision action. Single-click on the TRUE line.

11 Use the **Action** menu in the main menu bar to select **New Action > Log**. The Log Action dialog appears.



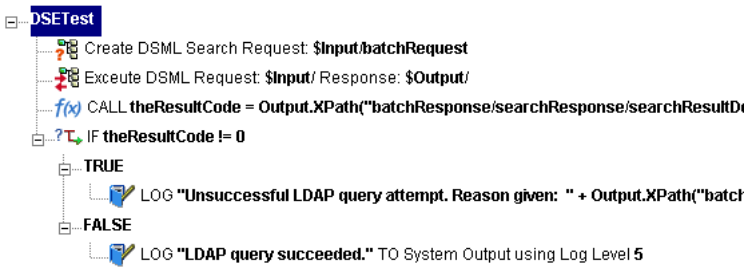
12 Choose one of the radio buttons under Log to, to specify a destination for any messages logged by this action. We've chosen System Output for test purposes, since (in Composer) this means the log message will show up in the Log tab at the bottom of the main Composer window, where it can be seen easily during design-time testing. For runtime purposes, you would probably want to (re)set this to a User or System Log.

13 Enter whatever descriptive text you would like to appear in the log message for this action. (Remember, this is in the TRUE branch, which means this message will be logged whenever the query result was non-zero—i.e., non-successful.) In this example, we used the Expression Builder to help construct an ECMAScript statement that finds the plaintext error message in the Output DSML document.

14 Click **OK** to dismiss the dialog.

15 Single-click on the FALSE line in the action model under the Decision action, and repeat the foregoing steps to create a Log Action for the FALSE branch, if desired. (This is the branch corresponding to success, not failure, of a query.)

The action model, at this point, should look something like this:



16 In the main menu bar, choose **Component > Reload XML Documents** (to purge your DOM windows), then Execute the component, or its individual actions, to test it.

17 **Save** your work.

ECMAScript and the LDAP Connect

Composer's ECMAScript binding provides a powerful and flexible tool for performing programmatic operations of various sorts, including straight-through calls to Java objects. Since the LDAP Connect for Composer uses Novell's JLDAP library (already included in the CLASSPATH for your projects), you can easily call straight into the Java-level LDAP API. An example of this is shown further below.

LDAP Extension Methods

When you are using the LDAP Component editor to create or edit an action model, several LDAP-related Composer extension methods are available for use in Function actions and other places where script expressions are permitted. The method you will use most often is `getLDAPAttr( )`, described below.

`getLDAPAttr(String connResource, String dn, String attr)`—Looks up a value stored in a particular attribute of a named object in an LDAP directory, using the connection resource whose name is supplied in the first argument. The second argument is the object's LDAP distinguished name. The third arg is the attribute of interest. The value returned may be numeric or String data. Use ECMAScript's `typeof` operator to determine if the value is of type "number" versus type "string."

Access Control List (ACL) Methods

Access Control Lists are used in Novell eDirectory as a way of managing access to directory information based on identities and privileges. Access control is implemented by an optional, multivalued attribute called ACL, which is defined on the top-level directory object called "top." Since all directory objects inherit from the top object, all objects can use the ACL attribute.

Each value stored in a tree entry's ACL attribute represents information about a *trustee* (a different object) whose access to the entry is to be controlled. In other words, the ACL stores information about *client objects* (accessors) rather than about the data-store object itself.

Composer's LDAP Component Editor does not expose UI tools for managing ACLs and privilege sets—for good reason. ACLs are at the heart of directory security. ACL-editing capability is not something you'd want to expose in a web service. Nevertheless, there may be situations in which it is necessary or desirable to set trustee characteristics on an object "under the covers," as part of the normal execution of a service. You can do this in Composer with ECMAScript.

Two ECMAScript extension methods are available for creating ACL (Access Control List) values that can be attached to objects in an eDirectory tree.

NOTE: These methods are relevant only to Novell eDirectory and NDS-compliant directory servers.

```
ndsACL.createEntryACL(boolean abBrowse, boolean abAdd, boolean abDelete,
boolean abRename, boolean abSupervisor, boolean abInherit, String asTrusteeDN)
```

Creates a properly formatted ACL value that can be added to the (optional, multivalued) ACL attribute of any object in a Novell eDirectory tree. The return value of this method represents a specific set of Access Control List privileges applicable to a particular "asTrusteeDN" accessor. The first six parameters are true/false flags indicating the rights to be granted. The "abInherit" argument determines whether children of the trustee object should inherit the rights of the parent.

```
ndsACL.createAttrACL(boolean abCompare, boolean abRead, boolean abWrite,
boolean abSelf, boolean abSupervisor, boolean abInherit, String asTrusteeDN,
String asProtectedAttrName )
```

Similar to the previous method, but creates access control policy for a given *attribute* on an existing object.

