Identity Manager® Driver Developer Kit™

Table of Contents

Table of Contents 2
NDK: NetIQ Identity Manager Driver Kit 3
1.0 Identity Manager and Identity Manager Drivers 4
1.1 Driver Basics 4
1.2 Requirements and Resources 8
1.2.1 Requirements 8
1.2.2 Resources 9
1.3 Identity Manager Architecture 9
1.3.1 Identity Manager Features 12
1.3.2 Identity Manager Engine and Driver Interaction 13
1.3.3 Driver Interaction with the Identity Vault Objects and Attributes 16
1.4 Identity Manager and Multiple Directories 17
1.5 Designing the Driver 18
1.6 Where to Get Started 19
2.0 Writing an Identity Manager Driver 19
2.1 Driver Overview 19
2.1.1 Driver Communication and Threads 20
2.1.2 Driver Life Cycle 20
2.2 Getting Started 22
2.2.1 Application Requirements 22
2.2.2 XML Interface 22
2.2.3 Language—C++ or Java 23
2.2.4 Overview of the Process 24
2.3 Starting with the Skeleton Driver 25
2.3.1 Setting Up a Skeleton Driver Instance to Run 25
2.3.2 Compiling the Java Skeleton Driver 26
2.3.3 Compiling the C++ Skeleton Driver 27
2.4 Constructing the Driver Object 27
2.4.1 Java Constructor 27
2.4.2 CreateDriver Function for C++ 27
2.5 Implementing the DriverShim Interface 28
DriverShim init 28
Syntax 29
Java 29
C++ 29
Parameters 29

Return Values 29
Remarks 29
Sample Code 31
Java Sample Code 31
C++ Sample Code 32
DriverShim getSubscriptionShim 34
Syntax 34
Java..34
C++ 34
Remarks 34
Sample Code 35
Java Sample Code 35
C++ Sample Code 35
Driver getPublicationShim 35
Syntax 35
Java..35
C++ 35
Remarks 36
Sample Code 36
Java Sample Code 36
C++ Sample Code 36
DriverShim shutdown 36
Syntax 36
Java..36
C++ 37
Parameters 37
Return Values 37
Remarks 37
Sample Code 38
Java Sample Code 38
C++ Sample Code 38
DriverShim getSchema 39
Syntax 39
Java..39
C++ 39
Parameters 39
Remarks 40
Sample Code 41
Java Sampe Code 41

C++ Sampe Code 42
DriverShim destroy (C++ only) 42
Syntax 43
C++ 43
Remarks 43
C++ Sample Code 43
2.6 Implementing the SubscriptionShim Interface 43
SubscriptionShim init 44
Syntax 44
Java 44
C++ 44
Parameters 44
Return Values 45
Remarks 45
Sample Code 46
Java Sample Code 46
C++ Sample Code 47
SubscriptionShim execute 48
Syntax 48
Java 48
C++ 48
Parameters 49
Remarks 49
Add Command 49
Modify Command 50
Query Command 51
Sample Code 53
Java Sample Code 53
C++ Sample Code 55
2.7 Implementing the PublicationShim Interface 57
PublicationShim init 57
Syntax 57
Java 57
C++ 57
Parameters 58
Return Values 58
Remarks 58
Sample Code 59
Java Sample Code 59

C++ Sample Code ... 60
PublicationShim start ... 62
Syntax ... 62
Java ... 62
C++ ... 62
Parameters ... 62
Remarks ... 62
Add Event ... 63
Modify Event ... 64
Delete Event ... 65
Sample Code ... 65
Java Sample Code ... 65
C++ Sample Code ... 66
2.8 Implementing the XmlQueryProcessor Interface ... 68
query ... 68
Syntax ... 68
C++ ... 68
Java ... 69
Parameters ... 69
Remarks ... 69
Sample Code ... 69
Java Example Code ... 69
C++ Example Code ... 69
2.9 Dealing with XML Documents ... 70
2.9.1 Java Sample Code ... 70
2.9.2 C++ Sample Code ... 72
2.10 Driver State ... 75
2.11 Driver Configuration ... 75
2.12 Additional Tips for C++ Drivers ... 78
2.12.1 Memory Management ... 78
2.12.2 C++ Utility Functions and Interfaces ... 79
3.0 Debugging the Driver ... 81
3.1 Using DSTrace and the Identity Manager Trace Log ... 81
3.1.1 Enabling Verbose Identity Manager Driver Messages ... 82
3.1.2 Enabling the Identity Manager Trace Log ... 82
3.1.3 Adding Trace Messages to Your Driver ... 83
3.2 Using a Debugger with a C++ Driver ... 84
3.2.1 DLLs on Windows (NT, 2000, XP) ... 84
3.3 Using a Debugger with a Java Driver ... 84

3.3.1 Agent Debugger 85
3.3.2 Java Platform Debugger Architecture (JPDA) 85
3.3.3 Visual Cafe 3.0 Debugger 85
3.3.4 JDB Debugger 86
3.3.5 JVM Variables 86
4.0 Introduction to the Rules and Filters 88
4.1 Event Filters 88
4.2 Transformation Rules 88
4.3 Channel-Independent Transformations 90
4.3.1 Schema Mapping Rules 90
4.3.2 Input Transformation Style Sheet 90
4.3.3 Output Transformation Style Sheet 91
4.4 Channel-Dependent Transformations 91
4.4.1 Matching Rules 92
4.4.2 Create Rules 92
4.4.3 Placement Rules 93
4.4.4 Event Transformation Rules 93
4.4.5 Command Transformation Rules 93
4.5 Event Processing 94
4.5.1 Subscriber Channel 94
4.5.2 Publisher Channel 95
6.0 Driver Installation 96
6.1 Copy the Driver 96
6.2 Create the Driver Objects 96
6.3 Exporting the Configuration 97
6.4 Set Up the Server Environment 97
7.0 DTD Commands and Events 97
7.1 Top Level Elements 98
<nds> 98
Description 98
Definition 98
Attributes of <nds> 99
Elements 99
Parent 99
Example 99
<driver-config> 100
Description 100
Definition 100
Attributes 101

Elements 101
Parent 101
Example 101
7.2 Input and Output Elements 102
<input> 102
Description 102
Definition 103
Elements 103
Parent 104
<output> 104
Description 104
Definition 104
Elements 105
Parent 105
7.3 Command and Event Elements 105
<add> 106
Description 106
Definition 106
Attributes 107
Elements 107
Request Format 108
Command 108
Event 108
Reply Format 108
Command 108
Event 109
Parent 109
Example 109
<add-association> 109
Description 110
Definition 110
Attributes 110
Request Format 110
Event 110
Reply Format 111
Parent 111
Example 111
<delete> 111
Description 111

Definition ... 111
Attributes ... 112
Elements ... 112
Request Format ... 112
Command ... 112
Event ... 112
Reply Format ... 113
Parent ... 113
Example ... 113
<init-params> ... 113
Description ... 113
Definition ... 113
Elements ... 114
Request Format ... 114
Command ... 114
Reply Format ... 115
Parent ... 115
Example ... 115
<instance> ... 119
Description ... 119
Definition ... 119
Attributes ... 119
Elements ... 120
Reply Format ... 120
Parent ... 120
Example ... 120
<modify> ... 121
Description ... 121
Definition ... 121
Attributes ... 121
Elements ... 122
Request Format ... 122
Command ... 122
Event ... 123
Reply Format ... 123
Parent ... 123
Example ... 123
<modify-association> ... 124
Description ... 124

Definition 124
Attributes 124
Elements 124
Request Format 125
Event 125
Reply Format 125
Parent 125
Example 125
<modify-password> 125
Description 125
Definition 126
Attributes 126
Elements 127
Parent 127
<move> 127
Description 127
Definition 127
Attributes 128
Elements 128
Request Format 128
Command 129
Event 129
Reply Format 129
Parent 129
Example 129
<query> 130
Description 130
Definition 130
Attributes 131
Elements 131
Request Format 132
Reply Format 132
Parent 132
Remarks 133
Example 133
<query-schema> 134
Description 134
Definition 134
Attributes 134

Request Format 134
Reply Format........ 135
Parent 135
Remarks 135
Example 135
<remove-association>........ 135
Description........ 135
Definition 135
Attributes........ 136
Request Format 136
Event 136
Reply Format........ 136
Parent 136
Example 136
<rename> 136
Description........ 136
Definition 137
Attributes........ 137
Elements........ 138
Request Format 138
Command........ 138
Event 138
Reply Format........ 139
Parent 139
Example 139
<schema-def>........ 139
Description........ 139
Definition 139
Attributes........ 140
Elements........ 140
Required Elements 140
Parent 141
Remarks 141
Example 141
<status>........ 143
Description........ 143
Definition 143
Attributes........ 143
Parent 143

Example 144
7.4 Other Elements 144
<add-attr> 144
Definition 144
Attributes 144
Elements 144
Parent 145
<add-value> 145
Definition 145
Elements 145
Parent 145
<allow-attr> 145
Definition 145
Attributes 145
Parent 146
<allow-class> 146
Definition 146
Attributes 146
Parent 146
<association> 146
Description 146
Definition 147
Attributes 147
Parent 147
<attr> 147
Definition 147
Attributes 147
Elements 148
Parent 148
<attr-def> 148
Definition 148
Attributes 148
Parent 149
Remarks 149
<authentication-info> 149
Definition 149
Elements 150
Parent 150
Remarks 150

<class-def> 150
Definition 150
Attributes 150
Elements 151
Parent 151
<component> 151
Definition 151
Attributes 151
Parent 152
<config-object> 152
Description 152
Definition 152
Attributes 152
Elements 152
Parent 152
<driver-filter> 152
Description 153
Definition 153
Attributes 153
Elements 154
Format 154
Parent 154
Sample Filter 154
<driver-options> 155
Definition 155
Elements 155
Parent 155
Remarks 155
Sample Option Tags 155
<driver-state> 156
Definition 156
Parent 156
Remarks 156
Sample State Tags 156
<modify-attr> 157
Description 157
Definition 157
Attributes 157
Elements 157

Parent 158
<old-password> 158
Definition 158
Elements 158
Parent 158
<parent> 158
Description 158
Definition 159
Attributes 159
Elements 159
Parent 159
Remarks 159
<publisher-options> 160
Definition 160
Elements 160
Parent 160
Remarks 160
Sample Option Tags 160
<publisher-state> 161
Definition 161
Parent 161
Remarks 161
Sample State Tags 161
<read-attr> 162
Definition 162
Attributes 162
Parent 162
<remove-value> 162
Definition 162
Elements 163
Parent 163
<search-attr> 163
Definition 163
Attributes 163
Elements 163
Parent 163
<search-class> 164
Definition 164
Attributes 164

Parent ... 164
<source> ... 164
Definition ... 164
Elements ... 165
Attributes of <product> ... 165
Parent ... 165
<subscriber-options> ... 165
Definition ... 165
Elements ... 165
Parent ... 165
Remarks ... 166
Sample Option Tags ... 166
<subscriber-state> ... 166
Definition ... 166
Parent ... 167
Remarks ... 167
Sample State Tags ... 167
<value> ... 167
Definition ... 167
Attributes ... 168
Elements ... 168
Parent ... 168
Remarks ... 168
8.0 Rule Reference ... 171
8.1 Schema Mapping Elements ... 171
<attr-name-map> ... 172
Description ... 172
Definition ... 172
Attributes ... 173
Elements ... 173
Parent ... 173
Examples ... 173
8.2 Matching Rule Elements ... 174
<matching-rules> ... 175
Description ... 175
Definition ... 176
Elements ... 176
Parent ... 177
Examples ... 177

<matching-rule> 178
Description 178
Definition 178
Attributes 178
Elements 178
Parent 179
<match-attr> 179
Description 179
Definition 179
Attributes 179
Elements 179
Parent 180
<match-class> 180
Definition 180
Attributes 180
Parent 180
<match-path> 180
Description 180
Definition 181
Attributes 181
Parent 181
8.3 Create Rule Elements 181
<create-rules> 182
Description 182
Definition 182
Elements 183
Parent 183
Sample Create Rules 183
<create-rule> 184
Description 184
Definition 185
Attributes 185
Elements 185
Parent 185
<match-attr> 186
Description 186
Definition 186
Attributes 186
Elements 186

Parent 186
<required-attr> 186
Description 187
Definition 187
Attributes 187
Elements 187
Parent 187
<template> 187
Description 187
Definition 188
Attributes 188
Parent 188
8.4 Placement Rule Elements 188
<placement-rules> 189
Description 189
Definition 189
Attributes 190
Elements 191
Parent 191
Sample Placement Rules 191
<placement-rule> 192
Description 192
Definition 193
Attributes 193
Elements 193
Parent 193
<match-attr> 193
Description 194
Definition 194
Attributes 194
Elements 194
Parent 194
<match-class> 194
Definition 194
Attributes 195
Parent 195
<match-path> 195
Description 195
Definition 195

Attributes 195
Parent 196
<placement> 196
Description 196
Definition 196
Attributes for <copy-attr> 196
Elements 197
Parent 197
8.5 Event Transformation Rules 197
8.5.1 Sample Event Transformation Rule 198
8.6 Command Transformation Rules 199
8.6.1 Sample Command Transformation Rules 199
8.7 Input Transformation Style Sheets 200
8.8 Output Transformation Style Sheets 201
9.0 Style Sheets 201
9.1 Restrictions 202
9.1.1 Matching Rule Restrictions 202
9.1.2 Create Rule Restrictions 202
9.1.3 Placement Rule Restrictions 203
9.2 Starting with an Identity Transformation 203
9.3 Using the Parameters that Identity Manager Passes 203
9.4 Using Extension Functions 206
9.5 Testing Style Sheets Outside of Identity Manager 207
9.6 Invoking the Novell XSLT Processor Directly 208
9.7 Creating a Password Example: Create Rule 209
9.8 Creating an Identity Vault User Example: Create Rule 210
10.0 Identity Manager Error Codes 216
11.0 Javadoc, FAQs, and DTD Reference 217
A.0 VRTest Application 217
A.1 Requirements and Installation 217
B.0 Identity Manager Definitions for the Schema 218
B.1 Identity Manager Object Class Definitions 218
Identity Manager-Driver 218
ASN.1 ID 218
Class Flags 218
Class Structure 219
Mandatory Attributes 219
Optional Attributes 219
Default ACL Template 220

Remarks 220
Identity Manager-DriverSet 220
ASN.1 ID 220
Class Flags 220
Class Structure 220
Mandatory Attributes 221
Optional Attributes 221
Default ACL Template 221
Remarks 222
DirXML-Publisher 222
ASN.1 ID 222
Class Flags 222
Class Structure 222
Mandatory Attributes 222
Optional Attributes 223
Default ACL Template 223
Remarks 223
DirXML-Rule 223
ASN.1 ID 223
Class Flags 224
Class Structure 224
Mandatory Attributes 224
Optional Attributes 224
Default ACL Template 225
Remarks 225
DirXML-StyleSheet 225
ASN.1 ID 225
Class Flags 225
Class Structure 226
Mandatory Attributes 226
Optional Attributes 226
Default ACL Template 226
Remarks 227
DirXML-Subscriber 227
ASN.1 ID 227
Class Flags 227
Class Structure 227
Mandatory Attributes 227
Optional Attributes 228

Default ACL Template ... 228
Remarks ... 228
StyleSheet ... 228
ASN.1 ID ... 228
Class Flags ... 229
Class Structure ... 229
Mandatory Attributes ... 229
Optional Attributes ... 229
Default ACL Template ... 230
B.2 DirXML Attribute Definitions ... 230
DirXML-ApplicationSchema ... 230
Syntax ... 230
Constraints ... 230
ASN.1 ID ... 230
Used In ... 231
Remarks ... 231
DirXML-Associations ... 231
Syntax ... 231
Constraints ... 231
ASN.1 ID ... 231
Used In ... 231
Remarks ... 231
DirXML-CreateRule ... 232
Syntax ... 232
Constraints ... 232
ASN.1 ID ... 232
Used In ... 232
Remarks ... 232
DirXML-DriverCacheLimit ... 232
Syntax ... 232
Constraints ... 233
ASN.1 ID ... 233
Used In ... 233
Remarks ... 233
DirXML-DriverFilter ... 233
Syntax ... 233
Constraints ... 233
ASN.1 ID ... 233
Used In ... 234

DirXML-DriverSetDN 234
- Syntax 234
- Constraints 234
- ASN.1 ID 234

DirXML-DriverStartOption 234
- Syntax 234
- Constraints 234
- ASN.1 ID 235
- Used In 235
- Remarks 235

DirXML-DriverStorage 235
- Syntax 235
- Constraints 235
- ASN.1 ID 235
- Used In 236
- Remarks 236

DirXML-DriverTraceLevel 236
- Syntax 236
- Constraints 236
- ASN.1 ID 236
- Used In 236
- Remarks 236

DirXML-EventTransformationRule 237
- Syntax 237
- Constraints 237
- ASN.1 ID 237
- Used In 237
- Remarks 237

DirXML-InputTransform 237
- Syntax 237
- Constraints 238
- ASN.1 ID 238
- Used In 238
- Remarks 238

DirXML-JavaDebugPort 238
- Syntax 238
- Constraints 238
- ASN.1 ID 238
- Used In 239

Remarks 239
DirXML-JavaModule 239
Syntax 239
Constraints 239
ASN.1 ID 239
Used In 239
Remarks 239
DirXML-JavaTraceFile 240
Syntax 240
Constraints 240
ASN.1 ID 240
Used In 240
Remarks 240
DirXML-MappingRule 240
Syntax 240
Constraints 240
ASN.1 ID 241
Used In 241
Remarks 241
DirXML-MatchingRule 241
Syntax 241
Constraints 241
ASN.1 ID 241
Used In 241
Remarks 242
DirXML-NativeModule 242
Syntax 242
Constraints 242
ASN.1 ID 242
Used In 242
Remarks 242
DirXML-OutputTransform 242
Syntax 243
Constraints 243
ASN.1 ID 243
Used In 243
Remarks 243
DirXML-PlacementRule 243
Syntax 243

Constraints 243
ASN.1 ID 244
Used In 244
Remarks 244
DirXML-ShimAuthID 244
Syntax 244
Constraints 244
ASN.1 ID 244
Used In 244
Remarks 245
DirXML-ShimAuthPassword 245
Syntax 245
Constraints 245
ASN.1 ID 245
Used In 245
DirXML-ShimAuthServer 245
Syntax 245
Constraints 245
ASN.1 ID 246
Used In 246
Remarks 246
DirXML-ShimConfigInfo 246
Syntax 246
Constraints 246
ASN.1 ID 246
Used In 246
Remarks 247
DirXML-ServerList 247
Syntax 247
Constraints 247
ASN.1 ID 247
Used In 247
DirXML-State 247
Syntax 247
Constraints 247
ASN.1 ID 248
Used In 248
Remarks 248
DirXML-Timestamp 248

Syntax 248
Constraints 249
ASN.1 ID 249
Used In 249
DirXML-XSLTraceLevel 249
Syntax 249
Constraints 249
ASN.1 ID 249
Used In 249
Remarks 250
XmlData 250
Syntax 250
Constraints 250
ASN.1 ID 250
Used In 250
Remarks 250
C.0 Revision History 251
D.0 Legal Notices 252
Novell Trademarks 252
Third-Party Materials 252

NDK: NetIQ Identity Manager Driver Kit

Novell Identity Manager, powered by Identity Manager engine, enables data synchronization between the Identity Vault and the connected application. Identity Manager has four main components:

- The Identity Manager engine which provides the framework.
- The Identity Manager policies which control the mapping of attributes and classes and the matching and creation of entries.
- Event filters which control the direction of data synchronization.
- The Identity Manager driver shims which serves as the interface between the application and the Identity Manager engine.

This document describes how to implement an Identity Manager Driver shim. It contains the following sections:

- Identity Manager and Identity Manager Drivers
- Writing an Identity Manager Driver
- Debugging the Driver
- Introduction to the Rules and Filters
- Novell exteNd Composer Driver
- Driver Installation
- DTD Commands and Events
- Rule Reference
- Style Sheets
- Identity Manager Error Codes
- Javadoc, FAQs, and DTD Reference
- VRTest Application
- Identity Manager Definitions for the Schema
- Revision History

Audience

This guide is intended for the XML developers interested in creating Identity Manager driver shims.

Feedback

We want to hear your comments and suggestions about this manual and the other documentation included with this product. Please use the User Comments feature at the bottom of each page of the online documentation.

Documentation Updates

For the most recent version of this document, see the Novell Identity Manager Driver NDK page.

Additional Information

For current Identity Manager documentation, see the Identity Manager Documentation Web site.

For current documentation on Novell Identity Manager Policies and Driver customization, see the Identity Manager Drivers Documentation Web site.

Documentation Conventions

In this documentation, a greater-than symbol (>) is used to separate actions within a step and items within a cross-reference path.

1.0 Identity Manager and Identity Manager Drivers

The Identity Manager engine provides the framework to connect disparate directories to the Identity Vault and to synchronize selected information. The Identity Manager driver is the application-specific piece that needs to be written for each application that you want to share and synchronize data with Identity Vault. Identity Manager allows an application to do the following:

- Share data with Identity Vault
- Synchronize data to Identity Vault when modified in the application
- Synchronize data to the application when modified in the Identity Vault

Identity Manager does not provide single-seat administration or solve the problem of multiple administrators, with an administrator for each application. It does solve the problem of having duplicate, but inconsistent data in each of the applications.