The general procedure for using these methods is:

- In a Function action, execute the method of interest, obtaining the return value in an ECMAScript variable. For sake of this discussion, assume that the variable's name is **aclEntry**.
- In the LDAP Component Editor, add a Create DSML Action to the action model. Specify "Modify" as the query type.
- In the native environment tabs for the action just created, add an Add operation to the Modify, targeting the particular DN (directory entry) of interest. Add an ACL attribute on this DN, with an ACL value of **aclEntry** (your ECMAScript variable).
- Execute the DSML query.

An example will help make this clearer: Suppose you have, in your eDirectory tree, an `inetOrgPerson` whose common name (CN) is Bob. You want Bob's `telephoneNumber` attribute to be exposed to another `inetOrgPerson`, named Jill. To do this, you would create an access control value using `ndsACL.createAttrACL()`, specifying Jill's DN as the trustee DN and `telephoneNumber` as the protected attribute name. In a Modify operation, you add the access control value to Bob's ACL attribute. (Remember, ACL is a multivalued attribute. Bob may have scores or even hundreds of values stored under his ACL attribute.) Jill would then have certain specific access rights to Bob's telephone number.

CAUTION: *ACLs are a critical component of eDirectory security. You should have a thorough understanding of ACL concepts before attempting to use the above methods. Misuse of ACL methods can cause security to be compromised at many different levels of a directory tree. Do not attempt ACL modifications unless you are confident that you thoroughly understand NDS security concepts and their implications.*

For a technical overview of ACL concepts, consult the **NDS Technical Overview** document (in particular, the chapter on eDirectory Security) available online at **http://developer.novell.com/ndk**.

Access to Novell LDAP Classes

The Novell Java LDAP (JLDAP) library classes, which are publicly available as part of the Novell NDK, are already a part of Composer (design-time as well as runtime) and are already in the classpath. Therefore you can write code of the following sort in Composer's Custom Script Resource editor (or in the Expression Builder dialogs, etc.):

```
myConn = new Packages.com.novell.ldap.LDAPConnection();
```

Notice that in ECMAScript, access to the Java classloader occurs via the Packages keyword. Hence, to access any class that is visible on the classpath, you need only prepend "Packages" to the qualified class name, as shown above. Since ECMAScript variables do not require a data type declaration, it is perfectly legal to execute the above line of code as-is. It both declares *and* initializes the variable **myConn** in one operation.

ECMAScript Example Involving LDIF

This section shows an example of how to use ECMAScript to call Novell LDAP classes directly, so as to accomplish tasks that would otherwise not be possible in the LDAP Component Editor. The example shown below involves LDIF.

Prior to the advent of DSML, directory users relied heavily (and still do) on the text-based LDIF (LDAP Data Interchange Format) file type for persisting LDAP objects and queries. (The LDIF specification is published in RFC 2849.) LDIF is convenient for many of the same reasons XML is: It's human readable, structured according to relatively simple rules, platform-neutral, etc. Many administrative tools are able to handle LDIF natively.

It is useful to compare the DSML and LDIF versions of a query result. The DSML version might look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<batchResponse xmlns="urn:oasis:names:tc:DSML:2:0:core">
  <searchResponse requestID="160">
    <searchResultEntry dn="cn=admin,ou=lapdog,ou=user,o=NOVELL" requestID="160">
      <attr name="cn">
        <value>admin</value>
      </attr>
      <attr name="loginTime">
        <value>20030315051622Z</value>
      </attr>
      <attr name="objectClass">
        <value>inetOrgPerson</value>
        <value>organizationalPerson</value>
        <value>person</value>
        <value>top</value>
        <value>ndsLoginProperties</value>
      </attr>
      <attr name="securityEquals">
        <value>ou=lapdog,ou=user,o=NOVELL</value>
...
```

[etc.]

The LDIF version of the same thing would look like this:

version: 1

dn: cn=admin,ou=lapdog,ou=user,o=NOVELL

cn: admin

loginTime: 20030315051622Z

objectClass: inetOrgPerson

objectClass: organizationalPerson

objectClass: person

objectClass: top

objectClass: ndsLoginProperties

securityEquals: ou=lapdog,ou=user,o=NOVELL

. . .

[etc.]

The structure of an LDIF file is not unlike that of a simple flat file with one record per line and each record representing an attribute-value pair delimited by a colon. (The syntax isn't quite *that* simple, but close enough for this discussion.)

For audit purposes, it's sometimes convenient to obtain an "LDIF dump" of a particular tree object. It doesn't take much ECMAScript to do this.

To programmatically query a directory and write the results to an LDIF file, start by creating a Custom Script Resource (using Composer's main menu bar, go to **File > New > xObject** then open the **Resource** tab and select **Custom Script**). Inside the custom script editor window, enter a custom function like that shown below.