Each application requires an Identity Manager driver to interface with the application and the Identity Manager engine. This document explains how to develop and implement such a driver. This chapter provides an overview of the following:

1.1 Driver Basics

Identity Manager is designed to synchronize information between data sources. The core piece of Identity Manager, the Identity Manager engine, handles all synchronization to and from the Identity Vault, which acts as the hub of the synchronization.

For each application that wishes to synchronize with Identity Vault, a driver must be written to do two basic things: Notify the Identity Manager engine of a change in the application, and receive notifications of changes in the Identity Vault from the Identity Manager engine and make those changes in the application.

In order to communicate with any number of applications, an XML format, called XDS, is used as the common denominator. XDS is eDirectory-flavored XML with elements to represent directory objects and the operations you might perform on them.

All information exchanged between your driver and the Identity Manager engine is in this format. XDS documents contain either a command or event, depending on the flow of information.

Commands are XDS documents received from the Identity Manager engine, containing changes from the Identity Vault. Events are XDS documents sent to the Identity Manager engine, containing changes in your application. This distinction is simple but important, because different actions are required depending whether you are handling a command or an event.

From the standpoint of driver development, the goal is straightforward: for each event that occurs in your application, pass the Identity Manager engine an XDS document that explains the event. For each command received from the Identity Manager engine as an XDS document, translate the document into API calls to apply the change.

Publisher and Subscriber

As previously mentioned, your driver must be able to publish changes to the Identity Manager engine, and receive changes from the Identity Manager engine. Identity Manager defines these actions as publishing and subscribing, named from the point of view of the application. Your application submits changes on the Publisher Channel and receives changes on the Subscriber channel.

Your driver must implement a separate process for each channel.

A Typical Transaction on the Subscriber Channel

For example, a user is added in the Identity Vault. This change could have occurred directly in the Identity Vault or in a human resources system that is synchronized to the Identity Vault using a PeopleSoft or some other Identity Manager driver. When this change occurs, the Identity Manager engine sends an XDS document to your driver on the subscriber channel containing information about the change. It might look similar to the following:

```
<nds dtdversion="2.0" ndsversion="8.7.3">
<source>
   <product version="2.0">DirXML</product>
   <contact>Novell, Inc.</contact>
</source>

<input>
   <add class-name="User" event-id="0" src-dn="\ACME\Sales\Smith"
   src-entry-id="33071">
      <add-attr attr-name="Surname">
         <value timestamp="1040071990#3" type="string">Smith</value>
      </add-attr>
      <add-attr attr-name="Telephone Number">
         <value timestamp="1040072034#1" type="teleNumber">111-1111
               </value>
      </add-attr>
   </add>
</input>
</nds>
```

Your driver now has an XML document containing a change that occurred in the Identity Vault.

Your driver shim parses this xml document to determine what actions need to occur to bring the application up to date, and then translates this into API calls.

For an LDAP application, the API calls might look similar to the following:

```
LDAPAttributeSet attributeSet = new LDAPAttributeSet();
String containerName = "ou=Sales,o=Acme";

attributeSet.add( new LDAPAttribute("objectclass", new String("User")));
attributeSet.add( new LDAPAttribute("sn", new String("Smith")));
attributeSet.add( new LDAPAttribute("telephonenumber",
```

```
                                                new String("111-1111")));
String  dn  = "cn=Smith," + containerName;       LDAPEntry newEntry = new
LDAPEntry( dn, attributeSet );
```

It is important to be aware that decisions about where to place new objects, how to create new objects, filling in missing mandatory attributes, and so on, are made in the Identity Manager engine, not in your driver. Your driver simply needs to report each event that occurs to the Identity Manager engine, and make any changes requested by the Identity Manager engine.

A Typical Transaction on the Publisher Channel

A Change to a user's manager occurs in your application, which for this example, is an HR database designed to handle your organization hierarchy. When this change occurs, your driver needs to first find out about the change, then translate the change into an XDS document to send it to the Identity Manager engine.

Traditionally, the most difficult aspect of authoring an Identity Manager driver is finding a reliable, consistent source to monitor changes in your application. Some applications, such as eDirectory, have full-featured event notification systems that can simplify this process. Some applications log all changes to a file or other database (which unfortunately can change from release to release), and others have no method of logging changes, which require driver writers to implement their own system. Most vendors consider data sharing a positive feature and are willing to help you determine a suitable method to monitor changes.

For this example, let's say your application has a change monitoring API or persistent search functionality, which asynchronously sends you the events you wish to monitor through a getResponse method when they occur:

```
//connect, authenticate, and enter the event monitor or persistent search API


while (( event = queue.getResponse()) != null )


//set up conditional statements to determine the type of change


if (event instanceof SomeTypeOfEvent)
{
  //transform this event into an XDS document with relevant information
  //and send it to the DirXML engine, or call another API to...
}
if (event instanceof SomeOtherTypeOfEvent)
{
  //transform this event into an XDS document with relevant information
  //and send it to the DirXML engine, or call another API to...
}


...
```

The implementation of finding and reporting changes is left to your creativity, the only requirement is that in the end, an XDS document is sent to the Identity Manager engine containing the change.

Policies, Transformations, Stylesheets, XSLT, and so on

If you have any exposure to Identity Manager, you have likely heard much talk about policies, stylesheets, transformations, and many other things that can cloud your idea of what exactly your driver must do.

Simply put, policies modify an event sent to the Identity Manager engine to make it work for an individual environment. For example, one organization might use the inetorgperson as the main user class, while another organization might use User. If you are synchronizing a

phone number, you don't really care what object class is used, but you can't write code to handle every situation. Therefore, a policy can be implemented to add the phone number change to an inetorgperson for the first organization, and a separate rule can be implemented to make it work for the User class.

Policies make schema transformations; specify matching criteria to determine if an object already exists in an application or the Identity Vault, and many other things. Because of this, an add event reported by your application may end out as a modify operation in the Identity Vault, if a matching policy determines that the object you added already exists in the data store.

Most of the time, your driver does not need to make many decisions about the events it reports. If you think that somebody might want to synchronize an event using your driver, you should probably report it. There are exceptions, however, in the case that your driver has to transfer a lot of data over the wire, you might want to limit the events you report, and you will probably think of others as you learn your application. This decision of which events to report is left entirely up to you, just be aware that it is often easier from an implementation standpoint to filter events in the engine than it is to re-write or re-compile your driver to handle a new situation.

On the subscription channel, it is basically the same in reverse. If an event is sent to your application, it should be logged. For example, a new user is created in the Identity Vault. Before sending this command to your driver, the Identity Manager engine calls a series of policies, one of which defines the way objects are created, in which rules can determine if a corresponding user already exists in your application, make decisions about placement, and provide default values for required attributes that are not specified, and so on. This add event may be transformed into a modify event if the object exists in your application, and attributes that were not contained in the original event could be added to conform with the object creation model of your application.

Once these rules are applied and a command is sent to your driver, your driver should make the change in the application. If the change is somehow incorrect, then logic needs to be added to the creation policy, not to your driver.

Once again, the goal is straightforward: for each event that occurs in your application, pass the Identity Manager engine an XDS document that explains the event. For each command received from the Identity Manager engine as an XDS document, translate the command into API calls and execute the command.

To learn more about policies, see the Policy Builder and Driver Customization Guide.

Working with XDS documents

From the introduction, you are probably aware that a good portion of your development time is spent parsing and creating XDS documents. To work with XDS documents, there are four APIs available:

- Document Object Model (DOM)
- Simple API for XML (SAX)
- Novell XDS Libraries
- Serial

The two most common XML parsing interfaces in use today are DOM and SAX.

DOM builds an XML document into a tree structure, you navigate this tree to find information. SAX is an event-driven approach that reports events using callbacks.

DOM and SAX are both open interfaces that can handle any sort of XML document. With this flexibility there is increased development overhead, because every type of XML document is not handled in a similar fashion.

To reduce development time, Novell has developed a custom XML parsing API, called XDS Libraries, which extends the DOM interface. The XDS Libraries are designed to work specifically with Identity Manager and the Identity Vault, enabling them to relieve a fairly large portion of the development overhead involved parsing XML documents.

The XDS Libraries provide a framework for each channel that calls a method you supply whenever a certain event is received. For example, when you driver receives an add event, the XDS Libraries determine the event type for you and call the method you have selected to handle add events. Methods are also supplied to create XDS documents using API calls, relieving you from creating the XDS documents yourself, and documents are validated to reduce the chance of encountering an error.

Each interface is better for different situations, and there are extensive guides to each of the open interfaces on the Web. XDS is simplified and can reduce development time, but it works only with the NDS DTD and cannot be used when processing other types of XML documents. Serial processing is usually avoided if another interface can be used.

These interfaces are discussed in further detail in Section 2.2, Getting Started.

Discovering Changes in Your Application

Typically, this is the most difficult part of creating a custom driver, and unfortunately, it is the part where we can only provide general guidelines. To monitor changes, you first need to determine how the application stores data, what interfaces are exposed in what programming languages, if the application logs data or supplies an event system, and so on.

If you are lucky, your application stores information in XML and has a full-featured event system. In this case you would simply transform the application's XML into XDS whenever an event occurs and pass this to the Identity Manager engine.

In some circumstances, the application might not have even a suitable event log, so you might need to build your own event log, implement a system to read this log, then transform the stored events into XDS in order to make the synchronization work.

As mentioned previously, this is left to your creativity.

Where Should I Start?

If you are new to Identity Manager driver development, Novell DeveloperNet University provides a free, online Identity Manager driver creation training course that is highly recommended. The course is designed to take approximately 24 hours to complete and walks you through the creation of a custom Identity Manager PBX driver.

DeveloperNet University: Custom Driver Development Course

Section 2.0, Writing an Identity Manager Driver discusses modifying the provided skeleton drivers to create your Identity Manager driver.

1.2 Requirements and Resources

In order to develop an Identity Manager driver, you need a test server running one of the following:

- Windows 2003 SP2 or later (32 bit), Windows Server 2008 or later(32 and 64-bit),Windows 2008R2, Windows 2012 R2 with Novell eDirectory 8.8.8 (build 88.x) or later.

- SLES 10, SLES 11 or later, OES 11 SP2,RHEL 6.5 (64-bit) or later versions with Novell eDirectory 8.8.8 (NDS build 88.x) or later.

  NOTE: Some platforms have become outdated with the release of Identity Manager 4.5 and the Metadirectory associated. Refer the documentation for the respective version of Identity Manager.

The eDirectory 8.8.8 server can be installed in an eDirectory tree with servers running earlier versions of eDirectory. You also need a development workstation with JDK 1.17b or later to write a Java Identity Manager driver, or a development workstation with a suitable compiler to write a C++ driver.

The following sections give you additional details on the requirements and the available resources.

1.2.1 Requirements

The Identity Manager driver has the following requirements:

- *eDirectory Version:* The Identity Manager engine runs on Novell eDirectory 8.8.8 or later which is available for Windows and Linux. The Identity Manager driver (or a portion of the driver) must run on the same computer as the Identity Manager engine.

- *eDirectory Objects:* The Identity Manager driver requires a number of eDirectory objects: a DirXML-DriverSet object, a DirXML-Driver object, a DirXML-Subscriber object, a DirXML-Publisher object, and DirXML-Rule objects. It can have optional DirXML-StyleSheet objects. NDS eDirectory 8.8.8 extends the schema to support these objects. Later versions include Identity Manager schema as part of the base schema.

- *External Application:* The application can reside on the eDirectory server or on a remote server. If the application is remote, it must

provide a communication method so that the driver running on the eDirectory server can communicate with the remote application. For example, the Identity Manager driver for the Netscape directory uses LDAP for remote communication.

- The external application must also provide a programming interface that allows the driver to read and write data to the application. Your driver also needs to receive notification of changes from the application. If the external application does not have an event notification system, your driver needs to develop one for the application by logging all modifications to a file, polling the application for modifications, or some other method.

- *Programming Languages:* The driver can be written in Java or C++. The sample code supplied is written in C++ and Java. Java requires JDK 1.1.7b or later.

- *XML, XSL, and XDS:* If the external application does not store the information in XML format, your driver will need to convert the data from its native format to XDS before sending it to eDirectory to store. XDS is the Identity Manager flavor of XML and is documented in the nds.dtd file. When your driver receives updates from the Identity Vault, your driver will need to take the XDS formatted data and convert it to the application's native format.

- XSLT (Extensible Stylesheet Language Transformations) can be used to transform the XDS format to another variant of XML or another standard format such as LDIF.

1.2.2 Resources

The following list contains several of the resources available to assist you in the creation of an Identity Manager driver:

- Custom DirXML Driver Development Course This course is an excellent introduction to the Identity Manager driver development and is highly recommended for new driver developers.

- Novell AppNotes Novell AppNotes regularly adds new articles related to Identity Manager, including architecture, analyzing NDS.DTD, and troubleshooting and debugging.

- Novell Identity Manager Cool SolutionsContains Identity Manager tips and solutions.

1.3 Identity Manager Architecture

In the Identity Manager framework, the Identity Vault is the hub of information. Other applications and directories publish their changes to the Identity Vault, and the Identity Vault sends changes to the applications and directories that have subscribed for them. Thus there are two main flows of data, as discussed in the driver introduction:

- Publisher channel. This is the flow of data to the Identity Vault. This flow is called the publisher channel because other applications are publishing their changes to the Identity Vault.

- Subscriber channel. This is the flow of data from the Identity Vault. This flow is called the subscriber channel because Identity Vault sends changes only to the applications that have subscribed to receive them.

The Identity Manager engine and the Identity Manager drivers are the key components that implement the publisher and subscriber channels and thus connect Identity Vault with the other application.

All data is exchanged in XML (eXtensible Mark-up Language) documents. The Identity Manager engine translates the Identity Vault event into an XML document and uses rules to determine how the modification is sent to the application. The engine uses the following types of rules and style sheets:

- Mapping rules which map the Identity Vault object class names and attribute names with an application's schema names.

- Matching rules which match the Identity Vault entry with an entry in the application.

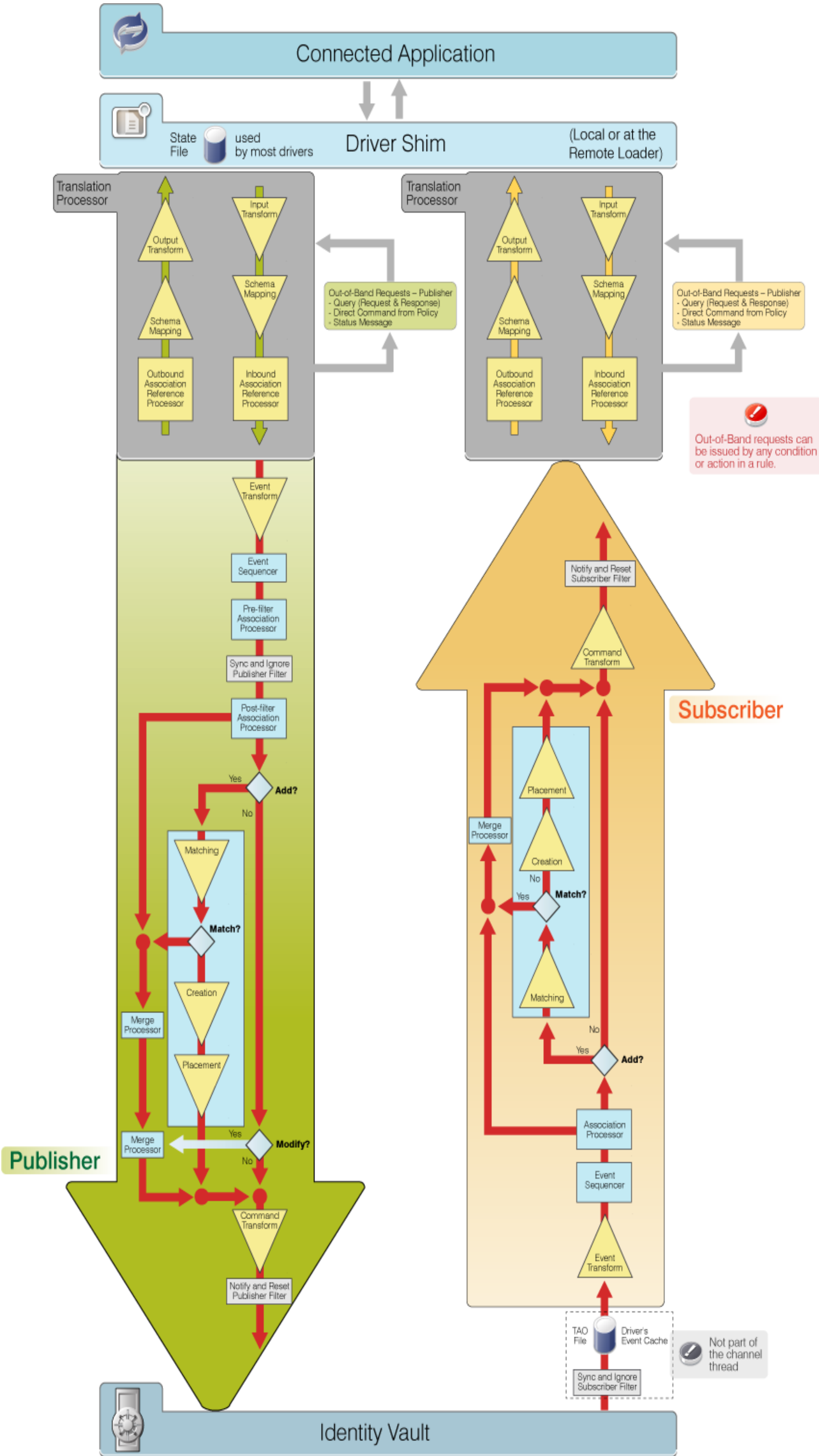
- Create rules which place conditions on creating new entries in the Identity Vault or the application.

- Placement rules which determine where in the Identity Vault or application hierarchy the new entry is created.

- Style sheets which transform input or output commands into a different command, change an event from one type to another, or perform other arbitrary XML transformations.

The figure below illustrates this architecture and shows that the Identity Manager engine does most of the work.

Figure 1-1 Identity Manager Architecture



The Identity Manager driver is designed to be a data pipe. It has the following responsibilities for the subscriber channel:

- Receiving the XML document and transforming the modifications into application commands.
- Interfacing with the application and sending the commands to the application.

The Identity Manager driver has the following responsibilities for the publisher channel:

- Interfacing with the application and obtaining its modifications.
- Transforming the modifications into an XML document and sending it to the Identity Manager engine.

An Identity Manager driver does not need to understand rules and style sheets because the driver has no responsibility for rule processing. The Identity Manager engine is responsible for all rule and style sheet processing.

1.3.1 Identity Manager Features

The flexibility and simplicity of Identity Manager come from the following features.

XML: All Identity Vault data and events are exchanged between the Identity Vault and the application in the format of XML (eXtensible Markup Language) documents. The use of this popular standard allows any XML-aware application to easily consume this data. XML is a subset of SGML (Standard Generalized Markup Language) and more flexible than HTML (Hyper Text Markup Language) because XML allows tags to be defined by the developer. Novell has used XML to define tags for such items as objects, attributes, and values. These definitions can be found in the nds.dtd file. This particular definition of XML tags is referred to as XDS.

Associations: Identity Manager does not need to share a unique ID with the other application. It uses an association attribute which links an Identity Vault entry to an entry in the external application. The attribute is multivalued, so that each Identity Manager driver that is synchronizing data with eDirectory can add its own association. What is placed in the attribute is dependent upon the application. For example, if the application uses a record number to uniquely identify an entry, the record number is put in the association attribute. If the application uses a distinguished name, that is used in the attribute.

Schema Mapping: The Identity Vault and the other applications will have different schema names for the same data. A mapping rule per driver allows you to map any eDirectory attribute or class to any application attribute or class. For example, the eDirectory Surname attribute can be mapped to the Last Name attribute in one application and to the Family Name field in another application.

Authoritative Data Sources: eDirectory rights and Identity Manager filters determine which application has sufficient rights to update an attribute or an entry. For example, if you synchronize with a PBX application, you can make the PBX Identity Manager driver the authoritative data source for telephone number attributes. Other applications with Identity Manager drivers can obtain updates to these attributes but they cannot change them in eDirectory.

Create Rules: Each application that is being synchronized with the Identity Vault might have different conditions for creating a new entry. The create rules can be configured to list those conditions, and the Identity Manager engine will ensure that the entry has values for all the required attributes before it is created.

For example, the Identity Vault requires a name, a surname, and an object class to create a User. If the PBX application requires a first name, surname, telephone number, and location, Identity Manager ensures the entry has values for these attributes before it sends a create command to the PBX application.

Selected Data: Most applications store information that is specific to the application. The Identity Manager architecture allows the administrator to select only the data that is relevant for sharing: the Identity Vault attributes and classes that correspond to relevant application-specific records and fields.

For example, an HR database would want to share user-type objects with the Identity Vault, but would not be interested in network resource objects such as servers, printers, and volumes. The Identity Vault would want to share a user's given name, surname, initials, telephone numbers, and work location, but would probably not be interested in storing the user's family information and employment history (unless another eDirectory application needs access to this information). If the Identity Vault does not currently have classes or attributes for the shared data, the Identity Vault schema can be extended to include them. In this example, the Identity Vault becomes the repository of information that eDirectory needs and which other applications can use. The application remains the repository for the information that is required only by the application.