```
function query2Ldif( ldapHost,
          loginDN,
          password,
          searchBase,
          scope,
          searchFilter,
          attribs,
          fileName )
{
       var jldap       = Packages.com.novell.ldap;
       var ldapPort    = jldap.LDAPConnection.DEFAULT_PORT;
       var ldapVersion = jldap.LDAPConnection.LDAP_V3;
       var typesOnly   = false;

       var lc = new jldap.LDAPConnection();

        // An ECMAScript array reference cannot
        // be passed to a Java method that expects
        // a Java array. So we have to copy our
        // 'attribs' array into a legit Java array:
       var javaStringArray =
           java.lang.reflect.Array.newInstance(
                       java.lang.String,
                       attribs.length);
       for (var i = 0; i < attribs.length; i++)
           javaStringArray[i] =
                   new java.lang.String(attribs[i]);


        // bind to server
       lc.connect( ldapHost, ldapPort );
       lc.bind(ldapVersion, loginDN, password );

        // set up file I/O objects
       var fos = new java.io.FileOutputStream(fileName);
       var writer = new jldap.util.LDIFWriter(fos);

        // send query
       var results = lc.search( searchBase,
            scope,
            searchFilter,
            javaStringArray,
            typesOnly);

        // write the results
       while ( results.hasMore() )
               writer.writeEntry( results.next() );

        // clean up
       writer.finish();
       fos.close();
       lc.disconnect();

       java.lang.System.out.println("LDIF file written.");
}
```

This function is purposely somewhat monolithic in order to show all of the intended functionality in one complete series of related steps. (In the real world, you'd probably factor this function out into two smaller functions: a function that does the bind and issues the query, and another that writes the query result out to an LDIF file.)

The eight arguments represent the standard parameters needed to bind an LDAP directory plus those needed to form a query.

The first line inside the function is:

```
var jldap = Packages.com.novell.ldap;
```

This lets us use a shorthand notation (of `jldap`) for the longish context string that begins with Packages. (ECMAScript offers access to Java methods through the Packages indirection mechanism.) After we've established this shorthand, we can do

```
var lc = new jldap.LDAPConnection();
```

instead of

```
var lc = new Packages.com.novell.ldap.LDAPConnection();
```

and save ourselves some typing every time we need to get to a JLDAP object.

Once an LDAPConnection object has been instantiated, we can go ahead and bind to the tree:

```
lc.connect( ldapHost, ldapPort );
lc.bind(ldapVersion, loginDN, password );
```

These are standard JLDAP calls, documented in the Novell NDK Javadocs.

Next, we set up our file-I/O objects:

```
   var fos = new java.io.FileOutputStream(fileName);
var writer = new jldap.util.LDIFWriter(fos);
```

The file will be written to whatever path was supplied in the `fileName` argument.

Querying the server requires one line of code:

```
// send query
var results = lc.search( searchBase,
                         scope,
                         searchFilter,
                         javaStringArray,
                         typesOnly);
```

The search results can be enumerated and written out very simply:

```
while ( results.hasMoreElements() )
 writer.writeEntry( results.nextElement() );
```

Finally, we close all files, streams, and connections:

```
writer.finish();
fos.close();
lc.disconnect();
```

Testing the Script

To test the script, we can put the following text in a Function Action and run it:

```
query2Ldif( 'www.nldap.com'
            'anonymous',
            '', /* no password */
            '', /* no search base DN */
            0, /* 0 for base-object scope */
            '(objectClass=*)', /* filter == all objects */
          new Array('*'), /* attrib array */
            'c:\\temp\\test.ldif' )
```

When the Function Action with the script runs, it queries the Novell public server for its DSE root information. An LDIF file is written to disk.

The LDIF file can be inspected with a simple text editor. It contains (in part):