Data Transformations: Identity Manager uses the Extensible Stylesheet Language Transformation (XSLT) as a mechanism for data transformation. For example, if one external application stores date information in a day-month-year format, an XSL transformation rule

can ensure that date information that is stored in a month-day-year format is converted before sending it to the external application. XSL is an extensible language that allows driver developers and system administrators to supply plug-ins for data transformations not covered in the basic language.

Identity Manager Engine: The Identity Manager engine is responsible for the Identity Vault communication, schema mapping, rule enforcement, and data filtering. Since the engine does most of the work, the Identity Manager driver, in comparison, is relatively simple and direct.

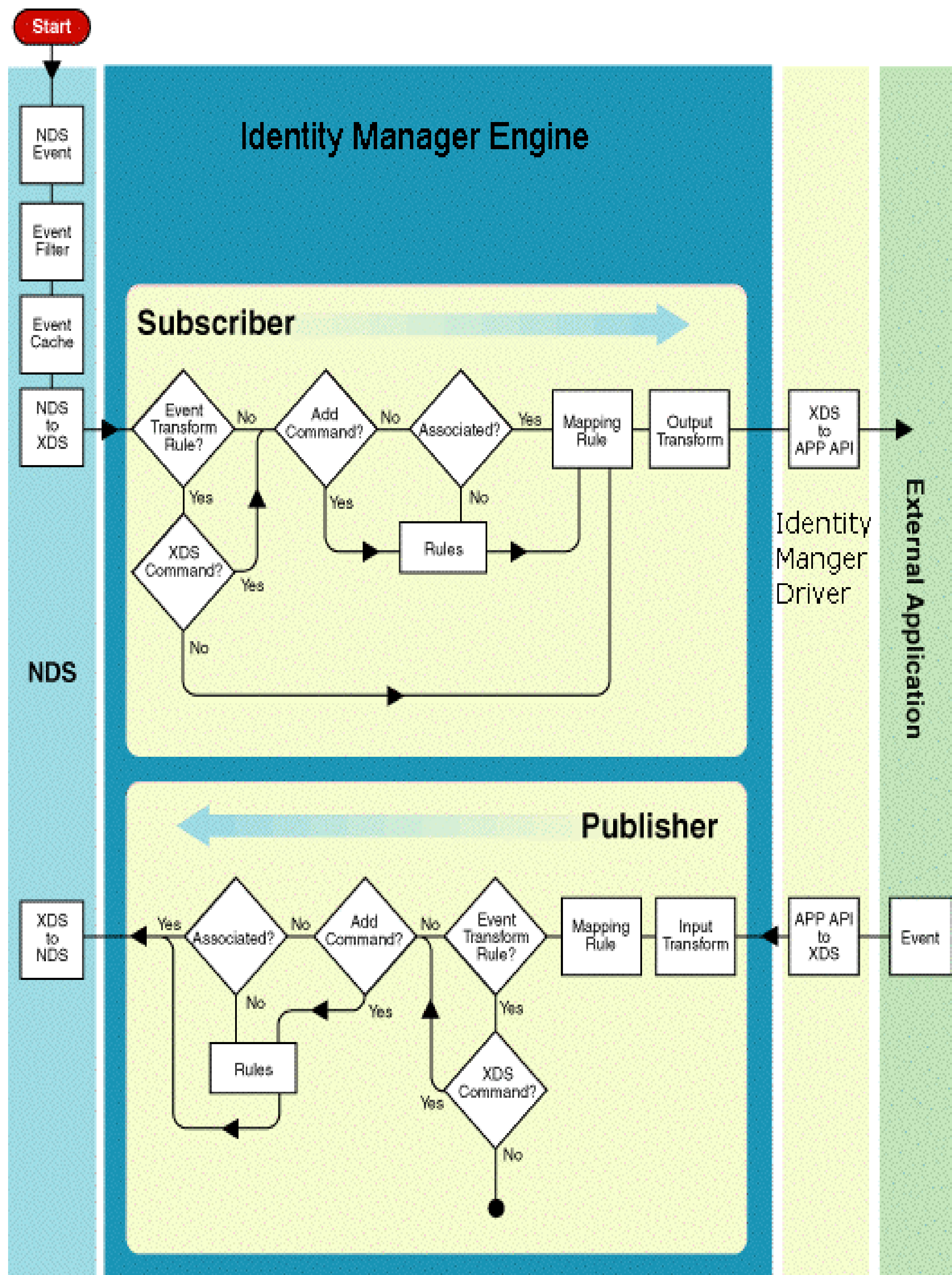
1.3.2 Identity Manager Engine and Driver Interaction

The Identity Manager engine is the key module in the Identity Manager architecture and provides the interface that allows Identity Manager drivers to synchronize external application information with the Identity Vault. The Identity Manager engine is the module that communicates directly with the Identity Vault and converts the Identity Vault data into XML format.

This section covers the following topics:

The following graphic illustrates the interactions among the Identity Manager engine, the Identity Vault, an Identity Manager driver, and an external application.

Figure 1-2 Identity Manager Interactions



When the Identity Vault initializes, it reads the event filter, registers the driver for the appropriate Identity Vault events, filters the data according to the filter's specification, and sets up a cache for the events. The Identity Vault then notifies the Identity Manager engine when an event occurs. The Identity Vault events are local to the server; they are not global to the eDirectory tree. Therefore, the Identity Manager engine receives only the modifications from local replicas. If user data is modified on a partition that does not reside on the local server, Identity Manager is not notified of the modifications. The eDirectory server with the Identity Manager engine must be configured to contain replicas of all the objects that are being synchronized with the external application.

eDirectory 8.8.8 added a new type of replica for filtering data. Filtered replicas allow you to select which object types (for example, Users) and which attributes (for example Surname, Given Name, CN, and Telephone Number) a replica contains. Other object types such as Printers and Servers and User attributes such as Title or Manager are not included in the filtered replica. With eDirectory 8.8.8, the server running Identity Manager can be set up with replicas that include the data specified in the Identity Manager filter and rules.

When the Identity Manager engine receives the Identity Vault events, it reads the rules you have set up for the subscriber (event transformations, matching, placement, create, mapping, and output transformations) and sends the data to your subscriber channel.

The subscriber channel of the Identity Manager driver receives the XML data from the Identity Manager engine, converts it to the application's APIs, and sends it to the external application.

The publisher channel of the Identity Manager driver checks for events in the external application, converts the events into XML, and sends the XML to the Identity Manager engine. The engine applies the rules, changes the XML to the Identity Vault commands, and sends them to the Identity Vault.

Subscriber Channel

The subscriber portion of your driver sends changes from the Identity Vault to your application. It is the object that implements the SubscriptionShim interface and is responsible for the following tasks:

- *Data Conversion.* Your driver needs to convert the XML data to a format that the external application can use. This tool kit includes the nds.dtd file which contains the XML definitions for the input and output commands and for the rules (matching, creation, placement, and mapping). You can also use an output transformation style sheet to help in the conversion. If possible, all data conversion should be handled in rules and style sheets because these can be modified to match a particular installation of the external application. If the driver handles the data conversion, modifications to the process require recompiling the driver.

- *Data Sending.* The driver needs to use the interface of the external application to send the data to the application. The driver should not return to the Identity Manager engine until the application returns a response or a reasonable timeout expires.

- *Response Handling.* The driver needs to convert the response from the application into an XDS document and return it to the Identity Manager engine.

The Identity Manager engine is responsible for filtering the Identity Vault data, converting it to XML, and applying the event transformations, rules, and output transformations. Event transformations can be used to change an XDS command (such as modify) to another XDS command (such as add). As a driver developer, you can also use an event transformation to change an XDS command into a non-XDS command understood by your application. The Identity Manager engine does not apply matching and create rules to non-XDS commands but sends them to the schema mapping rules and then to your driver.

Publisher Channel

The publisher portion of the driver performs the gathering and sending of updates from the external application to the Identity Vault. It is the object that implements the PublicationShim interface. The following tasks are driver-implementation specific.

- *Changes:* Your driver needs to be informed of changes to the information in the external application.

- *Filtering:* Once your driver has gathered the changed information, it should filter it to the correct set of data so that only data being shared is sent to the Identity Vault. If your driver doesn't filter the data, the Identity Vault will filter and discard the data that doesn't apply. However, system resources are used more efficiently when the driver filters the data.

- *Converting:* Your driver needs to convert the data from the application's commands to XDS format and then send the data to the Identity Manager engine. If the application uses XML, you can use input transformations to change from the application's XML to XDS.

- *Sending:* Your driver sends the data to the Identity Manager engine and receives an XML response document.

Once the data is sent to the Identity Manager engine, the Identity Manager engine is responsible for applying the rules and sending it to the Identity Vault. The event transform rules can be used to transform one XDS command into another. They can also convert an external application event into an XDS command. However, if the event is transformed to a format other than XDS, the Identity Manager engine drops it (the black hole in the figure) and returns an error.

The Identity Vault responds to the Identity Manager engine which converts the response to an XML document. The mapping rules, style sheets, and your driver convert the response to a format that you can send to the external application as completion codes, if necessary.

1.3.3 Driver Interaction with the Identity Vault Objects and Attributes

The Identity Manager driver has two main entities with which it communicates: the Identity Manager engine and the external application. However, the driver is affected by information stored in a number of the Identity Vault objects. The Identity Manager engine interacts with these Identity Vault objects. It knows which objects belong to your driver and uses the object information to modify the XDS formatted document that is sent to your driver. For a definition of the Identity Manager classes and attributes, see Section B.0, Identity Manager Definitions for the Schema.

The following Identity Manager attributes and objects are used to modify the document's contents.

- *Filter Attribute for the Subscriber.* The Identity Manager engine registers your driver for a standard set of the Identity Vault events: create object, delete object, rename object, add attribute value, delete attribute value, and move object. The Identity Vault server reads the subscriber filter to restrict the data to the object classes and attributes specified in the filter. For example, if the configuration is intended to synchronize only user information, the filter would specify User objects and modification to other the Identity Vault objects would be ignored. From the possible User class attributes, the filter would specify selected attributes, such as CN, Given Name, Surname, and Telephone Number. Modifications to other user class attributes would be ignored.
- Although your driver never reads the subscriber filter or interacts directly with it, the contents of the filter affects your driver because the filter determines what data is sent from the Identity Vault to your driver.

The subscriber filter is a DirXML-DriverFilter attribute of DirXML-Subscriber objects.

- *Filter Attribute for the Publisher.* The Identity Manager engine uses the publisher filter to ensure that the data it receives from the external application matches the data types specified in the filter. This filter is passed as part of the initialization data so that the driver can filter the events coming from the application to match the data defined in the filter.
- The publisher filter is a DirXML-DriverFilter attribute of DirXML-Publisher objects.
- *Association Attribute.* This attribute is an optional attribute for every object in the Identity Vault tree. It associates an Identity Vault entry with an entry in the external application. Well-designed matching rules automate the creation of associations between existing entries in the Identity Vault and the application. On add operations, the Identity Manager driver returns an association as part of an add command and includes an association as part of an add event.
- *Matching Rule Object.* Matching rules determine how objects in the Identity Vault are associated with records in the external application when an association has not already been established. The administrator sets these up, and they determine which attributes and values must match with which fields and values in the external application. For example, the Identity Vault administrator can set the rule up so that a User's Surname, Given Name, and Telephone Number attributes must match a Record's Last Name, First Name, and Phone fields before the Identity Manager engine can automatically create an association.
- *Create Rule Object.* Create rules specify what information the Identity Vault or the application must have before creating a new object or entry. In the rule, the administrator specifies the attributes that must have values, for example, last name, first name, phone number, and login name. This rule can be the same for both the subscriber and the publisher or different. In other words, one application can require more information than the other for creation to succeed.
- *Placement Rule Object.* Placement rules determine where new objects are placed in the application. Each driver typically requires at least two placement rules. The publisher needs to know where to created new Identity Vault objects when the external application creates a new object, and the subscriber needs to know where to create an external application object when a new object is created in the Identity Vault. You can have multiple rules. Because the Identity Vault is hierarchical, multiple rules are useful because they allow you to create objects in multiple containers.
- *Mapping Rule Object.* Mapping rules determine how the Identity Vault class and attribute names are mapped to application's class and attribute names. For example:

User class	with	Client class
Given Name attribute	with	First Name attribute
Surname attribute	with	Last Name attribute

Initials attribute	with	Middle Initial attribute

- *Configuration Attribute.* The DirXML-ShimConfigInfo attribute is an XML file that contains any configuration options which a network administrator needs to set for the driver, the subscriber, and the publisher. They are optional, but if you select to use them, your driver initialization code needs to parse the commands that are sent in an XDS init document.

- *Style Sheet Objects.* Style sheets are for event, input, and output transformations and may be used for any of the other rules.

For more information on the Identity Vault objects and attributes, see Section B.0, Identity Manager Definitions for the Schema.

For more information about the rules, filters, and style sheets, see Introduction to the Rules and Filters.

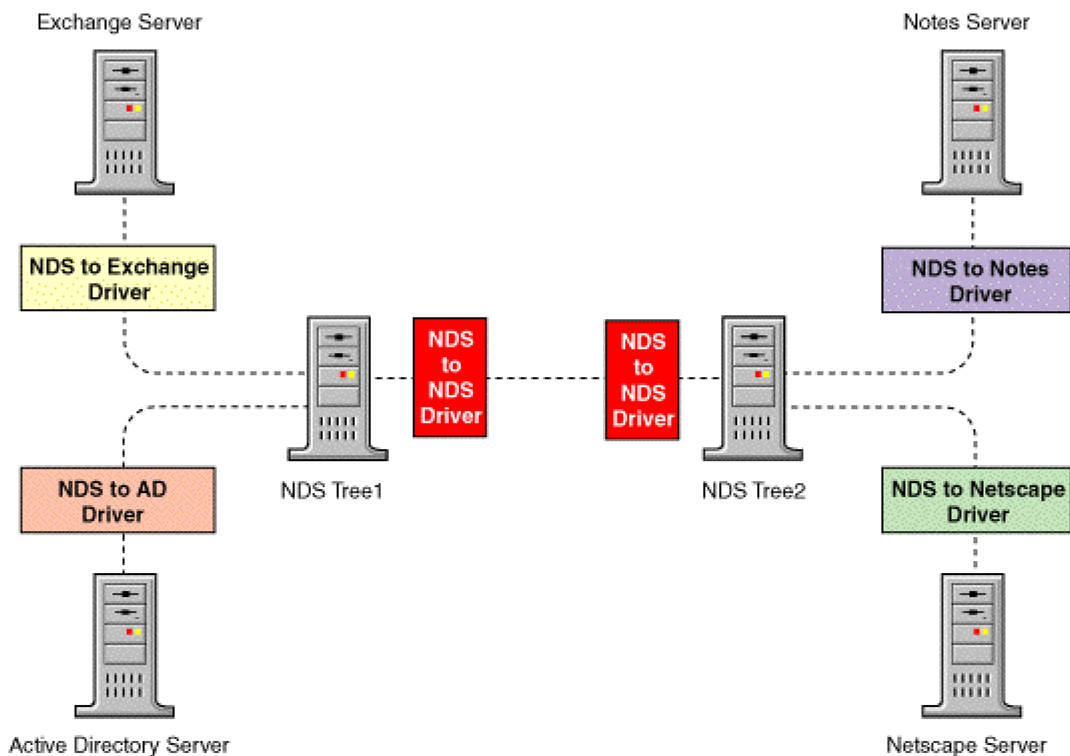
1.4 Identity Manager and Multiple Directories

Most companies have anywhere from 20 to 200 directories, with the "average" company having 180. How many depends upon your network configuration, applications, and number of operating systems. In most companies, the telephone system, human resources, e-mail, and each operating system will have their own directories. Keeping user information current in all of them requires multiple administrators and data entry for each directory. Identity Manager is designed to eliminate such duplicate data entry.

For example, if your company first enters newly hired employees in a human resource database, the Identity Manager driver that connects that database to the Identity Vault can be configured to create new users in the Identity Vault and place them in the appropriate container in the tree based on the department that hired them. Then, when the employee moves to another department and to a different container, the Identity Vault sends these changes to Identity Manager which sends them to the human resource database. The change needs to be made in only one database, and the other database is automatically updated.

If multiple applications are using Identity Manager to synchronize with the Identity Vault, Identity Manager synchronizes shared data with all the applications. The following figure illustrates a configuration with six Identity Manager drivers.

Figure 1-3 Sample Identity Manager configurations with six drivers



Driver 1. The NDS to Exchange driver synchronizes data with eDirectory Tree1.

Driver 2. The NDS to NDS driver synchronizes data from eDirectory Tree1 to eDirectory Tree2.

Driver 3. The NDS to Active Directory driver synchronizes data with eDirectory Tree1, the NDS to NDS driver synchronizes the data to eDirectory Tree2, and the NDS to Exchange driver synchronizes it to Exchange.

Driver 4. The NDS to Notes driver synchronizes data with eDirectory Tree2.

Driver 5. The NDS to NDS driver synchronizes the data from eDirectory Tree2 to eDirectory Tree1, the NDS to Exchange driver synchronizes the data to Exchange, and the NDS to AD driver synchronizes it to Active Directory.

Driver 6. The NDS to Netscape driver synchronizes the data with eDirectory Tree2 and the NDS to Notes driver synchronizes it to Notes. The NDS to NDS driver synchronizes the data to eDirectory Tree1, the NDS to Exchange driver synchronizes the data to Exchange, and the NDS to AD driver synchronizes it to Active Directory

If a change occurs in one application, that change is propagated to the others. For example, if a user name is changed in the Netscape directory, the Identity Manager drivers propagate the change to the other five applications: Active Directory, eDirectory Tree1, eDirectory Tree2, Notes, and Exchange.

1.5 Designing the Driver

The first step in designing a driver is to obtain a thorough understanding of the application for which you are writing a driver. You need to understand its application programming interface, security system, schema, procedures for change notification, operating system requirements, and remote protocol support. An in depth knowledge of an application will allow you to design a simple and efficient driver.

As you design your driver, you need to plan on a method to handle the following:

Data conversion: Your driver design should aim to make the driver a pipe with as much data conversion handled with rules and style sheets as possible. You can modify the rules and style sheets to change data conversion in a particular environment, and such modifications do not require recompiling the driver. If data conversion is handled by the driver, you may need to modify the data conversion code and recompile the driver to make it work in a particular environment.

Remote communication: The Identity Manager driver either runs on the server or is loaded using a remote loader. To use a remote loader, no changes are necessary. Identity Manager runs on Windows , Solaris, and Linux. If your application does not run on one of these operating systems, you must design your driver for remote communication. If your application supports a directory protocol such as LDAP, you can use that protocol for remote communication.

Passwords: To access the external application, your driver may need to log in. Identity Manager can be used to store the login information, and Identity Manager encrypts the application's password before storing it in the Identity Vault. Identity Manager engine version 1.1 and later can synchronize user passwords between the Identity Vault and the external application. For versions earlier than 1.1 use Novell Single Sign-on for password solutions.

Loopback detection: The Identity Manager engine has been designed to detect application loopback. It ensures that a change sent by your driver to the Identity Vault is not sent back to your driver as a change needed in your external application. Your driver is responsible for the Identity Vault loopback. It should ensure that a change sent by the Identity Vault to your driver is not sent back by your driver as a change needed in the Identity Vault.

Driver filter: The Identity Manager engine has been designed to filter changes so that the commands only contain the attributes and classes specified in the filter. However, Identity Manager runs more efficiently when each driver uses its filter to determine what data should be sent to the Identity Manager engine. If your application is remote, the filtering should be enforced, if possible, on the remote machine so that the eliminated data is not sent across the wire.

Data gathering: Your driver needs to use the most efficient and stable method for determining what has changed in the external application. If the application has an event system, you can use the event system to register for the events and efficiently obtain modification of changes. However, if the external application does not have an event system, you will need to implement another method. Many applications generate a log file of changes, and you can use the file to determine what has changed and to track what changes your driver has made and what changes are new. The disadvantage of using a log file is its format often varies from release to release. Since most application vendors view data sharing as a positive feature, work with the vendor's technical support department to determine the best method.

Schema differences: You need to be very familiar with the schema for both the Identity Vault and the external application, and you will need to design your driver to handle the differences. For example, an the Identity Vault attribute definition can flag an attribute as naming, single-valued, or multi-valued. If your external application has no concept of any of these, your driver must be designed to handle the differences.

The class definitions that are being mapped (such as User in the Identity Vault and System User in the external application) will probably contain different attributes. Identity Manager does not require that both applications share the same definition, only that both definitions contain the attributes that are going to be synchronized. If the Identity Vault schema does not contain required attributes, you can extend the schema. Novell recommends using auxiliary classes rather than adding attributes to existing classes. (For more information, see the NDS Schema Reference.)

Attributes will often have different formats for the information. Where possible, you should use style sheets to handle the data transformation.

1.6 Where to Get Started

Chapter	Purpose
Section 2.0, Writing an Identity Manager Driver	Shows how to add the required classes and methods to a skeleton driver.
Section 3.0, Debugging the Driver	Describes how to debug both a C++ and Java driver.
Section 4.0, Introduction to the Rules and Filters	Provides an overview of the rules. For a description of the DTD elements, see Section 8.0, Rule Reference.
Section 6.0, Driver Installation	Describes the installation requirements for a driver.
Section 7.0, DTD Commands and Events	Describes the command and event elements in the NDS DTD.

2.0 Writing an Identity Manager Driver

This chapter describes the basic requirements and architecture needed to write a Identity Manager driver that supports synchronizing data between an application or directory and the Identity Vault.

An Identity Manager driver is compiled code written in either Java or C++ that performs the communication between the Identity Manager engine and an application. Also referred to as a shim, the driver is invariant between installations and installation-specific requirements are addressed by Identity Manager rules and filters.

2.1 Driver Overview

An Identity Manager driver serves primarily as a data pipe between the Identity Manager engine and the application. As such, the driver is not responsible for decisions about what data to synchronize and when to synchronize the data. In general, the driver receives commands from the Identity Manager engine that instruct the driver to modify the application, and the driver reports data events occurring in the application to the Identity Manager engine.

Decisions about what data to synchronize and how to achieve data synchronization are specified on an installation-specific basis by the system administrator through filters and rules.

A general application driver should be written such that all data in an application can be synchronized with the Identity Vault if the Identity Manager rules and filters specify it.

The following sections describe:

- Driver Communication and Threads
- Driver Life Cycle

See the Driver FAQ for a list of Frequently Asked Questions about driver development.

2.1.1 Driver Communication and Threads

There are two channels of communication between the Identity Manager engine and the Identity Manager application driver: the subscriber channel and the publisher channel. The subscriber channel is used for sending commands from the Identity Manager engine to the application driver; for example, the application “subscribes” to changes from the Identity Vault. The Publisher channel is used for sending events from the application driver to the Identity Manager engine; for example, the application “publishes” changes in the application to the Identity Vault.

Conceptually, the Identity Manager driver has (and is typically implemented as) three objects: the driver object, the subscriber object, and the publisher object. The driver object is responsible for initialization that is common to both channels, creating the subscriber object and the publisher object, and for shutting down the driver when instructed by the engine to do so. Each object has an interface it is responsible for implementing. The publisher object typically implements the XmlQueryProcessor interface as well.

Table 2-1 Identity Manager Interfaces

Interface Name	Java Name	C++ Name
DriverShim	com.novell.nds.dirxml.driver.DriverShim	DriverShim in NativeInterface.h
SubscriptionShim	com.novell.nds.dirxml.driver.SubscriptionShim	SubscriptionShim in NativeInterface.h
PublicationShim	com.novell.nds.dirxml.driver.PublicationShim	PublicationShim in NativeInterface.h
XmlQueryProcessor	com.novell.nds.dirxml.driver.XmlQueryProcessor	XmlQueryProcessor in NativeInterface.h

Each channel of communication runs on separate threads of execution, referred to as the subscriber thread and publisher thread.

The subscriber thread is the thread on which

- The driver object is constructed
- The driver object methods are called
- The subscriber methods are called

The subscriber thread spends most of its time in Identity Manager engine code waiting for the Identity Vault events. When an the Identity Vault event occurs, the engine processes the event according to the Identity Manager rules and, depending on the result of the rule processing, may issue a command to the driver on the subscriber channel by calling the subscriber object’s execute method. The Identity Manager engine also uses the subscriber thread to instruct the driver to shut down by calling the driver object’s shutdown method. The driver object’s shutdown method is responsible for causing the publisher thread to return from the publisher object’s start method.

The publisher thread is the thread on which the publisher methods are called. The publisher thread spends most of its time in the publisher object’s start method. When an application change is detected, the publisher object calls the Identity Manager engine to inform the engine of the event using an object passed to the start method.

2.1.2 Driver Life Cycle

There are two separate modes of operations that a driver object must support:

- Normal synchronization

- Schema query

A single driver instance will only be required to operate in a single mode. For example, if a driver instance is constructed and the getSchema method of the DriverShim interface is called, no normal synchronization methods are called on that driver instance. Similarly, if a driver instance is constructed and the init method of the DriverShim interface is called, the getSchema method is never called on that driver instance.

Each time a driver is run, a new instance is obtained. In addition, multiple instances of the same driver may run on a single Identity Manager server (synchronizing different instances of the application, for example). The application driver should have no modifiable global or static data. Global or static data may prevent your driver from operating properly when it is started, stopped, and then started again. Such data may also prevent the driver from operating properly when multiple instances are running on the same server.

The life cycle of a driver instance for normal synchronization is as follows.

Subscriber Thread	**Publisher Thread**
1. The driver object is constructed. • For a Java driver, the engine calls the driver’s no-argument constructor. • For a C++ driver, the engine calls the driver’s CreateDriver function.	
2. The engine calls the driver object’s init method.	
3. The engine calls the driver object’s getSubscriptionShim method.	
4. The engine calls the driver object’s getPublicationShim method.	
5. The engine calls the subscriber object’s init method.	
	1. The engine calls the publisher object’s init method.
	2. The engine calls the publisher object’s start method.
6. The engine calls the subscriber object’s execute method, zero or more times.	3. The publisher object calls the Identity Manager engine via the XmlCommandProcessor object that was passed as a parameter to the start method, zero or more times.
7. The engine calls the driver object’s shutdown method.	4. The publisher thread returns from the start method after notification from the subscriber thread.
8. For a C++ driver, the engine calls the driver object's destroy method. For a Java driver, the engine makes the driver object available for garbage collection.	

A driver instance used for a schema query has the following life cycle.

Subscriber Thread	**Publisher Thread**
1. The driver object is constructed.	Not used.

- For a Java driver, the engine calls the driver’s no-argument constructor.
- For a C++ driver, the engine calls the driver’s CreateDriver function.

2. The engine calls the driver object’s getSchema method.

3. For a C++ driver, the engine calls the driver object's destroy method. For a Java driver, the engine makes the driver object available for garbage collection.

2.2 Getting Started

The following sections describe the decisions you need to make before you start coding:

- Application Requirements
- Language—C++ or Java
- XML Interface

2.2.1 Application Requirements

Writing an application driver for Identity Manager requires a thorough understanding of the target application, including the application’s schema, APIs, authentication, and access requirements. The following areas must be considered.

Application Programming Interfaces (APIs). The APIs of an application are the methods through which your driver will communicate with the application for the purposes of modifying and querying data. These may include access methods such as LDAP or direct function calls through an application-supplied library. The choice of API (when there is one) affects how your driver can be used. For example, if your application supports both a function-call interface and an IP interface, choosing the function-call interface may restrict your driver to running on the same physical server as the application.

Authentication or Log on. Many applications require both users and programs to authenticate or log on to the application before data may be accessed. You must be familiar with the authentication requirements of your application. Identity Manager provides a secure way to store an application's password in the Identity Vault on behalf of your driver, and provides standard locations for specifying other connection and authentication parameters which are passed to your driver as part of the initialization procedure.

Change Notification Mechanism. Publishing data to the Identity Vault from the application requires determining when data change in the application. The application may support an event notification system or a polling mechanism. The application may not directly support any method of notification; in such cases you will have to design a method using the tools the application provides.

Association Values. A Identity Manager association value is a value that uniquely identifies an object or record in the application. The association value can be anything that uniquely identifies the object; it is most convenient if the value is invariant, such as a record number or a GUID (Global Unique Identifier). However, it is possible to use values that change (such as a distinguished name) but your driver must be carefully written to inform Identity Manager when such an association value changes by publishing a <modify-association> event with the old and new association values. Examples of association values used by Novell-written drivers include the following: GUID (NDS to NDS and Active Directory drivers), DN (iPlanet directory server driver), and User Name (NT 4 domain driver).

2.2.2 XML Interface

The Extensible Markup Language (XML) is a standard issued by the World Wide Web Consortium that describes how data are marked up using application defined tags and attributes. All communication of data between the Identity Manager engine and the application driver is in the form of XML documents. An XML document is a collection of data, tree-like in structure, with the data being marked by XML tags and attributes.

Identity Manager supports two standard interfaces for accessing and manipulating XML data: the Document Object Model (DOM) and the Simple API for XML (SAX). In addition, Identity Manager provides access to XML documents in serialized form. Serialized XML, however, is the least convenient way of representing XML data for programmatic use.

The DOM presents a tree-like view of the XML document and is typically the most convenient way to access and manipulate XML data programmatically. SAX presents XML documents as a series of events for which handlers are registered. SAX is typically used when only a small part of an XML document is interesting and when navigation among the XML data is not required.

Novell also provides an API designed to make handling DOM documents much easier, called XDS Libraries. The elements in a DOM tree are abstracted to a Java API making it easier to access XML data, as well as construct valid xml documents and driver parameters.

The following table outlines several advantages and disadvantages to each XML interface. There are also many resources on the Internet that helps to determine which XML interface better suits your driver requirements.

XML Interface	Advantages	Disadvantages
DOM	• Tree model is more intuitive • More convenient data access	• Often requires more memory
SAX	• Requires less memory • Event driven model can be faster and more efficient	• Difficult to randomly access data • Event driven model is less-intuitive
XDS Libraries	• Document and driver parameter validation • Increased stability • Reduced complexity and development time • Reduced driver memory signature (Java only)	• Does not support SAX • Non-standard Interface • Increased processing and memory overhead

The XML documents used for communication between the Identity Manager engine and the application driver have well-defined tags and attributes for describing the data. The "flavor" or "dialect" of XML used is specific to Identity Manager and is often referred to as XDS (XML Directory Services).

2.2.3 Language—C++ or Java

You must decide whether to write your application driver in Java or in C++. In general, Java is the preferred choice because using Java allows your driver to run on all of the platforms on which Identity Manager runs. For example, the Novell NDS to NDS driver is written in Java and runs unchanged (using the same .jar file) on all of the platforms on which Identity Manager runs (currently Windows NT, Windows 2000, NetWare 5.x, Solaris, and Linux).

In addition, choosing Java as your implementation language relieves you of the memory management issues associated with C++.

However, there are instances where writing a C++ driver makes more sense. Reasons may include the lack of a suitable interface to the application easily usable from Java or application programming tools provided only in C or C++.

C++ Considerations

The interfaces that your driver must implement are found in the header file NativeInterface.h.

The Identity Manager SDK provides a number of utility functions to perform common tasks necessary in your driver. These include factory functions for creating concrete implementations of various Identity Manager interfaces and support for creating Identity Manager-specific XML documents. The utility functions may be found in the following header files: InterfaceFactory.h, Trace.h, NdsDtd.h, Base64Codec.h, UTFConverter.h, and DriverFilter.h.

There are no standard language bindings for the DOM and SAX to C++. Therefore, Novell provides a binding for these two interfaces. The language binding for DOM is found in dom.h. The language binding for SAX is found in sax.h. Factories for creating concrete DOM objects and SAX implementations are found in InterfaceFactory.h.

In order to use the factories and utility functions it is necessary to link your application with the appropriate import library for your platform.

Library	Platform
dirxmllib.lib	Windows 2003 and above
libdirxml.so	Solaris, Linux, and Tru64 Unix

For more information about these functions, see XML Interfaces for C++.

Java Considerations

The interface definitions and utility classes provided by Identity Manager are found primarily in the package com.novell.nds.dirxml.driver. In addition, many useful classes may be found in the XML/XSLT support packages such as com.novell.xml and descendants. For more information, see the javadocs.

There are standard language bindings for the DOM interface and for the SAX interface supported by Identity Manager. These bindings are found in org.w3c.dom and org.xml.sax.

The Identity Manager classes are found in dirxml.jar and nxsl.jar; you will need to include these .jar files in your class path.

You should compile your Java driver against JDK version 1.6 or before. Compiling on version 1.7 and above is not supported yet.

Note that in Identity Manager 4.x, you can add a new jar file without shutting down eDirectory. You will still need to shut down eDirectory to update an existing jar file.

2.2.4 Overview of the Process

The following sections outline the suggested steps for writing an Identity Manager driver. The instructions start with setting up the skeleton driver, which is used as sample code for the rest of the instructions. The instructions then explain how to develop the driver object and the required interfaces:

- Section 2.4, Constructing the Driver Object
- Section 2.5, Implementing the DriverShim Interface
- Section 2.6, Implementing the SubscriptionShim Interface
- Section 2.7, Implementing the PublicationShim Interface
- Section 2.8, Implementing the XmlQueryProcessor Interface

Novell recommends developing the subscriber channel first since this channel will help you work out communication issues with the application, schema differences, and the conversion of XML commands to application commands. The publication channel requires you to deal with these issues as well as ensuring that the XML documents created by your driver include all the required information.

This chapter also includes information about the following topics:

- Section 2.9, Dealing with XML Documents
- Section 2.10, Driver State
- Section 2.11, Driver Configuration
- Section 2.12, Additional Tips for C++ Drivers

2.3 Starting with the Skeleton Driver

Novell provides a "skeleton" driver as a place to start writing a driver. The skeleton driver is implemented in both Java and C++, and basic or XDS Library-enabled; the implementations are functionally equivalent. The sample code in this chapter is primarily from the skeleton driver implementations.

The recommended way of starting to write an application driver is to start with the skeleton driver code.

- For a Java driver, you must rename the classes to reflect your driver and it is recommended that you put your code in an appropriate package. If this is not done, you may find that your driver conflicts with another driver in an installation if both drivers are based on the skeleton driver and neither changes class names or packages.
- For a C++ driver, you should rename the .dll, .nlm, or shared library to avoid potential conflicts.

The skeleton driver implements all of the required interfaces and demonstrates how various driver tasks are done. For example, the init methods parse initialization parameters, the PublicationShim start method implements a polling loop, and the SubscriptionShim execute method demonstrates how to parse the XML command documents.

Studying the skeleton driver code before starting on your driver is highly recommended. The following table lists the location of each skeleton driver:

Driver	Location
Java Skeleton	[install root]\samples\skeletondriver\dom\java\com\novell\nds\dirxml\driver\skeleton
C++ Skeleton	[install root]\samples\skeletondriver\dom\cpp
XDS Library Java Skeleton	[install root]\samples\skeletondriver\xds\java\com\novell\nds\dirxml\driver\xds\skeleton
XDS Library C++ Skeleton	[install root]\samples\skeletondriver\xds\cpp

2.3.1 Setting Up a Skeleton Driver Instance to Run

As a starting point, you should install a skeleton driver instance (Java or C++, depending on your language choice, and basic or XDS enabled). Since these drivers do not come with an installation program, you will need to copy the driver executable to the server and then use iManager to create the Identity Vault objects and configure the driver. To install the driver, complete the following steps.

1. Copy the driver executable to the server.
    - For the Java skeleton driver, copy the vr_skel.jar to the lib directory.
        - On Windows, this is usually the c:\novell\nds\lib directory.
        - On Unix, copy skeleton.jar to the /opt/novell/eDirectory/lib/dirxml/classes directory.
    - For the XDS libraries Java skeleton driver, copy the XDS.jar and XDSSkeletonShim.jar to the lib directory.
    - For the C++ Windows driver, copy the CSkeletonDriver.dll to the Identity Vault folder. This is usually the c:\novell\nds directory.
    - For the XDS Libraries C++ Windows driver, copy the CppSkeletonDriver.dll to the eDirectory folder. This is usually the c:\novell\nds directory.
    - For the C++ NetWare driver, copy the skeldrvr.nlm to the sys:\system directory.

- For the C++ Unix driver, copy libcskeldrv.so to the /usr/lib/nds-modules directory

2. During the creation process select the following XML driver configuration file.

    - For the Java skeleton driver, this is the j_skel.xml file.
    - For the C++ skeleton driver, this is the c_skel.xml file.
    - For the XDS Libraries Java skeleton driver, this is the JavaXDSSkeleton.xml file.
    - For the XDS Libraries C++ skeleton driver, this is the cpp_skel_options.xml file.

2.3.2 Compiling the Java Skeleton Driver

To compile the Java skeleton driver on NetWare and Windows, complete the following steps.

1. Place the following skeleton java files in a separate directory

    - SkeletonDriverShim.java
    - SkeletonPublicationShim.java
    - SkeletonSubscriberShim.java
    - CommonImpl.java

    The three *Shim files define the main classes to which you will add code.

2. Copy the following files to the same directory that you put the skeleton source files:

    - vrd.jar
    - nxsl.jar

3. Compile the java files into classes.
4. Jar them into a jar name of your choice.
5. Copy your jar into an eDirectory “lib” directory.

For Win32 platforms, this is usually the c:\novell\nds\lib directory.

The driver initialization code looks for jars in the “lib” directory. If you use classes instead of jars then the driver initialization code looks in the “classes” directory on the same level as the “lib” directory.

To compile the Java skeleton driver on Unix, complete the following steps.

1. Make changes to the source files in the Ndk/samples/dirxml_samples/Java/skeleton directory.
2. Set the PATH variable.

    - On Linux systems set PATH=[base path for JDK1.6]/bin.
    - On Solaris systems set PATH=[base path for JDK1.6]/bin.
    - For Tru64 Unix systems set PATH=[base path for JDK1.6]/bin.

3. Compile the sources into classes and jar them. A Makefile is provided for compilation.
4. Copy the jar file into the /opt/novell/eDirectory/lib/dirxml/classes directory.

2.3.3 Compiling the C++ Skeleton Driver

You will need to link the skeleton driver to the Identity Manager library for your platform. The Identity Manager SDK provides the following libraries in the \C\lib directory:

Library	Platform
dirxmllib.lib	Windows
libdirxml.so	Solaris, Linux, and Tru64 Unix

The header files are found in the \C\inc directory.

Complete the following steps for Windows.

1. Using the header files, compile the skeleton driver into a DLL for a windows server.
2. Copy the driver to the appropriate directory on the eDirectory server:

   On NT, to the Novell\nds directory (or the location of the DHost.exe file)

Complete the following steps for Unix.

1. Make changes to the source files in the Ndk/samples/dirxml_samples/C/skeleton directory.
2. Using the header files in the C/include directory, compile the skeleton driver into a shared library. A Makefile is provided.
3. Copy the driver to the appropriate directory.

2.4 Constructing the Driver Object

Each time an application driver is run, the Identity Manager engine obtains a new instance of the DriverObject. A Java driver is constructed by calling a no-argument constructor. A C++ driver is constructed by calling the CreateDriver function that is exported from the driver module (such as a DLL, NLM, or shared library).

The Identity Manager engine starts the driver. The Identity Manager driver stores the name of its executable in one of the attributes of its DirXML-Driver object. If the driver is a Java application, it stores the name in the DirXML-JavaModule attribute. If the driver is a native module (DLL, NLM, or shared library), the driver stores the name in the DirXML-NativeModule attribute.

2.4.1 Java Constructor

The following sample code illustrates how to create the constructor for the driver.

```
public SkeletonDriverShim ()
```

2.4.2 CreateDriver Function for C++

Your driver module must export a factory method for creating a new instance of your driver object. The factory method must be exported by name.

The CreateDriver function has the following syntax:

```
DriverShim * CreateDriver(
    void);
```

On Win32 platforms, one of three names may be used (due to compiler name generation): "CreateDriver", "_CreateDriver", or "_CreateDriver@0". The following sample code illustrates how to create the function for the Win32 platforms.

```
extern "C" CSKELETONDRIVER_API DriverShim * METHOD_CALL
CreateDriver(void)
{
   return new CSkeletonDriver();
}
```

On NetWare the function name must consist of "CreateDriver" with the uppercase name of the NLM appended to it. For example if the driver NLM is named COOLDRVR.NLM then the exported name must be "CreateDriverCOOLDRVR"). The following code illustrates how to create the function for the NetWare platform.

```
extern "C" DriverShim *
CreateDriverSKELDRVR()
{
   DriverShim   * shim = new NW_DriverShim();
   //add to our housekeeping stuff
   addShim(shim);
   return   shim;
}
```

2.5 Implementing the DriverShim Interface

The DriverShim interface consists of five methods in Java and six methods in C++:

- DriverShim init—performs channel-independent initialization.
- DriverShim getSubscriptionShim—returns a reference (Java) or pointer (C++) to the object implementing the SubscriptionShim interface.
- Driver getPublicationShim—returns a reference (Java) or pointer (C++) to the object implementing the PublicationShim interface.
- DriverShim shutdown—notifies the driver to disconnect from the application, cleanup, and otherwise shutdown.
- DriverShim getSchema—called to obtain a representation of the application schema.
- DriverShim destroy (C++ only)—called to free all resources used by the driver, including the driver object itself.

Each of these methods is described in detail in the following sections.

DriverShim init

Initializes the Identity Manager driver object.

Syntax

Java

```
XmlDocument init(
   XmlDocument   initParameters);
```

C++

```
#include "NativeInterface.h"

XmlDocument * METHOD_CALL init(
   XmlDocument   *initParameters);
```

Parameters

initParameters

(IN) Points to an XML document that contains the initialization data for the Identity Manager driver.

Return Values

Returns an XML document containing a status report on the initialization process. The status can be "success", "warning", "error" or "fatal". If "fatal" is returned, the driver start is aborted.

Remarks

The init method in the driver is typically where the subscriber and publisher objects are created. It is also where the initialization parameters are parsed for channel-independent configuration parameters such as application server name or IP address and for authentication parameters required for logging in to the application.

The initParameters argument is an XML document containing initialization data such as authentication information, driver-specific parameters, and driver state. The format of individual driver-specific parameters and driver state are specific to your driver.