```
# This LDIF file was generated by the LDIF APIs of Novell's Java LDAP SDK
version: 1

dn:
errors: 21428
subschemaSubentry: cn=schema
directoryTreeName: DEVNET-TREE
securityErrors: 972
compareOps: 218
bindSecurityErrors: 850
outBytes: 2279628812
vendorVersion: eDirectory v8.7.0 (10410.57)
simpleAuthBinds: 383333
supportedFeatures: 1.3.6.1.4.1.4203.1.5.1
supportedFeatures: 2.16.840.1.113719.1.27.99.1
repUpdatesIn: 0
abandonOps: 572
supportedSASLMechanisms: EXTERNAL
supportedSASLMechanisms: DIGEST-MD5
supportedSASLMechanisms: NMAS_LOGIN
referralsReturned: 0
removeEntryOps: 6683
searchOps: 654935
addEntryOps: 20253
strongAuthBinds: 32
modifyEntryOps: 888
vendorName: Novell, Inc.
listOps: 0
modifyRDNOps: 9
chainings: 300
...
[ etc. ]
```

A LDAP Glossary

Alias A directory entry that names another directory entry.

Anonymous Bind A connection to a directory server established without a password (and usually also without a user ID). The rights granted under an anonymous bind are usually restrictive.

Asynchronous Request Any request made without expectation of an immediate response. Usually, a client that makes an asynchronous request will begin other processing immediately, without waiting for a response from the server. This is in contrast to a synchronous request, in which the client issues a request and then blocks until a response has been received from the host.

Authentication The process of verifying the identity of a participant in a conversation. ("Is this person who he says he is?")

Base DN The partially qualified name (or container context) specifying the "starting point" for a search or for access to a directory.

Bind To obtain access to a directory based on a set of credentials. (When access is granted based on empty credentials, it is said to be an *anonymous* bind.)

CA Certification Authority. An entity that issues digital certificates and/or can vouch for the authenticity of a certificate.

cn Common name.

dn see *Distinguished Name*

Entry In a generic sense, an entry in a directory is analogous to a record or row in a database. The node holding the name `Robert` can also hold information about the person, such as his manager's name, e-mail, instant messaging name, and so forth. The whole node is an *entry.*

Attribute An attribute is associated with a value. For example, a `cn` (common name) attribute might be associated with a value of `Robert`. Objects in a tree are collections of attributes and their associated values.

Chaining A name-resolution facility whereby the server, acting as a proxy for the client, locates non-local DIT entries by following referrals. This type of referral-following is not under the control of the client.

Container A directory object that can contain other objects.

DAP Directory Access Protocol (X.519)

Directory Information Tree (DIT) The entire information tree of the directory itself is called the DIT (Directory Information Tree).

Distinguished Name (DN) A *distinguished name* is a fully qualified name that uniquely identifies an entity in a directory. For example, a user of a website might be entered into a directory with a unique DN of `cn=Theo87,ou=Visitors,o=Blogsville`. There can be only one entry with that particular DN. (Notice that the order and reading direction of the DN are critical. The DN is parsed left-to-right with the "leaf" or terminus portion—in this case, `cn=Theo87`—coming first.)

DSA Directory Server Agent—the X.500 term for a directory server or (L)DAP host.

DSE DSA-specific-entry—a root-level entry in a directory, describing server capabilities.

DSML Directory Services Markup Language—an XML grammar for encoding directory information and requests.

JLDAP Java LDAP library—an open-source LDAP SDK developed by Novell.

LDIF LDAP Data Interchange Format

Object A collection of attributes and values—an instance of an object class. (See *Object Class*, below.)

Object Class The formal definition of an object (as contained in the directory schema), including the number and types of required and optional attributes, the OID, the object type (abstract, structural, or auxiliary), and the object class name.

OID (Object Identifier) A string, in dotted-decimal form, that identifies an object type.