When the Identity Manager engine calls the DriverShim init method, it sends an XML document similar to the following:

```
<nds dtdversion="1.0" ndsversion="8.5">
   <source>
      <product version="1.0">DirXML</product>
      <contact>Novell, Inc.</contact>
   </source>
   <input>
      <init-params src-dn="\PERIN-TAO\novell\Driver Set\Java Skeleton Driver">
```

```
         <authentication-info>
            <server>server.app:400</server>
            <user>User1</user>
            <password><!-- content suppressed --></password>
         </authentication-info>
         <driver-options>
            <option-1 display-name="Sample String option">
This is a string</option-1>
            <option-2 display-name="Sample int option (enter
an integer)">10</option-2>
         </driver-options>
      </init-params>
   </input>
</nds>
```

Most of the information in the initialization document corresponds to information configured using iManager/Designer in the DirXML-Driver object properties dialog.

The src-dn attribute in the <init-params> element is the distinguished name of the DirXML-Driver object that is used to contain configuration information for the skeleton driver.

The content of the <authentication-info> element corresponds to the driver authentication parameters found in the properties dialog. The content of the <password> element is suppressed because the trace facility suppresses sensitive data (as defined by Identity Manager). The actual password value is available to the driver.

The content of the <driver-options> element corresponds to driver-specific options specified in the properties dialog. Driver-specific options are specified using an XML file that describes the options. The following example is an XML file used for the skeleton driver:

```
<?xml version="1.0" encoding="UTF-8"?>
  <driver-config name="Skeleton Driver">
     <driver-options>
        <option-1 display-name="Sample String option">This is a string</option-1>
        <option-2 display-name="Sample int option (enter an integer)">10</option-2>
     </driver-options>
     <subscriber-options>
        <sub-1 display-name="Sample Subscriber option">String for Subscriber</sub-1>
     </subscriber-options>
     <publisher-options>
        <pub-1 display-name="Sample Publisher option">String for Publisher</pub-1>
        <polling-interval display-name="Polling interval in seconds">4</polling-
interval>
     </publisher-options>
  </driver-config>
```

Using such an XML file to describe driver-specific configuration parameters allows the administrator to easily configure your driver using existing eDirectory management tools such as iManager.

The driver must return a status document as the response to the init method. The following are examples of potential status documents.

Example 1

```
<nds dtdversion="1.0" ndsversion="8.5">
     <output>
        <status level="success"/>
     </output>
  </nds>
```

Example 2

```
<nds dtdversion="1.0" ndsversion="8.5">
     <output>
        <status level="warning">No authentication information</status>
     </output>
  </nds>
```

Sample Code

Java Sample Code

The following code from the Java skeleton driver sets up a trace message, gets the authentication and configuration parameters from the initParameters document, creates the subscriber and publisher objects, and returns a status document.

```
public XmlDocument init(XmlDocument initParameters )
{
   try
   {
      tracer.trace("init");
      //create an output document for returning
      //status to DirXML
         Element output = createOutputDocument();

         //setup the shared authentication information
         authParams = getAuthenticationParams( initParameters.getDocument());
         //If we don’t have any authentication parameters,
         //report a warning. This is intended to serve as an
         //example of how to report a warning or an error.
         //A real driver may or may not need information
         //in the authentication parameters.
         if (authParams.authenticationId == null &&
            authParams.authenticationContext == null &&
```

```
            authParams.applicationPassword == null)
         {
            //a real driver would probably want to report
            //a fatal error if required parameters
            //are not supplied
            addStatusElement(output,STATUS_WARNING,"No authentication
information",null);
         }

         //get any non-authentication options from
         //the init document
         params = getShimParams(initParameters.getDocument(),
"driver",DRIVER_PARAMS);

         //create the objects that do the real work
         subscriptionShim = new SkeletonSubscriptionShim (authParams);
         publicationShim = new SkeletonPublicationShim (authParams);

         //if we didn't already add a status element,
         //add a success
         if (output.getElementsByTagName("status").item(0) == null)
         {
            addStatusElement(output,STATUS_SUCCESS,null,null);
         }

         //return the status document
         return   new XmlDocument(output.getOwnerDocument());
      } catch (Throwable t)
      {
         //something bad happened...
         return   createStatusDocument(STATUS_FATAL, t.getMessage());
   }
}
```

In the Identity Manager sample code, see the init method in the SolutionDriverShim.java file and the DriverShimImpl.java file.

C++ Sample Code

The following code from the skeleton driver sets up a trace message, gets the authentication and configuration parameters from the initParmeters document, creates the subscriber and publisher objects, and returns a status document.

```
XmlDocument * METHOD_CALL CSkeletonDriver::init(
```

```
    XmlDocument * initParameters)

{
    try
    {
        common.tracer->trace("init");
        //NOTE: only trace this here to test the trace facility.
        //The document is already traced by the engine
        //so it will appear twice in the trace screen
        common.tracer->trace(initParameters);

        //create an output document for returning status to DirXML
        Element * output = NdsDtd_newOutputDocument();

        //setup the shared authentication information
        authParams = common.getAuthenticationParams(initParameters->getDocument());
        //If we don't have any authentication parameters, report a warning.
        //This is intended to serve as an example of how to report a warning
        //or an error. A real driver may or may not need information
        //in the authentication parameters.
        if (authParams->authenticationId == 0 &&
            authParams->authenticationContext == 0 &&
            authParams->applicationPassword == 0)
        {
            //A real driver would probably want to report a fatal error if required
           //parameters are not supplied
            NdsDtd_addStatus(output,STATUS_LEVEL_WARNING,MSG_NO_AUTH_INFO,0);
        }

        //get any non-authentication options from the init document
        params = common.getShimParams(initParameters-
>getDocument(),TEXT_DRIVER,DRIVER_PARAMS);

        //create the objects that do the real work
        subscriptionShim = new SkeletonSubscriber(authParams);
        publicationShim = new SkeletonPublisher(authParams);

        //if we didn't already add a status element, add a success
        if (output->getElementsByTagName(common.ndsDtd->TAG_STATUS)->item(0) == 0)
```

```
        {
           NdsDtd_addStatus(output,STATUS_LEVEL_SUCCESS,0,0);
        }

        //return the status document
        return common.setReturnDocument(output->getOwnerDocument());
    } catch (ShimException e)
    {
        return
common.setReturnDocument(common.createStatusDocument(STATUS_LEVEL_FATAL,e.getMessag
e()));
    } catch (...)
    {
        //something bad happened...
        return
common.setReturnDocument(common.createStatusDocument(STATUS_LEVEL_FATAL,MSG_BAD));
    }
}
```

In the sample code, see the init method in the DriverShimImpl file and the parseInitParams method in the CommonImpl.cpp file.

DriverShim getSubscriptionShim

Returns a reference to the subscriber object.

Syntax

Java

```
public SubscriptionShim getSubscriptionShim ()
```

C++

```
#include "NativeInterface.h"

SubscriptionShim * METHOD_CALL getSubscriptionShim (
   void);
```

Remarks

The Identity Manager engine calls the getSubscriptionShim method to obtain the driver's implelemtation of the SubscriptionShim

interface. Typically, the object implementing SubscriptionShim is constructed in either the Driver object constructor or the DriverShim init method.

Sample Code

Java Sample Code

The following sample code from the Java skeleton driver returns a reference to the subscriber.

```
public SubscriptionShim getSubscriptionShim()
{
   tracer.trace("getSubscriptionShim");
   return subscriptionShim;
}
```

C++ Sample Code

The following sample code from the C++ skeleton driver sends a trace message and returns a handle to the subscriber.

```
SubscriptionShim * METHOD_CALL
CSkeletonDriver::getSubscriptionShim(void)
{
   common.tracer->trace("getSubscriptionShim");
   return subscriptionShim;
}
```

Driver getPublicationShim

Returns a reference to the publisher object.

Syntax

Java

```
public PublicationShim getPublicationShim ()
```

C++

```
#include "NativeInterface.h"

PublicationShim * METHOD_CALL getPublicationShim (
```

```
   void);
```

Remarks

The Identity Manager engine calls the getPublicationShim method to obtain the driver's implementation of the PublicationShim interface. Typically, the object implementing PublicationShim is constructed in either the driver object constructor or the DriverShim init method.

Sample Code

Java Sample Code

The following sample code from the skeleton driver returns a reference to the publisher.

```
public PublicationShim getPublicationShim()
{
   tracer.trace("getPublicationShim");
   return publicationShim;
}
```

C++ Sample Code

The following sample code from the skeleton driver sends a trace message and returns a handle to the publisher.

```
PublicationShim * METHOD_CALL CSkeletonDriver::getPublicationShim(void)

{
   common.tracer->trace("getPublicationShim");
   return publicationShim;
}
```

DriverShim shutdown

Shuts down the Identity Manager driver.

Syntax

Java

```
XmlDocument shutdown (XmlDocument
   reason);
```

C++

```
#include "NativeInterface.h"


XmlDocument * METHOD_CALL shutdown(
   XmlDocument   *reason);
```

Parameters

reason

(IN) Points to an XML document that contains the reason for the shutdown. Currently NULL is sent because the Identity Manager engine does not supply a reason.

Return Values

Returns an XML document containing the results of the shutdown operation.

Remarks

The Identity Manager engine calls this method on the subscriber thread to inform the driver that it is to cease all processing and shutdown in an orderly fashion. The shutdown method is responsible for informing the publisher (running on a different thread) that it must return from the PublicationShim start method. After the shutdown method returns, the Identity Manager engine waits for up to 60 seconds for the publisher thread to return from the start method.

The "reason" argument is intended so that the Identity Manager engine can pass an XML document detailing the reason for the shutdown (such as user action or server going down) but currently a null argument is passed. The return document is a status document detailing the success or failure of the operation. The following is an example return document from the skeleton DriverShim shutdown method:

```
<nds dtdversion="1.0" ndsversion="8.5">
   <output>
      <init-params>
         <subscriber-state>
            <current-association>6</current-association>
         </subscriber-state>
      </init-params>
      <status level="success"/>
   </output>
</nds>
```

This document contains a <subscriber-state> element. The skeleton Subscriber saves some state information between invocations and uses the return document from the shutdown method to cause the Identity Manager engine to write the state information. The engine returns the state information to the driver in the SubscriptionShim init method.

Sample Code

Java Sample Code

The following sample code from the skeleton driver calls a stop method on the publisher. It then returns an XmlDocument document that contains the status of the shutdown.

```
public XmlDocument shutdown(XmlDocument reason)
{
   tracer.trace("shutdown");
   //create an output document so the subscriber can
   //write its state info
   Element output = createOutputDocument();
   subscriptionShim.setState(output);

   // shutdown whatever needs shutting down
      publicationShim.stop();

   //add a successful status
   addStatusElement(output,STATUS_SUCCESS,null,null);

   //return the status and state to DirXML
   return   new XmlDocument(output.getOwnerDocument());
}
```

In the Identity Manager sample code, see the shutdown method in the SolutionDriverShim.java file and the DriverShimImpl.java file.

C++ Sample Code

The following sample code from the skeleton driver sends a trace message, calls a stop method on the publisher. It then returns to the Identity Manager engine an XmlDocument that contains the status of the shutdown.

```
XmlDocument * METHOD_CALL  CSkeletonDriver::shutdown(XmlDocument * reason)
{
   try
   {
      common.tracer->trace("shutdown");
      common.tracer->trace(reason);
         //create an output document so the subscriber can write its state info
      Element * output = NdsDtd_newOutputDocument();
      subscriptionShim->setState(output);
```

```
        // shutdown whatever needs shutting down
        publicationShim->stop();

        //add a successful status
        NdsDtd_addStatus(output,STATUS_LEVEL_SUCCESS,0,0);

        //return the status and state to DirXML
        return common.setReturnDocument(output->getOwnerDocument());
    } catch (...)
    {
        //something bad happened...
        return
common.setReturnDocument(common.createStatusDocument(STATUS_LEVEL_FATAL,MSG_BAD));
    }
}
```

In the Identity Manager sample code, see the shutdown method in the DriverShimImpl.cpp file.

DriverShim getSchema

Returns an XML document that defines the schema of the application.

Syntax

Java

```
public XmlDocument getSchema (
   XmlDocument   initParameters)
```

C++

```
#include "NativeInterface.h"

XmlDocument * METHOD_CALL getSchema (
   XmlDocument   *initParameters);
```

Parameters

initParameters

(IN) Points to an XML document that contains the configuration parameters for the driver so that the driver can authenticate to the application to obtain the schema.

Remarks

The Identity Manager engine calls the getSchema method to obtain an XML document containing a representation of the application's schema. The getSchema method is called on a driver instance constructed explicitly for the query schema operation. After the getSchema method returns, the instance used for the getSchema called is destroyed (C++) or made available for garbage collection (Java).

When a driver is run for the first time, the Identity Manager engine calls the getSchema method before starting the driver for normal synchronization. This is done because the engine requires a schema representation to properly perform merges between the Identity Vault and the application objects that are associated through a matching rule. In addition, iManager requires the application schema for the Mapping Rule policies to function correctly. When the schema has been obtained, it is stored in the DirXML-ApplicationSchema attribute on the DirXML-Driver object corresponding to the driver.

The "initParameters" argument contains a document that contains the same initialization parameters as are sent to the DriverShim init method, as well as the <subscriber-options> and <publisher-options> elements that are sent to SubscriptionShim init and PublicationShim init methods. The return document should contain either a status, in the case of an error, or the schema representation.

If the application has a modifiable schema the driver should query the application and build the XML schema representation from the results of the query. If the application has an invariant schema, a reasonable implementation is to place a serialized XML document representing the schema into a string in the driver and return that XML document.The following is an example of the response to a getSchema call from the VRTest driver.

```
<nds dtdversion="1.0" ndsversion="8.5">
   <output>
      <schema-def hierarchical="true">
         <class-def class-name="O" container="true">
            <attr-def attr-name="Name" case-sensitive="false" multi-valued="false"
naming="true" read-only="false" required="false" type="string"/>
            <attr-def attr-name="Object Path" case-sensitive="false" multi-
valued="false" naming="false" read-only="false" required="true" type="string"/>
            <attr-def attr-name="Unique Id" case-sensitive="false" multi-
valued="false" naming="false" read-only="false" required="true" type="string"/>
         </class-def>
         <class-def class-name="OU" container="true">
            <attr-def attr-name="Name" case-sensitive="false" multi-valued="false"
naming="true" read-only="false" required="false" type="string"/>
            <attr-def attr-name="Object Path" case-sensitive="false" multi-
valued="false" naming="false" read-only="false" required="true"
               type="string"/>
            <attr-def attr-name="Unique Id" case-sensitive="false" multi-
valued="false" naming="false" read-only="false" required="true" type="string"/>
         </class-def>
         <class-def class-name="User Object" container="false">
            <attr-def attr-name="name" case-sensitive="false" multi-valued="false"
naming="true" read-only="false" required="true" type="string"/>
            <attr-def attr-name="Family Name" case-sensitive="false" multi-
valued="false" naming="false" read-only="false" required="false" type="string"/>
            <attr-def attr-name="First Name" case-sensitive="false" multi-
valued="false" naming="false" read-only="false" required="false" type="string"/>
```

```
            <attr-def attr-name="Telephone" case-sensitive="false" multi-valued="true" naming="false" read-only="false" required="false" type="string"/>

            <attr-def attr-name="Object Path" case-sensitive="false" multi-valued="false" naming="false" read-only="false" required="true" type="string"/>

            <attr-def attr-name="Unique Id" case-sensitive="false" multi-valued="false" naming="false" read-only="false" required="true" type="string"/>

         </class-def>

         <class-def class-name="Bogus" container="false">

            <attr-def attr-name="Whatever" case-sensitive="false" multi-valued="true" naming="true" read-only="false" required="false" type="string"/>

            <attr-def attr-name="Object Path" case-sensitive="false" multi-valued="false" naming="false" read-only="false" required="true" type="string"/>

            <attr-def attr-name="Unique Id" case-sensitive="false" multi-valued="false" naming="false" read-only="false" required="true" type="string"/>

         </class-def>

      </schema-def>

   </output>

</nds>
```

For information on the format of this schema document, see <schema-def>.

The following code examples show how the skeleton driver implements the getSchema method.However, since the skeleton driver doesn’t support an actual application, an error is returned rather than a schema representation.

Sample Code

Java Sampe Code

The sample code in the Java skeleton driver returns a document stating the driver doesn't support this feature.

```
public XmlDocument getSchema(XmlDocument initParameters)

{

   //setup the shared authentication information

   authParams = getAuthenticationParams(initParameters. getDocument());


   //However, since we are just a skeleton, this code

   //creates a return document that says we can’t do it.

      return   createStatusDocument(STATUS_ERROR,"Skeleton driver doesn’t support the getSchema operation");

}
```

In the Identity Manager sample code, see the getSchema method in the SolutionDriverShim.java file and the DriverShimImpl.java file.

C++ Sampe Code

The following sample code from the skeleton driver sends a trace message, obtains the authentication information, returns a document that the driver is unable to read the schema.

```
XmlDocument * METHOD_CALL  CSkeletonDriver::getSchema(
   XmlDocument * initParameters)
{
   try
   {
      common.tracer->trace("getSchema");

      //setup the shared authentication information
      authParams = common.getAuthenticationParams(initParameters->getDocument());

//However, since this driver is just a skeleton...
//
//Create a return document that says we can't do it. Note that this

//causes Identity Manager to display a warning in DSTrace that it is unable to
//read the application schema
      return
common.setReturnDocument(common.createStatusDocument(STATUS_LEVEL_ERROR,MSG_NO_SCHE
MA));
   } catch (ShimException e)
   {
      return
common.setReturnDocument(common.createStatusDocument(STATUS_LEVEL_FATAL,e.getMessag
e()));
   } catch (...)
   {
      //something bad happened...
      return
common.setReturnDocument(common.createStatusDocument(STATUS_LEVEL_FATAL,MSG_BAD));
   }
}
```

In the Identity Manager sample code, see the getSchema method in the DriverShimImpl.cpp file.

DriverShim destroy (C++ only)

Frees any resources allocated by the driver, including the driver object.

Syntax

C++

```
#include "NativeInterface.h"


void METHOD_CALL destroy (
   void);
```

Remarks

The Identity Manager engine calls this method for a C++ driver when the driver object is no longer required by the engine. The destroy method is called after the DriverShim shutdown method is called or after the getSchema method is called, depending on the mode. The destroy method will also be called after the DriverShim init method is called if the init method returns a fatal error in its return document.

The destroy method is responsible for freeing all resources used by the driver instance, including the driver instance itself. After the destroy method returns, no further calls or references to the driver instance are made.

C++ Sample Code

The following code from the C++ skeleton driver implements this method.

```
void METHOD_CALL CSkeletonDriver::destroy(void)


{
   common.tracer->trace("destroy");
   delete   this;
}
```

In the Identity Manager sample code, see the destroy method in the DriverShimImpl.cpp file.

2.6 Implementing the SubscriptionShim Interface

The SubscriptionShim interface consists of two methods:

- SubscriptionShim init—performs subscriber channel-specific initialization.
- SubscriptionShim execute—accepts commands from the Identity Manager engine and executes those commands on the application. The execute method is inherited from the XmlCommandProcessor interface.

When the Identity Manager engine initializes a Identity Manager driver, the engine adds the driver to its notification list for the Identity Vault events. When an Identity Vault event occurs, the Identity Manager engine converts the Identity Vault data to XML and sends it to the subscriber.

The subscriber needs to be configured for the Identity Vault filter. The Identity Manager engine uses the filter to register the Identity Manager driver for the appropriate Identity Vault events and then filters the data to the appropriate objects and attributes.

The Identity Manager driver also needs to be configured for mapping, creation, matching, placement, and optionally, event transformation rules. The Identity Manager engine converts the Identity Vault data to XML and applies these rules before sending the data to the subscriber. These rules allow the system administrator to determine what data and how the data is shared between the two. For more information, see Introduction to the Rules and Filters.

The subscriber requires four main routines, three of which are driver-specific:

- Driver-specific: the initialization process.
- Driver-specific: the process for converting the XML formatted data into the external database's native format.
- Driver-specific: the process that sends the data to the external database. You must use the application's programming interface for this process.
- Generic to all drivers: the query callback process that requests more information from the Identity Vault before performing an update to the external application.

The SubscriptionShim interface inherits from the XmlCommandProcessor interface.

For Javadoc, see SubscriptionShim.

SubscriptionShim init

Initializes the subcriber of the Identity Manager driver.

Syntax

Java

```
public XmlDocument init (
   XmlDocument   initParameters)
```

C++

```
#include "NativeInterface.h"


XmlDocument * METHOD_CALL init (
   XmlDocument   *initParameters);
```

Parameters

initParameters

(IN) Points to an XML document that contains the initialization data for

Return Values

Returns an XML document containing a status report on the initialization process. The status can be "success", "warning", "error" or "fatal". If "fatal" is returned, the driver start is aborted.

Remarks

The Identity Manager engine calls the init method to allow the Subscriber object to perform any channel-specific initialization necessary before beginning the execution of commands.

The initParameters argument contains an XML document with initialization data such as authentication information, driver-specific subscriber parameters, and the subscriber channel filter.

When the Identity Manager engine calls the SubscriptionShim init method, it sends an XML document similar to the following.

```
<nds dtdversion="1.0" ndsversion="8.5">
   <source>
      <product version="1.0">DirXML</product>
      <contact>Novell, Inc.</contact>
   </source>
   <input>
      <init-params src-dn="\PERIN-TAO\novell\Driver Set\Java Skeleton Driver\Subscriber">
         <authentication-info>
            <server>server.app:400</server>
            <user>User1</user>
            <password><!-- content suppressed --></password>
         </authentication-info>
         <driver-filter>
            <allow-class class-name="User">
               <allow-attr attr-name="Given Name"/>
               <allow-attr attr-name="Surname"/>
               <allow-attr attr-name="Telephone Number"/>
            </allow-class>
         </driver-filter>
         <subscriber-options>
      <sub-1 display-name="Sample Subscriber option">String for Subscriber</sub-1>
   </subscriber-options>
         <subscriber-state>
            <current-association>6</current-association>
         </subscriber-state>
      </init-params>
```

```
   </input>
</nds>
```

Most of the information in the initialization document corresponds to information configured using iManager/Designer in the DirXML-Driver object properties dialog.

The src-dn attribute on the <init-params> element is the distinguished name of the DirXML-Subscriber object that corresponds to the driver's Subscriber object. The DirXML-Subscriber object is the Identity Vault object that contains the Subscriber channel filter and references to the Subscriber channel rules.

The content of the <authentication-info> element corresponds to the driver authentication parameters found in the DirXML-Driver properties dialog. The content of the <password> element is suppressed because the trace facility suppresses sensitive data (as defined by Identity Manager). The actual password value is available to the driver.

The content of the <subscriber-options> element corresponds to driver-specific options specified in the DirXML-Driver properties dialog. Driver-specific options are specified using an XML file that describes the options. See DriverShim init for an example XML file used with the skeleton driver. For information about the possible elements in this document, see <subscriber-options>.

The <subscriber-state> element contains state information that the skeleton driver writes at driver shutdown. (For more information , see DriverShim shutdown.

The <driver-filter> element contains the subscriber filter. This filter is a list of the classes and attributes for which the subscriber receives events. For information about the possible elements in the filter, see <driver-filter>.

Sample Code

Java Sample Code

The following code from the skeleton driver retrieves the subscriber parameters, checks the document for state information and configuration parameters, and then returns a status document.

```
public XmlDocument init(XmlDocument initParameters )
{
   try
   {
      tracer.trace("init");
      //get any non-authentication options from
      //the init document
      params = getShimParams(initParameters.getDocument(),
"subscriber",SUBSCRIBER_PARAMS);


      //Get any state that may have been passed in.


      //The skeleton driver fakes associations to DirXML so
      //that it appears more like a real driver
      //see addHandler()
      int assocState = params.getIntParam("current-association");
      if (assocState != -1)
```

```
        {
            //setup our fake association for handling adds
            currentAssociation = assocState;
        }
        //perform any other initialization that might be
        //required.

        return createSuccessDocument();
    } catch (Throwable t)
    {
        //something bad happened...
        return   createStatusDocument(STATUS_FATAL, t.getMessage());
    }
}
```

In the Identity Manager sample code, see the init method in the SolutionSubscriptionShim.java file and the SubscriptionShimImpl.java file.

C++ Sample Code

The following code from the skeleton driver sends a DSTrace message, retrieves the subscriber parameters, and returns a status document.

```
XmlDocument * METHOD_CALL SkeletonSubscriber::init(
XmlDocument * initParameters)


{
    try
    {
        common.tracer->trace("init");


        //get any non-authentication options from the init document
        params = common.getShimParams(initParameters-
>getDocument(),TEXT_SUBSCRIBER,SUBSCRIBER_PARAMS);


        //Get any state that may have been passed in.


        //The skeleton driver fakes associations to DirXML so that it appears
        //more like a real driver. See addHandler()
        int assocState = params->getIntParam(TEXT_CURRENT_ASSOCIATION);
        if (assocState != -1)
        {
```

```
        //setup our fake association for handling adds
        currentAssociation = assocState;
     }
     //perform any other initialization that might be required.

     return common.setReturnDocument(common.createSuccessDocument());
  } catch (ShimException e)
  {
     return
common.setReturnDocument(common.createStatusDocument(STATUS_LEVEL_FATAL,e.getMessag
e()));
  } catch (...)
  {
     //something bad happened...
     return
common.setReturnDocument(common.createStatusDocument(STATUS_LEVEL_FATAL,MSG_BAD));
  }
}
```

In the Identity Manager sample code, see the init method in the SubcriptionShimImpl.cpp file.

SubscriptionShim execute

Sends the Identity Vault information to the Identity Manager driver for the external application.

Syntax

Java

```
public XmlDocument execute (
   XmlDocument          doc,
   XmlQueryProcessor    query)
```

C++

```
#include "NativeInterface.h"

XmlDocument * METHOD_CALL execute (
   XmlDocument          *document,
   XmlQueryProcessor    *query);
```

Parameters

document

(IN) Points to an XML document.

query

(IN) Points to a callback method that the Identity Manager driver can call if a command in the document does not contain enough information.

Remarks

The Identity Manager engine uses the execute method to send commands to the driver, and the driver executes them in the application. For example, the Identity Manager engine could send an XML document that instructs the driver to create a new user in the application. Other examples include commands to modify an attribute on an application object or to delete an application object. In addition, the Identity Manager engine can send a query element as part of a document passed to the execute method. The query instructs the driver to query the application for the data contained in the query.

The doc argument is the command document which is an XML document containing one or more commands that must be executed by the driver. This may be any mix of acceptable commands, depending on the rules, and may include any of the following: <add>, <modify>, <delete>, <rename>, <move>, and <query>.

IMPORTANT: Although the Identity Manager engine usually sends just one command per document, the subscriber must be prepared to handle any number of commands in a single document.

The query argument is an interface that allows the subscriber to query the Identity Vault for any additional information it may require to complete a command. The XmlQueryProcessor interface has a method named query which accepts an XDS document containing one or more queries.

The return from the execute method is an XML document containing the status of the command processing. For the format of all possible input and output documents, see Section 7.0, DTD Commands and Events.

One or more of the commands in the command document might fail. In order to allow the driver to return multiple <status> elements, each corresponding to a single command, both the command elements and the <status> elements have an attribute named event-id. The event-id value is unique in the document and is used to tell Identity Manager which command element the <status> element refers to. If a <status> element has no event-id attribute, the <status> element is assumed to apply to all commands that were in the command document.

The class names and attribute names contained in the command document will be the application's names, if a mapping rule is in place for the driver. The examples from the skeleton driver in this section are from a skeleton driver installation with no mapping rule, so the names in the example documents are the Identity Vault names.

Add Command

The following sample command document instructs the skeleton driver to add a user.

```
<nds dtdversion="1.0" ndsversion="8.5">
   <source>
      <product version="1.0">DirXML</product>
      <contact>Novell, Inc.</contact>
   </source>
```

```
   <input>
      <add class-name="User" event-id="0" src-dn="\PERIN-TAO\novell\John" src-
entry-id="35868">
         <add-attr attr-name="Surname">
            <value timestamp="965252204#5" type="string">Doe</value>
         </add-attr>
         <add-attr attr-name="Telephone Number">
            <value timestamp="965252229#6" type="teleNumber">(801) 555-5555</value>
         </add-attr>
         <add-attr attr-name="Given Name">
            <value timestamp="965252229#1" type="string">Jonathan</value>
         </add-attr>
      </add>
   </input>
</nds>
```

The above document tells the skeleton driver to add a user with three attributes. The response to an add command is a status or output document containing either an error status or an <add-association> informing Identity Manager the object was successfully added to the application and informing Identity Manager of the application value used for associating the Identity Vault and the application objects.

The skeleton driver does not actually support an application, so it cannot add a user, but the skeleton subscriber does have code that returns a fake association value telling Identity Manager that it did add the user. This is so that the skeleton driver can also illustrate how to receive modify and delete commands. The following example is a return document for the add example.

```
<nds dtdversion="1.0" ndsversion="8.5">
   <output>
      <add-association dest-dn="\PERIN-TAO\novell\John" event-id="0">7</add-
association>
   </output>
</nds>
```

The above document causes the Identity Manager engine to write a string ("7") that associates the Identity Vault object with the application object. All further commands relating to the application object will contain this association value so that the driver can reference the object in the application.

For more information about the possible elements in an add operation, see <add>.)

Modify Command

The following command modifies the Identity Vault object represented by the <add> command above. It contains a command to change the Given Name attribute.

```
<nds dtdversion="1.0" ndsversion="8.5">
   <source>
      <product version="1.0">DirXML</product>
      <contact>Novell, Inc.</contact>
   </source>
   <input>
```

```
      <modify class-name="User" event-id="0" src-dn="\PERIN-TAO\novell\John" src-
entry-id="35868" timestamp="965252836#6">
         <association state="associated">7</association>
         <modify-attr attr-name="Given Name">
            <remove-value>
               <value timestamp="965252836#6" type="string">Jonathan
                    </value>
            </remove-value>
            <add-value>
               <value timestamp="965252836#6" type="string">Johnny
                    </value>
            </add-value>
         </modify-attr>
      </modify>
   </input>
</nds>
```

The above document instructs the skeleton driver to modify the user with the unique application key of "7" such that the Given Name attribute value changes from "Jonathan" to "Johnny". The skeleton driver pretends the operation succeeds and returns the following status document:

```
<nds dtdversion="1.0" ndsversion="8.5">
   <output>
      <status event-id="0" level="success"/>
   </output>
</nds>
```

For more information about the possible elements in a modify operation, see <modify>.

Query Command

The command document passed to the execute method may also contain a query. A query is typically issued to the driver due to a matching rule. The Identity Manager engine sends a query when the following conditions occur:

- An event occurs for an Identity Vault object that matches the subscriber filter for the driver
- The Identity Vault object has not yet been associated with an object in the application
- A matching rule is present on the subscriber channel

The Identity Manager engine uses the query to locate a match in the application according to the criteria present in the matching rule.

The following command document, containing a query, was sent to the skeleton driver previous to the <add> command above. The query was generated because a matching rule was in place on the subscriber channel that instructed Identity Manager to attempt to find an object in the application with the same value for the "Surname" and "Telephone Number" attributes.

```
<nds dtdversion="1.0" ndsversion="8.5">
   <source>
      <product version="1.0">DirXML</product>
```

```
      <contact>Novell, Inc.</contact>
   </source>
   <input>
      <query class-name="User" event-id="0">
         <search-class class-name="User"/>
         <search-attr attr-name="Surname">
            <value timestamp="965252204#5" type="string">Doe</value>
         </search-attr>
         <search-attr attr-name="Telephone Number">
            <value timestamp="965252229#6" type="teleNumber">
                    (801) 555-5555</value>
         </search-attr>
         <read-attr/>
      </query>
   </input>
</nds>
```

Because the skeleton driver has no application, it is unable to find a match, and returns the following document.

```
<nds dtdversion="1.0" ndsversion="8.5">
   <output>
      <status event-id="0" level="success"/>
   </output>
</nds>
```

The absence of an <instance> element in the return document indicates no matching objects were found. The <status> element in the return document is success because the query command was successfully executed.

The following examples illustrate a query that returns a matched object. These examples were generated by the VRTest driver.

The Identity Manager engine sends the following query:

```
<nds dtdversion="1.0" ndsversion="8.5">
   <source>
      <product version="1.0">DirXML</product>
      <contact>Novell, Inc.</contact>
   </source>
   <input>
      <query class-name="User Object" event-id="0">
         <search-class class-name="User Object"/>
         <search-attr attr-name="name">
            <value type="string">Jane</value>
         </search-attr>
         <read-attr/>
```

```
        </query>
    </input>
</nds>
```

The VRTest driver returns the following response:

```
<nds dtdversion="1.0" ndsversion="8.5">
    <output>
        <instance class-name="User Object" src-dn="\novell\Jane">
            <association>25</association>
        </instance>
    </output>
</nds>
```

The above query instructed the VRTest driver to query the VRTest application for an object of class "User Object" with an attribute named "name" with the value "Jane". The empty <read-attr/> element instructs the driver that Identity Manager does not require any attributes to be read.

The response from the driver indicates that a single match was found, and that the unique application key for the object is "25". If more than one matching object had been found, multiple <instance> elements would appear in the result document.

For more information about the possible elements in a query operation, see <query>.

Sample Code

Java Sample Code

The following code from the skeleton driver code sends a message to DSTrace, searches the document for elements, dispatches them to command handlers, and returns a result document. It also shows how to handle the most common error conditions.

```
public XmlDocument execute( XmlDocument doc, XmlQueryProcessor query)
    {
        int   retryCount = 2;
        tracer.trace("execute");
        try
        {
            //setup the return document for use by command handlers
            outputElement = createOutputDocument();

            //try and connect with our mythical app
            while (retryCount-- > 0)
            {
                try
                {
                    connect();
```

```
               Document document = doc.getDocument();

               //find the <input> element
               Element input =
(Element)document.getElementsByTagName("input") .item(0);

               //iterate through the children, dispatching commands
               Node childNode = input.getFirstChild();
               while (childNode != null)
               {

          //only elements are interesting...ignore any nterspersed
                  //text, comments, etc.
                  if (childNode.getNodeType() == Node.ELEMENT_NODE)
                  {
                     dispatch((Element)childNode);
                  }
                  childNode = childNode.getNextSibling();
               }

               //return the result of whatever we were told to do
             return new XmlDocument(outputElement.getOwnerDocument());
            } catch (java.io.IOException e)
            {
               if (retryCount <= 0)
               {
                  //done trying
                  throw   e;
               }
            }
         }
         //if we fall through here, we failed to connect
         throw new java.io.IOException("failed to connect");
      } catch (java.io.IOException e)
      {
         //somehow failed in talking to app, tell DirXML to retry later
         return   createStatusDocument(STATUS_RETRY,e.toString());
      } catch (Throwable t)
      {
```

```
        //something bad happened...
        return   createStatusDocument(STATUS_ERROR,t.getMessage());
    }
}
```

In the Identity Manager sample code, see the execute method in the SolutionSubscriptionShim.java file and the SubscriptionShimImpl.java file. For sample code that processes the commands, see the dispatch method in the SolutionSubscriptionShim.java file and the handler methods (addHandler, modifyHandler, deleteHandler, renameHandler, moveHandler, and queryHandler) in the SkeletonSubscriptionShim.java file.

C++ Sample Code

The following code from the skeleton driver sends a message to DSTrace, searches the document for elements, dispatches them to command handlers, and returns a result document. It also shows how to handle the most common error conditions.

```
XmlDocument * METHOD_CALL SkeletonSubscriber::execute(
   XmlDocument * doc, XmlQueryProcessor * queryInterface)

{
   int   retryCount = 2;
   common.tracer->trace("execute");
   try
   {
      //setup the return document for use by command handlers
      outputElement = NdsDtd_newOutputDocument();

      //try and connect with our mythical app
      while (retryCount-- > 0)
      {
         try
         {
            //connect may throw a ConnectException
            connect();
            Document * document = doc->getDocument();

            //find the <input> element
            Element * input = (Element *)document->getElementsByTagName
           (common.ndsDtd->TAG_INPUT)->item(0);

            //iterate through the children, dispatching commands
            Node * childNode = input->getFirstChild();
            while (childNode != 0)
            {
```

```
            //only elements are interesting...ignore any interspersed
            //text, comments, etc.
            if (childNode->getNodeType() == Node::ELEMENT_NODE)
            {
               dispatch((Element *)childNode);
            }
            childNode = childNode->getNextSibling();
         }
         //return the result of whatever we were told to do
         return common.setReturnDocument(outputElement
          ->getOwnerDocument());
      } catch (ConnectException e)
      {
         if (retryCount <= 0)
         {
            //done trying
            throw   e;
         }
      }
   }
   //if we fall through here, we failed to connect,
   //so get the enclosing handler
   throw   ConnectException(MSG_CONNECT_FAILURE);
} catch (ConnectException e)
{
   //somehow failed in talking to app, tell DirXML to retry later...
   //"retry" status means the eDirectory event will not be discarded
  //and that DirXML will resubmit it later (default is 30 seconds
  // later). This status should be used for errors related to
 //connection problems since the event won't be lost. For errors
 // such  as application errors due to data, etc., use error since
 //it will cause DirXML to discard the event.
   return  common.setReturnDocument(common.createStatusDocument
(STATUS_LEVEL_RETRY,e.getMessage()));
} catch (ShimException e)
{
   return   common.setReturnDocument(common.createStatusDocument
(STATUS_LEVEL_ERROR,e.getMessage()));
} catch (...)
{
```

```
        //something bad happened...
        return
common.setReturnDocument(common.createStatusDocument( STATUS_LEVEL_FATAL,MSG_BAD));
    }
}
```

In the Identity Manager sample code, see the execute method in the SubscriptionShimImpl.cpp file. For sample code that processes the commands, see the dispatch method in the SubscriptionShimImpl.cpp file and the handler methods (addHandler, modifyHandler, deleteHandler, renameHandler, moveHandler, and queryHandler) in the SkeletonSubscriber.cpp file.

2.7 Implementing the PublicationShim Interface

The PublicationShim interface consists of two methods:

- PublicationShim init—performs publisher channel-specific initialization
- PublicationShim start—monitors the application and publishes application changes to the Identity Manager engine

Each of these methods is described in detail in the following sections.

In addition, the publisher must provide a query callback function for the Identity Manager engine. The publisher usually implements the XmlQueryProcessor interface.

For Javadoc, see PublicationShim.

PublicationShim init

Initializes the publisher of the Identity Manager driver.

Syntax

Java

```
public XmlDocument init (
   XmlDocument   initParameters)
```

C++

```
#include "NativeInterface.h"


XmlDocument * METHOD_CALL init (
   XmlDocument   *initParameters);
```

Parameters

initParameters

(IN) Points to an XML document that contains the configuration information for the publisher.

Return Values

Returns an XML document containing a status report on the initialization operation. The status can be "success", "warning", "error", or "fatal". If "fatal" is returned, the driver start is aborted.

Remarks

The Identity Manager engine calls the init method to allow the publisher object to perform any channel-specific initialization necessary before monitoring the application and publishing changes to Identity Manager.

The initParameters argument contains an XML document with initialization data such as authentication information, driver-specific publisher parameters, and the publisher channel filter.

When the Identity Manager engine calls the PublicationShim init method, it sends an XML document similar to the following.

```
<nds dtdversion="1.0" ndsversion="8.5">
   <source>
      <product version="1.0">DirXML</product>
      <contact>Novell, Inc.</contact>
   </source>
   <input>
      <init-params src-dn="\PERIN-TAO\novell\Driver Set\Java Skeleton
               Driver\Publisher">
         <authentication-info>
            <server>server.app:400</server>
            <user>User1</user>
            <password><!-- content suppressed --></password>
         </authentication-info>
         <driver-filter>
            <allow-class class-name="User">
               <allow-attr attr-name="Given Name"/>
               <allow-attr attr-name="Surname"/>
            </allow-class>
         </driver-filter>
         <publisher-options>
      <pub-1 display-name="Sample Publisher option">String for
```

```
                    Publisherr</pub-1>
            <polling-interval display-name="Polling interval in
                    seconds">4</polling-interval>
    </publisher-options>
        </init-params>
    </input>
</nds>
```

Most of the information in the initialization document corresponds to information configured using iManager/Designer in the DirXML-Driver object properties dialog.

The src-dn attribute on the <init-params> element is the distinguished name of the DirXML-Publisher object that corresponds to the driver's publisher object. The DirXML-Publisher object is the Identity Vault object that contains the publisher channel filter and references to the publisher channel rules.

The content of the <authentication-info> element corresponds to the driver authentication parameters found in the DirXML-Driver properties dialog. The content of the <password> element is suppressed because the trace facility suppresses sensitive data (as defined by DirXML). The actual password value is available to the driver.

The content of the <publisher-options> element corresponds to driver-specific options specified in the DirXML-Driver properties dialog. Driver-specific options are specified using an XML file that describes the options. See DriverShim init for an example XML file used with the skeleton driver. For information about the possible elements in this document, see <publisher-options>.

The <driver-filter> element contains the publisher filter. This filter is a list of the classes and attributes for which the publisher sends events to the DirXML engine. For information about the possible elements in the filter, see <driver-filter>.

Sample Code

Java Sample Code

The following code from the skeleton driver retrieves the document, gets the publisher configuration parameters, sets the polling interval, and constructs a filter.

```
public XmlDocument init(XmlDocument initParameters )
{
      tracer.trace("init");

      //get the driver filter for the publication shim to use
      //for filtering application events
      Document initDoc = initParameters.getDocument();

      //get any non-authentication options from the init document
      ShimParams params = getShimParams(initDoc,"publisher",PUBLISHER_PARAMS);

      //get any the polling interval that may have been passed in
      int pi = params.getIntParam("polling-interval");
      if (pi != -1)
```

```
        {
           //change our default polling interval to whatever was setup
           //using iManager
           pollingInterval = pi;
        }

        //setup a filter for use in start()
         //NOTE: the skeleton publisher doesn't actually make use of the
         // filter, but this code is here to illustrate how to create the
         // filter based on the init parameters
        NodeList filterList = initDoc.getElementsByTagName("driver-
                 filter");
        int      i = 0;
        Element   filterElement;
        while ((filterElement = (Element)filterList.item(i++)) != null)
        {
           String   type = filterElement.getAttribute("type");
           if (type.length() == 0 || type.equals("publisher"))
           {
              filter = new DriverFilter(filterElement);
              break;
           }
        }
        if (filter == null)
        {
           //if weren't able to setup a filter, setup a null
           //filter so we don't have to check for filter != 0 everywhere
           filter = new DriverFilter();
        }
        return createSuccessDocument();
     }
```

In the Identity Manager sample code, see the init method in the SolutionPublicationShim.java file and the PublicationShimImpl.java file.