Referral A name-resolution hint. A server can send a referral to a client to help the client locate information that is not local to the current host. It is up to the client whether to follow the referral or not.

Relative Distinguished Name (RDN) RDN (Relative Distinguished Name) is a *portion* of an entity's fully qualified DN, containing (or equal to) the terminal or "leaf-node" identifier for the entity, such as `cn=Rich`.

RFC Request for Comment. A mechanism by which the Internet Engineering Task Force (IETF) publishes web-protocol specifications.

Schema The schema of an LDAP directory gives the layout of the information it contains and specifies how the information is grouped. It therefore allows clients or external interfaces to discover structural features of the directory and how the tree can be accessed in terms of search, addition, deletion, modification, and so on. Refer to RFC 2256 for information on the LDAP object classes and attributes.

Scope The bounds within which an operation is valid. For an LDAP search request, scope can be one of base, first child level, or subtree. If a search is scoped to base level, only entries within the base-DN container will be searched. If the search is scoped to first child level, the container and its immediate children will be searched. "Subtree" scope means the container, its child objects, and all children-of-children, etc. (down to terminal entries) will be searched.

Subordinate Entry An object or entry that is contained by a "container object."

TLS Transport Layer Security—a non-proprietary industry standard for implementing encrypted, authenticated communications over network connections. It can accommodate, but is not limited to, conventional SSL (Secure Socket Layer) methodologies.

X.500 A document, published by the International Telecommunications Union, that describes the fundamental concepts underlying the notion of a directory. Often, X.500 is used as a synonym for "the non-lightweight directory protocos" (otherwise known as DAP), but in fact the DAP protocol is specified in X.519, and the complete ITU directory "specification" is distributed across a dozen or so X.500-series publications.

B LDAP Result Codes

Value	Result Code
0	SUCCESS (success)
1	OPERATIONS_ERROR (operationsError)
2	PROTOCOL_ERROR (protocolError)
3	TIME_LIMIT_EXCEEDED (timeLimitExceeded)
4	SIZE_LIMIT_EXCEEDED (sizeLimitExceeded)
5	COMPARE_FALSE (compareFalse)
6	COMPARE_TRUE (compareTrue)
7	AUTH_METHOD_NOT_SUPPORTED (authMethodNotSupported)
8	STRONG_AUTH_REQUIRED (strongAuthRequired)
10	REFERRAL (referral)
11	ADMIN_LIMIT_EXCEEDED (adminLimitExceeded)
12	UNAVAILABLE_CRITICAL_EXTENSION (unavailableCriticalExtension)
13	CONFIDENTIALITY_REQUIRED (confidentialityRequired)
14	SASL_BIND_IN_PROGRESS (saslBindInProgress)
16	NO_SUCH_ATTRIBUTE (noSuchAttribute)
17	UNDEFINED_ATTRIBUTE_TYPE (undefinedAttributeType)
18	INAPPROPRIATE_MATCHING (inappropriateMatching)
19	CONSTRAINT_VIOLATION (constraintViolation)
20	ATTRIBUTE_OR_VALUE_EXISTS (AttributeOrValueExists)
21	INVALID_ATTRIBUTE_SYNTAX (invalidAttributeSyntax)
32	NO_SUCH_OBJECT (noSuchObject)
33	ALIAS_PROBLEM (aliasProblem)
34	INVALID_DN_SYNTAX (invalidDNSyntax)
35	IS_LEAF (isLeaf)
36	ALIAS_DEREFERENCING_PROBLEM (aliasDereferencingProblem)
48	INAPPROPRIATE_AUTHENTICATION (inappropriateAuthentication)
49	INVALID_CREDENTIALS (invalidCredentials)

Value	Result Code
50	INSUFFICIENT_ACCESS_RIGHTS (insufficientAccessRights)
51	BUSY (busy)
52	UNAVAILABLE (unavailable)
53	UNWILLING_TO_PERFORM (unwillingToPerform)
54	LOOP_DETECT (loopDetect)
64	NAMING_VIOLATION (namingViolation)
65	OBJECT_CLASS_VIOLATION (objectClassViolation)
66	NOT_ALLOWED_ON_NONLEAF (notAllowedOnNonLeaf)
67	NOT_ALLOWED_ON_RDN (notAllowedOnRDN)
68	ENTRY_ALREADY_EXISTS (entryAlreadyExists)
69	OBJECT_CLASS_MODS_PROHIBITED (objectClassModsProhibited)
71	AFFECTS_MULTIPLE_DSAS (affectsMultipleDSAs
80	OTHER (other)

Local errors, resulting from actions other than an operation on a server.

Value	Result Code
81	SERVER_DOWN
82	LOCAL_ERROR
83	ENCODING_ERROR
84	DECODING_ERROR
85	LDAP_TIMEOUT
86	AUTH_UNKNOWN
87	FILTER_ERROR
88	USER_CANCELLED
90	NO_MEMORY
91	CONNECT_ERROR
92	LDAP_NOT_SUPPORTED
93	CONTROL_NOT_FOUND
94	NO_RESULTS_RETURNED
95	MORE_RESULTS_TO_RETURN
96	CLIENT_LOOP
97	REFERRAL_LIMIT_EXCEEDED
100	INVALID_RESPONSE
101	AMBIGUOUS_RESPONSE
112	TLS_NOT_SUPPORTED

Index

T

U

V

W

X