C++ Sample Code

The following code from the C++ skeleton driver retrieves the document, gets the publisher configuration parameters, sets a polling interval, and constructs a filter.

```
XmlDocument * METHOD_CALL SkeletonPublisher::init(
   XmlDocument * initParameters)
```

```
{
   try
   {
      common.tracer->trace("init");

      //construct a driver filter for the publication shim to use for filtering
      //application events In an actual driver, the publisher would use
      //the filter to filter events from the application
      //to avoid publishing unnecessary events to DirXML.
      Document * initDoc = initParameters->getDocument();

      //get any non-authentication options from the init document
      CommonImpl::ShimParams * params = common.getShimParams(initDoc,
TEXT_PUBLISHER,PUBLISHER_PARAMS);

      //get any the polling interval that may have been passed in
      int pi = params->getIntParam(TEXT_POLLING_INTERVAL);
      if (pi != -1)
      {
         //change our default polling interval to whatever was
         //set up using iManager
         pollingInterval = pi;
      }
      //setup a filter for use in start()
      //NOTE: the skeleton publisher doesn't actually make use of the
      // filter, but this code is here to illustrate how to create the
      // filter based on the init parameters
      Element   * filterElement = (Element *)initDoc->getElementsByTagName
(common.ndsDtd->TAG_DRIVER_FILTER)->item(0);
      if (filterElement != 0)
      {
         filter = DriverFilter_new(filterElement);
      } else
      {
         //if weren't able to setup a filter, setup a null
         //filter so we don't have to check for filter != 0 everywhere
         filter = DriverFilter_new(0);
      }
      return common.setReturnDocument(common.createSuccessDocument());
```

```
    } catch (ShimException e)

    {

        return   common.setReturnDocument(common.createStatusDocument
(STATUS_LEVEL_FATAL,e.getMessage()));

    } catch (...)

    {

        //something bad happened...

        return
common.setReturnDocument(common.createStatusDocument( STATUS_LEVEL_FATAL,MSG_BAD));

    }

}
```

In the Identity Manager sample code, see the init method in the PublicationShimImpl.cpp file.

PublicationShim start

Publishes data from the external application to the Identity Vault.

Syntax

Java

```
public XmlDocument start (
   XmlCommandProcessor   execute)
```

C++

```
#include "NativeInterface.h"


XmlDocument * METHOD_CALL start (
   XmlCommandProcessor   *execute);
```

Parameters

execute

(IN) Points to the callback method that starts the publisher which runs on a separate thread and loops listening for data from the external database.

Remarks

The Identity Manager engine calls the start method to allow the driver to monitor the application and publish application events (changes)

to the Identity Manager engine. The publisher should not return from the start method until instructed to do so from the DriverShim shutdown method. An exception to this rule is if a fatal error occurs. If the publisher returns from the start method before the DriverShim shutdown method is called, the Identity Manager engine shuts down the driver.

The execute argument is an interface through which the publisher submits event documents to the Identity Manager engine. The XmlCommandProcessor interface has a single method named execute with the following syntax.

Java Syntax

```
XmlDocument execute(
   XmlDocument         doc,
   XmlQueryProcessor   query);
```

C++ Syntax

```
XmlDocument * METHOD_CALL execute(
    XmlDocument         *doc,
    XmlQueryProcessor   *query);
```

The publisher constructs an XML document describing the application event and submits it through the execute method, along with an interface that the Identity Manager engine can use to query back to the publisher if the engine determines that it requires additional data. The Identity Manager engine returns an XML document containing the status of the event processing.

The skeleton driver does not publish any information to Identity Manager other than status documents should an error occur. However the VRTest driver does publish events, and therefore, examples from the VRTest driver will be used in this section to illustrate the publication of events. For the format of all possible input and output documents, see Section 7.0, DTD Commands and Events.

Add Event

The VRTest driver published the following document to the Identity Manager engine in response to a VRTest application change.

```
<nds dtdversion="1.0" ndsversion="8.5">
   <input>
      <add class-name="User Object" src-dn="\novell\JJones">
         <association>27</association>
         <add-attr attr-name="Family Name">
            <value>James</value>
         </add-attr>
         <add-attr attr-name="First Name">
            <value>Jones</value>
         </add-attr>
         <add-attr attr-name="Telephone">
            <value>(801) 555-1234</value>
         </add-attr>
      </add>
   </input>
```

```
</nds>
```

The above document indicates that an object of application class "User Object" with three attributes was added to the VRTest application.

The Identity Manager engine responded with the following document.

```
<nds dtdversion="1.0" ndsversion="8.5">
   <source>
      <product version="1.0">DirXML</product>
      <contact>Novell, Inc.</contact>
   </source>
   <output>
      <status event-id="" level="success"></status>
   </output>
</nds>
```

The response document indicates that the event was processed by the engine according to any rules in place and that no errors occurred.

All events submitted to the Identity Manager engine must have an <association> value. For more information on the elements that must be included in an add event, see <add>.

Modify Event

When the object corresponding to the example above was later modified, the VRTest publisher sent the following document to the Identity Manager engine.

```
<nds dtdversion="1.0" ndsversion="8.5">
   <input>
      <modify class-name="User Object" src-dn="\novell\JJones">
         <association>27</association>
         <modify-attr attr-name="Telephone">
            <add-value>
               <value>(801) 555-1235</value>
            </add-value>
            <add-value>
               <value>(801) 555-1236</value>
            </add-value>
         </modify-attr>
      </modify>
   </input>
</nds>
```

The above document informs Identity Manager that the application object associated with the Identity Vault object containing the association value "27" had two values added to its "Telephone" attribute.

For more information on the elements that must be included in a modify event, see <modify>.

Delete Event

If the application object is later deleted, the publisher submits the following event document.

```
<nds dtdversion="1.0" ndsversion="8.5">
   <input>
      <delete class-name="User" src-dn="\novell\JJones">
         <association>27</association>
      </delete>
   </input>
</nds>
```

For more information on the elements that must be included in a delete event, see <delete>.

Sample Code

Java Sample Code

The following code from the Java skeleton driver sends messages to DSTrace, checks to see if it has received a stop request, and returns an XMLDocument.

```
public XmlDocument start( XmlCommandProcessor execute)
   {
      //NOTE: this implements a polling method of communication with the
      //application. This may not be appropriate if the application supports
      //an event notification system
      tracer.trace("start");

      //loop until we're told to shutdown (or some fatal error occurs)
      while(!shutdown)
      {
         // skeleton implementation just wakes up every so often to
         // see if it needs to shutdown and return.
         try
         {
            tracer.trace("polling...");
            //In a real driver, we'd do whatever was necessary to ask the
            //application what changed and build an input document to publish
            //the change events to DirXML.

            //wait for subscriber channel thread to wake us up, or for polling
```

```
            //interval to expire.
             //NOTE: the use of the semaphore is highly recommended. It prevents
            //a long polling interval from interfering with the orderly
            //shutdown of the driver.
             synchronized(semaphore)
             {
                //our pollingInterval value is in seconds,
                //The Object.wait() takes milliseconds
                semaphore.wait(pollingInterval * 1000);
             }
          }
          catch(InterruptedException ie)
          {
          }
       }
       tracer.trace("stopping");
       return createSuccessDocument();
    }
```

In the Identity Manager sample code, see the start method in the SolutionPublicationShim.java file and the PublicationShimImpl.java file.

C++ Sample Code

The following code from the skeleton driver sends messages to DSTrace, checks to see if it has received a stop request, and returns an XmlDocument.

```
XmlDocument * METHOD_CALL SkeletonPublisher::start(
   XmlCommandProcessor * ndsExecute)


{
   try
   {
      //NOTE: this implements a polling method of communication with the
      //application. This may not be appropriate if the application
      //supports an event notification system
      common.tracer->trace("start");

      //loop until we're told to shutdown (or some fatal error occurs)
      while(!shutdown && !aborted)
      {
         try
```

```
        {
            // skeleton implementation just wakes up every so often to
            // see if it needs to shutdown and return.
            common.tracer->trace("polling...");

            //In a real driver, we'd do whatever was necessary to ask
            //the application what changed and build an input document
            //to publish the change events to DirXML.

            //wait for subscriber channel thread to wake us up, or for
            //polling interval to expire.
             //NOTE: the use of the semaphore is highly recommended. It
            //prevents a long polling interval from interfering with the
           //orderly shutdown of the driver.
            waitSemaphore(semaphore,pollingInterval * 1000);
        } catch (ShimException e)
        {
            //some sort of error happened...publish the error to DirXML
            //but don't quit (definitely don't quit if all that happened
            //is we lost communication with the app...
            //we should simply try and reestablish it later)
            Element   * input = NdsDtd_newInputDocument();
            NdsDtd_addStatus(input,STATUS_LEVEL_ERROR,e.getMessage(),0);
            XmlDocument   * pubDoc = XmlDocument_newFromDOM(input-
>getOwnerDocument());
            ndsExecute->execute(pubDoc,this);
            XmlDocument_destroy(pubDoc);

            //NOTE that we must destroy the XmlDocument and the DOM
            //document separately. The XmlDocument doesn't take
            //ownership of the DOM document
            input->getOwnerDocument()->destroy();

            //loop around to try again
        }
    }
    if (!aborted)
    {
        common.tracer->trace("stopping");
```

```
         common.setReturnDocument(common.createSuccessDocument());
      } else
      {
         common.tracer->trace("aborting");

common.setReturnDocument(common.createStatusDocument(STATUS_LEVEL_FATAL,MSG_ABORTED
));
      }
      return common.getReturnDocument();
   } catch (...)
   {
      //something bad happened...
      return   common.setReturnDocument(common.createStatusDocument
(STATUS_LEVEL_FATAL,MSG_BAD));
   }
}
```

In the Identity Manager sample code, see the start method in the PublicationShimImpl.cpp file.

2.8 Implementing the XmlQueryProcessor Interface

The XmlQueryProcessor interface must be implemented by the driver because the XmlCommandProcessor execute method requires as its second parameter an object that implements the XmlQueryProcessor interface. The XmlQueryProcessor interface is used by the Identity Manager engine to query the publisher when additional data is required in the course of processing an event.

An object implementing the XmlQueryProcessor interface is also passed to the Subscriber's execute method so that the subscriber can query back into the Identity Vault, if necessary.

Most drivers implement the XmlQueryProcessor interface in the publisher object, although this is not required.The implementation of the XmlQueryProcessor interface and the query handling for the subscriber execute method can usually share most code.

The XmlQueryProcessor interface consists of a single method, query, which performs the query described in the XML document argument.

For Javadoc, see XmlQueryProcessor.

query

Executes an XML-encoded query and returns an XML-encoded results.

Syntax

C++

```
#include "NativeInterface.h"
```

```
XmlDocument * METHOD_CALL query (
   XmlDocument   *doc);
```

Java

```
public XmlDocument query (
   XmlDocument   doc)
```

Parameters

doc

(IN) Points to a document containing an XML encoded command.

Remarks

The doc argument contains an XML document with the formulated query. The return document contains either the result of the query or a status element in case of error.

For information on the elements that can be contained in the query document, see <query>.

Sample Code

Java Example Code

The following code from the SkeletonPublication.java file shows how to set up the query method.

```
public XmlDocument query(XmlDocument doc)
{
   tracer.trace("query");
   //since this is a skeleton, and there is nothing to query,
   // just return an empty output document with a success
   // status. The absence of an <instance> element
   //tells DirXML that nothing matched the query.
   return createSuccessDocument();
}
```

In the Identity Manager sample code, see the query method in the SolutionQueryHandler.java file and the PublicationShimImpl.java file.

C++ Example Code

The following code from the SkeletonPublisher.cpp file shows how to set up the query method.

```
XmlDocument * METHOD_CALL
SkeletonPublisher::query(
   XmlDocument * document)
{
   common.tracer->trace("query");
   //since this is a skeleton, and there is nothing to query,
   // just return an empty output document with a success
   // status. The absence of an <instance> element tells
   // DirXML that nothing matched the query.
   return common.setReturnDocument(common.createSuccessDocument());
}
```

In the Identity Manager sample code, see the query method in the PublicationShimImpl.cpp file.

2.9 Dealing with XML Documents

All communication between the Identity Manager engine and the application driver takes the form of XML documents (with the exception of DriverShim getSubscriptionShim and getPublicationShim methods).

Identity Manager SDK provides several methods of dealing with XML documents:

- Document Object Mode (DOM)
- Simple API for XML (SAX)
- XDS Libraries
- Serialized XML (byte sequence)

Typically your driver will deal with XML documents using the DOM or XDS libraries, but there may be drivers for which it is more convenient to use SAX or even a serialized document.

All XML document arguments and returns for interface methods use an abstraction named XmlDocument to encapsulate the underlying XML representation. This allows the Identity Manager engine to supply the document in one form and the driver to consume the document in another form. The converse is also true. For example, the driver could receive a serialized XML document from the application and supply it in that form to the XmlDocument object. The Identity Manager engine would then consume it as a DOM tree.

2.9.1 Java Sample Code

The XmlDocument abstraction has the following methods for obtaining the underlying XML document in the desired form.

- org.w3c.Document getDocument();
- byte[] getDocumentBytes(java.lang.String encoding);
- org.xml.sax.Parser getDocumentSAX();
- org.xml.sax.InputSource getDocumentInputSource();
- java.lang.String getDocumentString();

The XmlDocument abstraction is implemented as a class in Java (com.novell.nds.dirxml.driver.XmlDocument). The following examples illustrate the most common ways of obtaining the document representation and setting the document representation. There are additional methods. See the online javadocs.

DOM

The following code illustrates obtaining an XML document as a DOM tree from the XmlDocument object.

```
void example(XmlDocument doc)
{
   Document inputDocument = doc.getDocument();
}
```

The following code illustrates creating a DOM document and setting the XmlDocument object from the DOM document.

```
XmlDocument example()
{
   //create a DOM Document using the document factory
Document returnDoc = com.novell.xml.dom.DocumentFactory.newDocument();
   //create the <nds> root element
   Element nds = returnDoc.createElement("nds");
   returnDoc.appendChild(nds);
   //set the various xds attributes
   nds.setAttribute("ndsversion","8.5");
   nds.setAttribute("dtdversion","1.0");
   return   new XmlDocument(returnDoc);
}
```

SAX

The following sample illustrates obtaining the XML document as a series of SAX events from the XmlDocument object.

```
class Handler implements DocumentHandler
{
   ...
};
void example(XmlDocument doc)
{
Parser parser = doc.getDocumentSAX();
InputSource inputSource = doc.getDocumentInputSource();
parser.setDocumentHandler(new Handler());
parser.parse(inputSource);
}
```

The following sample illustrates setting the XmlDocument object from objects that implement the SAX Parser and SAX InputSource interfaces.

class MyParser implements Parser { ... } XmlDocument example() { MyParser parser = new MyParser(); InputSource inputSource = new InputSource("file.xml"); return new XmlDocument(parser,inputSource); }

Serialized XML

The following sample illustrates obtaining the XML document in a serialized form.

```
void example(XmlDocument doc)
{
   byte[] bytes = doc.getDocumentBytes("UTF-8");
}
```

The following sample illustrates setting the XmlDocument object from a serialized XML document contained in a byte array:

```
XmlDocument example(byte [] bytes)
{
   return new XmlDocument(bytes);
}
```

2.9.2 C++ Sample Code

The XmlDocument abstraction has the following C++ methods for obtaining the underlying XML document in the desired form.

```
   DOM::Document * METHOD_CALL getDocument();   SAX::Parser * METHOD_CALL
getDocumentSAX();   SAX::InputSource * METHOD_CALL getDocumentInputSource();
const unsigned char * METHOD_CALL getDocumentBytes(const
      unicode * encoding, int endian, int * length);
```

The XmlDocument abstraction is implemented as an interface in C++ (defined in NativeInterface.h). Methods are also defined in the following header files.

Table 2-2 XmlDocument header files

File	Description
dom.h	Defines interfaces and methods which are patterned after the Java implementation of the W3C DOM (Document Object Model) Level 1.
InterfaceFactory.h	Defines the factories and the destructors.
sax.h	Defines interfaces and methods for receiving information about XML documents. They are patterned after the Java implementation of version 1 of the Simple API for XML (SAX) event interface.
OutputStream.h	Defines an interface modeled on java.io.OutputStream for writing to a byte sink.
XMLWriter.h	Creates an XMLWriter.

The following examples illustrate the most common ways of obtaining the document representation and setting the document representation. There are additional methods. See the NativeInterface.h file.

DOM

The following code illustrates obtaining the XML document as a DOM tree from the XmlDocument interface:

```
XmlDocument *
SubscriptionShimImpl::execute(
XmlDocument * inputDoc
XmlQueryProcessor * query)
{
DOM::Document * document = inputDoc->getDocument();
    ...
}
```

The following illustrates creating an XmlDocument as a DOM tree. (The factory methods are defined in InterfaceFactory.h)

```
//create a document using DirXML factory method
DOM::Document * document = Document_new();
//create the document tree
DOM::Element * ndsElement = document->createElement(NDS_ELEMENT_NAME);
 ...
//create the XmlDocument instance
XmlDocument * returnDoc = XmlDocument_newFromDOM(document);
...
//sometime later, after returnDoc is no longer being used:
XmlDocument_destroy(returnDoc);
document->destroy();
```

SAX

The following code illustrates obtaining the XML document as a series of SAX events.

```
XmlDocument *
SubscriptionShimImpl::execute(
XmlDocument * inputDoc,
XmlQueryProcessor * query)
{
SAX::Parser * eventGenerator;
SAX::InputSource * inputSource;
eventGenerator = inputDoc->getDocumentSAX();
inputSource = inputDoc->getDocumentInputSource();
//my event handler is a class that implements the DocumentHandler interface
//declared in sax.h
```

```
eventGenerator->setDocumentHandler(myEventHandler);

//this call causes the SAX events to be sent to the document handler

eventGenerator->parse(inputSource);

...

}
```

The following code illustrates creating an XmlDocument using SAX events. For ease of illustration, the SAX event source is the Identity Manager supplied XML parser, using a file as input.

```
SAX::InputSource * inputSource = InputSource_new();

inputSource->setSystemId("tempfile.xml");

SAX::Parser * parser = Parser_new();

XmlDocument * xmlDoc = XmlDocument_newFromSAX(parser, inputSource);

...

//sometime later, after xmlDoc is no longer being used:

XmlDocument_destroy(xmlDoc);

Parser_destroy(parser);

InputSource_destroy(inputSource);
```

Serialized XML

The following code illustrates obtaining a serialized representation of the XML document.

```
XmlDocument *

SubscriptionShimImpl::execute(

XmlDocument * inputDoc

XmlQueryProcessor * query)

{

static const unicode ENCODING[] = {'U','T','F','-','8',0};

const unsigned char * bytes;

int length;

bytes = inputDoc->getDocumentBytes(ENCODING, LITTLE_ENDIAN, &length);

...

}
```

The following code illustrates creating an XmlDocument from a byte array.

```
static const unicode ENCODING[] = {'U','T','F','-','8',0};

//make a driver-specific call to get the serialized XML from somewhere

int length;

unsigned char * bytes = get_bytes_from_somewhere(&length);

//create the XmlDocument from the serialized XML

XmlDocument * xmlDoc =
XmlDocument_newFromBytes(bytes,length,ENCODING,LITTLE_ENDIAN);

...
```

```
//sometime later, when xmlDoc is no longer in use...
XmlDocument_destroy(xmlDoc);
release_bytes_from_somewhere(bytes);
```

2.10 Driver State

Identity Manager provides a mechanism for your driver to save state information between invocations. State information may be saved separately for the driver object, the subscriber object, and the publisher object. Any saved state information is passed to the object's init method as part of the initialization parameters document. The state information is stored in an attribute on the DirXML-Driver object corresponding to your driver.

It is important to note that, in general, it is inappropriate for a driver to store configuration information outside of the Identity Vault where it cannot be remotely configured using iManager or other remote configuration utilities.

Driver state is written by returning or publishing an <init-params> element containing one or more of the following elements: <driver-state>, <subscriber-state>, and <publisher-state>. When returning state to be written, the <init-params> element is a child of the <output> element in the XDS document. When publishing state, the <init-params> element is a child of the <input> element in the XDS document. State may be written at any time in a document returned to the Identity Manager engine and may be written at any time by publishing a document to the Identity Manager engine on the publisher channel.

The actual state information can be any information desired. The content of the <driver-state>, <subscriber-state>, and <publisher-state> elements is stored exactly as presented to the Identity Manager engine.

The following example XML document shows a sample document sent to the Identity Manager engine that writes publisher state.

```
<nds dtdversion="1.0" ndsversion="8.5">
   <input>
      <init-params>
         <publisher-state>
            <sync-up-to>965252784</sync-up-to>
            <last-object>
               <name>Test</name>
               <id>2349</id>
            </last-object>
         </publisher-state>
      </init-params>
   </input>
</nds>
```

The content of the <publisher-state> element is not defined by Identity Manager. The content may be any XML representation of the data that is convenient for your driver. The state for any or all of the driver, subscriber, and publisher objects can be written on the subscriber channel as an <output> document or publisher channel as an <input> document.

2.11 Driver Configuration

The Identity Manager engine, in conjunction with iManager, provides a mechanism for you to define driver-specific configuration parameters. This is accomplished with an XML file that describes the parameters. iManager uses this XML file to dynamically construct a simple user interface that administrators can use to configure your driver for a particular installation.

In general, your driver must be configurable from Novell adminstrator utilities such as Designer and iManager.

A sample XML configuration file from the Java skeleton driver appears below:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--  this file contains a sample driver options xml document;
      it is designed for use with the Java DirXML skeleton
      driver
com.novell.nds.dirxml.driver.skeleton.SkeletonDriverShim

      It illustrates some of the concepts in driver options
      files
-->
<driver-config name="Java Skeleton Driver">
   <driver-options>
      <option-1 display-name="Sample String option">This is a string</option-1>
      <option-2 display-name="Sample int option (enter an integer)">10</option-2>
   </driver-options>
   <subscriber-options>
      <sub-1 display-name="Sample Subscriber option">String for Subscriber</sub-1>
   </subscriber-options>
   <publisher-options>
      <pub-1 display-name="Sample Publisher option">String for Publisher</pub-1>
      <polling-interval display-name="Polling interval in seconds">2</polling-
interval>
   </publisher-options>
</driver-config>
```

Your configuration XML file must conform to the above format, in that it must contain a <driver-config> element with zero or one of each the following elements as children: <driver-options>, <subscriber-options>, and <publisher-options>.

The children of the <driver-options>, <subscriber-options>, and <publisher-options> elements can be named anything you like and should have an attribute named "display-name". The value of the display-name attribute is used to label an entry field in the user interface generated by iManager.

The <driver-options>, <subscriber-options>, and <publisher-options> together with their content are passed to the corresponding object init methods in your driver as children of the <init-params> element.

In addition, the Identity Manager engine provides a mechanism for attribute values from an Identity Vault object to be passed to your init methods. This is accomplished by placing a <config-object> element as a child of one of the options elements. The content of the <config-object> element is a <query> that specifies the Identity Vault object and attributes to read. The <config-object> element has a display-name attribute that is used to label a field in the generated user interface. The user interface field allows the user to specify the Identity Vault object referenced in the <query> element's "dest-dn" attribute (see example, below). The <read-attr> element underneath a <query> element has a "type" attribute with two possible values: "default" and "xml". If "xml" is specified, the attribute is assumed to be serialized XML and will be parsed and pasted into the initialization document as XML, rather than as a string, octet string, or stream.

For example, the following modifications to the XML configuration file for the Java skeleton driver results in the XML data from the skeleton driver's Create Rule (in this particular installation) being passed to the DriverShim init method:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--  this file contains a sample driver options xml document
```

```
      it is designed for use with the Java DirXML skeleton driver
      com.novell.nds.dirxml.driver.skeleton.SkeletonDriverShim

      It illustrates some of the concepts in driver options files
-->
<driver-config name="Java Skeleton Driver">
   <driver-options>
      <config-object display-name="Create Rule">
         <query dest-dn="novell\Driver Set\Java Skeleton Driver\Subscriber\Create
Rule" scope="entry">
            <read-attr attr-name="XmlData" type="xml"/>
         </query>
      </config-object>
      <option-1 display-name="Sample String option">This is a string</option-1>
      <option-2 display-name="Sample int option (enter an integer)">10</option-2>
   </driver-options>
   <subscriber-options>
      <sub-1 display-name="Sample Subscriber option">String for Subscriber</sub-1>
   </subscriber-options>
   <publisher-options>
      <pub-1 display-name="Sample Publisher option">String for Publisher</pub-1>
      <polling-interval display-name="Polling interval in seconds">2</polling-
interval>
   </publisher-options>
</driver-config>
```

Using this configuration file, the Identity Manager engine sends the following initialization document to the skeleton driver.

```
<nds dtdversion="1.0" ndsversion="8.5">
   <source>
      <product version="1.0">DirXML</product>
      <contact>Novell, Inc.</contact>
   </source>
   <input>
      <init-params src-dn="\PERIN-TAO\novell\Driver Set\Java Skeleton Driver">
         <authentication-info>
            <server>server.app:400</server>
            <user>User1</user>
            <password><!-- content suppressed --></password>
         </authentication-info>
```

```
        <driver-options>
            <instance class-name="DirXML-Rule" src-dn="PERIN-TAO\novell\Driver
Set\Java Skeleton Driver\Subscriber\Create Rule" src-entry-id="35867">
                <attr attr-name="XmlData">
                    <value timestamp="965252142#6" type="xml">
                        <create-rules>
                            <create-rule class-name="User">
                                <required-attr attr-name="Surname"/>
                                <required-attr attr-name="Telephone Number"/>
                            </create-rule>
                        </create-rules>
                    </value>
                </attr>
            </instance>
            <option-1 display-name="Sample String option">This is a
                        string</option-1>
            <option-2 display-name="Sample int option (enter an
                        integer)">10</option-2>
        </driver-options>
    </init-params>
  </input>
</nds>
```

The above document shows that the data of the XmlData attribute has been embedded in the initialization document. Reading the create rule is not useful for a driver, but it does serve to illustrate the concept of obtaining the value of an Identity Vault attribute as part of the driver initialization data.

2.12 Additional Tips for C++ Drivers

C++ requires memory management and does not provide generated documentation for helper classes and methods. The following sections provide information on these topics.

2.12.1 Memory Management

As with any C++ program, an Identity Manager driver must carefully manage the objects it creates. Primarily, a driver must destroy any XML document objects that it creates, and a driver must not destroy any XML document objects passed to it. This requires management for two primary areas.

Documents submitted to Identity Manager. Documents submitted to Identity Manager are submitted either through the XmlCommandProcessor interface passed to the PublicationShim::start method or through the XmlQueryProcessor interface passed to the SubscriptionShim::execute method. In either case the driver must handle destroying the XmlDocument object passed to Identity Manager as well as the underlying XML representation. For example, if the driver uses the factory methods Document_new and XmlDocument_newFromDOM, then the driver must destroy the XmlDocument object through XmlDocument_destroy method and the DOM object through DOM::Document::destroy method.

Documents returned to Identity Manager. Documents returned to Identity Manager are returned from DriverShim::init, DriverShim::shutdown, SubscriptionShim::init, SubscriptionShim::execute, PublicationShim::init, and PublicationShim::start. Such

documents must not be destroyed until they are no longer used. Identity Manager guarantees that a return document is no longer referenced when

- Another method in the same interface is called. For example, the document returned from SubscriptionShim::init() may be destroyed when SubscriptionShim::execute method is called.
- The same method is called again. For example, a document returned from a previous call of SubscriptionShim::execute method may be destroyed during a subsequent call to SubscriptionShim::execute method.
- The DriverShim::destroy method is called.

Any other objects that the driver creates are also the driver's responsibility to destroy. For example, if the driver calls FileOutputStream_newFromName, then the driver must also call FileOutputStream_destroy with the returned object when it is no longer used.

2.12.2 C++ Utility Functions and Interfaces

There are a number of utility functions available. These fall into the following general categories:

- Interface factories and destructors
- Encoding support
- Support for Identity Manager's XML dialect (XDS Support)

The following sections provide a brief overview of the utility functions. For a more complete description, see XML Interfaces for C++.

Interface Factories and Destructors

Identity Manager SDK supplies implementations of all required interfaces.The interface factories and corresponding destructors are defined in InterfaceFactory.h. If a factory method does not have a corresponding destructor method, the destructor method is defined in the interface itself (see DOM::Node::destroy, for example).

A driver writer is free to implement the interfaces for passing data to Identity Manager from the driver if such an implementation works better for a particular driver. However, all interfaces passed to the driver from Identity Manager will use Identity Manager's underlying implementation.

General Interfaces

- DriverFilter_new—Creates an implementation of DriverFilter. This is a useful for drivers who need to make use of the Event Filter in their driver.
- DriverFilter_destroy—Destroys an implementation of DriverFilter returned from the factory method.
- Trace_new—Creates an implementation of Trace. This is useful for debugging. The Trace interface causes messages to be written to the DSTrace screen and, optionally, a file.
- Trace_destroy—Destroys an implementation of Trace returned from Trace_new.

DOM Interfaces

- Document_new—Creates a new DOM Document object using the Identity Manager native implementation.
- Document_destroyInstance—Destroys a DOM Document object returned from Document_new.
- XmlDocument_newFromDOM—Creates a new XmlDocument implementation from a DOM Document object.

- XmlDocument_destroy—Destroys an XmlDocument implementation returned from a factory method.

Serialized XML Interfaces

- FileOutputStream_newFromName—Creates an implementation of OutputStream that uses a standard C library FILE as the underlying stream.
- FileOutputStream_newFromFILE—Creates an implementation of OutputStream that uses a standard C library FILE as the underlying stream.
- FileOutputStream_destroy—Destroyes a FileOutputStream returned from a factory method.
- ByteArrayOutputStream_new—Creates an implementation of OutputStream that uses a byte array as the underlying stream.
- ByteArrayOutputStream_getDataSize—Returns the size of the data in a ByteArrayOutputStream returned from ByteArrayOutputStream_new.
- ByteArrayOutputStream_getBytes—Returns a pointer to the data in a ByteArrayOutputStream returned from ByteArrayOutputStream_new.
- ByteArrayOutputStream_destroy—Destroyes a ByteArrayOutputStream returned from the factory method.
- XmlDocument_newFromBytes—Creates a new XmlDocument implementation from a byte array using the Identity Manager native implementation.
- XmlDocument_destroy—Destroys an XmlDocument implementation returned from a factory method.

SAX Interfaces

- Parser_new—Creates a new SAX Parser implementation using the Identity Manager native implementation.
- Parser_destroy—Destroys a SAX Parser returned from Parser_new.
- InputSource_new—Creates a new SAX InputSource implementation using the Identity Manager native implementation.
- InputSource_destroy—Destroys a SAX InputSource returned from InputSource_new.
- SAXException_new—Creates an implementation of SAXException. This is useful for drivers implementing the SAX DocumentHandler interface.
- SAXParseException_new—Creates an implementation of SAXParseException. This is useful for drivers implementing the SAX Parser interface.
- XmlDocument_newFromSAX—Creates a new XmlDocument implementation from a SAX Parser and InputSource.
- XmlDocument_destroy—Destroys an XmlDocument implementation returned from a factory method.

Encoding Support

The encoding support functions are for converting to and from Base64 encoding and for converting between UTF-8 and UTF-16 encoding. Base64 encoding is how Identity Manager encodes binary data in the XML documents used for encoding commands and events. UTF-8 and UTF-16 are two encoding methods for Unicode characters using 8- and 16-bit encoding units, respectively. Base64 functions are declared in Base64Codec.h, and the UTF functions are in UTFConverter.h.

- Base64Codec_encode—Encodes a byte array into UTF-16 characters using Base64 encoding.
- Base64Codec_encodeFree—Frees the UTF-16 string returned from Base64Codec_encode.
- Base64Codec_decode—Decodes a UTF-16 character string that represents binary data into a byte array.

- Base64Codec_decodeFree—Frees the byte array returned from Base64Codec_decode.
- UTFConverter_16to8—Converts a UTF-16 string to UTF-8.
- UTFConverter_8to16—Converts a UTF-8 string to UTF-16.
- UTFConverter_free—Frees a string returned from UTFConverter_16to8 or UTFConverter_8to16.

XDS Support

XDS (XML Directory Services) is the name of the XML dialect used by Identity Manager. The XDS support functions allow access to the strings necessary to create a valid XDS document and provide simple methods to create empty XDS input and output documents. The XDS support functions are in NdsDtd.h and include the following:

- NdsDtd_getStrings—Returns a pointer to a structure containing pointers to const unicode strings. These strings are tag names, attribute names, and attribute values defined in nds.dtd. See the actual include file for the structure declaration.
- NdsDtd_newInputDocument—Creates a new, empty input document for publishing data to Identity Manager.
- NdsDtd_newOutputDocument—Creates a new, empty output document for returning data to Identity Manager.
- NdsDtd_addStatus—Adds a status element to an input or an output element in an XDS document.

3.0 Debugging the Driver

An Identity Manager driver interacts with many external variables: the rules, eDirectory, the application, and the Identity Manager engine. To simplify the process of discovering the source of a problem, use the following.

3.1 Using DSTrace and the Identity Manager Trace Log

DSTrace is an eDirectory trace facility that displays messages from internal eDirectory activities. These messages can be displayed on the DSTrace screen, logged to a file, or both. The messages can be filtered by category. For Identity Manager driver development, you will want to activate the Identity Manager drivers messages.

To enable the Identity Manager drivers messages, select the platform and follow the steps:

Windows NT

1. From the eDirectory console, select NDS Trace Facility and press the Start button.

   The NDS Trace Facility screen appears.

2. From the Edit option on the NDS Trace Facility menu bar, select Options.
3. Press the Clear All button.
4. Check the Identity Manager Drivers box.
5. If you want Identity Manager Drivers messages enabled each time you run DSTrace, press the Save Default button.
6. To activate the settings and return to the trace screen, press the OK button.

Linux and Tru64 Unix

1. Enter following command in the shell: ndstrace.
2. Shut off all debug flags to avoid unnecessary messages. At the ndstrace screen, enter set ndstrace = nodebug.
3. To enable Identity Manager driver messages, enter dstrace +dvrs.
4. To enable Identity Manager events, enter dstrace +dxml.
5. To enable the log file option, enter dstrace file on.

   On Unix systems, the events are written to the /var/nds/NDSTRACE.LOG file.

3.1.1 Enabling Verbose Identity Manager Driver Messages

Verbose Identity Manager driver messages can be used to determine what is happening in the Identity Manager engine and what your driver is processing. To enable verbose messages, complete the following steps:

1. In iManager, select the required Driver Set.
2. Click Driver Set > Edit Driver Set properties.
3. In the Identity Manager tab, select the Misc tab.
4. In the Trace level field, enter one of the following values in the:
   - 0 — Displays no verbose messages.
   - 1— Displays Identity Manager engine basic processing messages.
   - 2 — Displays messages from level 1 plus the XML documents that are passed between the engine and driver.
   - 3 — Displays messages from level 2 plus additional rule processing messages. In addition, displays the XML documents that result from stylesheet rule processing.
5. Click the OK button to save the changes.

3.1.2 Enabling the Identity Manager Trace Log

DSTrace messages are useful, but they eventually scroll out of the DSTrace buffer. To preserve the Identity Manager messages, you can set up a log file into which the Identity Manager driver messages will be written (even when DSTrace is not running). To enable the trace log, complete the following steps:

1. In iManager, select the required Driver Set.
2. Click Driver Set > Edit Driver Set properties.
3. In the Identity Manager tab, select the Misc tab.
4. In the Trace file field, enter the name of a file into which the Identity Manager driver messages are to be written.

   The location for the log file depends on the platform:

   - On Win32, if you do not enter a path with the filename, the default location is the c:\novell\nds\dibfiles directory.
5. Click the OK button to save the changes.

3.1.3 Adding Trace Messages to Your Driver

Drivers should send messages to DSTrace by creating and using one or more Identity Manager trace objects. Using these objects, your driver can write messages and XML documents to the DSTrace screen and the Identity Manager trace log.

Creating the Trace Object

Each trace object has an identifying string associated with it, and this string appears in the trace message. A driver typically creates three trace objects:

- One associated with the driver object
- One associated with the subscriber object
- One associated with the publisher object

For example, if the subscriber object creates a trace object with the identifying string "NdsToNds Subscriber" then all trace messages output using that object will look similar to the following.

```
TRACE:  NdsToNds Subscriber: <message>
```

Java

In Java the trace objects are simply instances of the com.novell.nds.dirxml.driver.Trace class. For example:

```
//In the class definition
Trace tracer;

//In the constructor
tracer = new Trace("My Subscriber");
```

C++

In C++ a factory method from InterfaceFactory.h, Trace_new(), is called.

```
//in the class definition:
Trace * tracer;

//in the constructor:
Trace * tracer = Trace_new("My Subscriber");

//in the destructor:
Trace_destroy(tracer);
```

Writing to the Trace Object

Writing a message is as simple as calling the trace method with a literal string argument. Additionally, XML documents can be written to the trace screen and log, using an overload of the trace method. There are also trace method overloads that let you specify at what level the associated message appears. For example, if you want to specify an excruciating level of detail for debugging, you could specify that such

messages only appeared at DirXML-DriverTraceLevel values of 4 or higher. See the Javadocs and Trace Interface in XML Interfaces for C++ for detailed descriptions of the trace overloads.

Java

```
//output a message to the trace facility
tracer.trace("init");

//output a document to the trace facility
tracer.trace(initDocument);
```

C++

```
//output a message to the trace facility
tracer->trace("init");

//output a document to the trace facility
tracer->trace(initDocument);
```

3.2 Using a Debugger with a C++ Driver

Debugging varies by platform. Currently, C++ driver debugging is not available on Unix platforms.

3.2.1 DLLs on Windows (NT, 2000, XP)

Identity Manager drivers run as part of the eDirectory process (dhost.exe). This means that you need to attach a debugger to the dhost.exe process in order to debug your driver. The process is easier if dhost.exe is not running as a service. To start dhost.exe from the command line, use the following command line option:

```
dhost /datadir=c:\novell\nds\dibfiles
```

If you have not used a default installation, replace the specified path with the location of the dibfiles directory.

You can also start dhost.exe from within a debugger using the above command line as a model.

Once a debugger attached, you can set your breakpoints within your driver code. This may require preloading your driver DLL module.

For Visual Studio, preloaded modules are configured under Project | Settings..., Debug tab, Additional DLLs.

Your driver then needs to be started. Use iManager.

3.3 Using a Debugger with a Java Driver

You cannot currently debug a Java driver on NetWare or Unix systems. For the Windows platform (NT or 2000), the following debug interfaces and debuggers are available:

- Agent Debugger

* Java Platform Debugger Architecture (JPDA)
* Visual Cafe 3.0 Debugger
* JDB Debugger

3.3.1 Agent Debugger

To use a debug agent, you must install the debug agent and then enable it. Complete the following steps.

1. Obtain a copy of JDK 1.7.
2. Make a backup copy of the Identity Manager jre directory (default location—C:\novell\nds\jre).
3. Delete the Identity Manager jre directory.
4. Copy the JRE directory from the JDK into the Identity Manager jre directory.
5. Copy the contents of the jdk/lib directory (includes tools.jar) into the Identity Manager jre/lib directory.
6. Add the DirXML-JavaDebugPort attribute to the DirXML-DriverSet object.
    1. In iManager, select the required Driver Set.
    2. Click Driver Set > Edit Driver Set properties.
    3. In the Identity Manager tab, select the Misc tab.
    4. Set the value for the field Java debug port to some TCP port not in use on your machine (8080 usually works).
    5. Click the OK button.
7. Restart eDirectory.
8. Before the first driver starts, a small dialog will pop up with the debugger password. At this point, you can choose to attach a debugger to the JVM and press OK.

   Sometimes you can attach a debugger after you have dismissed the dialog that pauses for you to attach to the JVM; sometimes you can't.

3.3.2 Java Platform Debugger Architecture (JPDA)

To use a JPDA type debugger, you must set options for the JVM using an Identity Manager environment variable: DHOST_JVM_OPTIONS. Java 2 options are set using this environment variable and passed to the JVM by Identity Manager.

For more information on the environment variables, see JVM Variables. For information on the JVM options and JPDA see Java Platform Debugger Architecture.

3.3.3 Visual Cafe 3.0 Debugger

The Visual Cafe debugger is an agent-type debugger.To set up a debug session, complete the following steps.

1. Create a project for debugging.
2. Set the main class to "sun.tools.agent.EmptyApp".

3. Set your source directories and input class files directories appropriately for your environment.

4. Attach the debugger using Projects | Debug in Waiting VM. Use “localhost” as the host name and the password supplied by the dialog box.

   If you stop the debugging session in VisualCafe, it shuts down the JVM inside of eDirectory and you will have to restart eDirectory to get it back up again.

   The VisualCafe debugger has a lot of quirks. Be aware of the following:

   - Sometimes it has trouble attaching to the JVM.
   - Sometimes it gets internal java exceptions, after which no data will display until you shut down VisualCafe and restart it.
   - Often it will still stop at breakpoints that have been removed or disabled.

3.3.4 JDB Debugger

To attach the jdb debugger, complete the following steps.

1. Enter the following command:

```
jdb -host localhost -password <password>
```

2. Set the path to your source directories using the “use” command.

If you stop the debugging session in jdb, it shuts down the JVM inside of eDirectory and you will have to restart eDirectory to get it back up again.

You can create a file called startup.jdb in the working directory that can contain commands that will be executed when jdb starts. This is a good place to put the “use” command to set up the source directories.

The password is really just an obfuscated version of the port number. Since you are picking a static port number, you can reuse any of the passwords that are generated. If you see one that is easy to memorize, remember and use it instead of using a new password every time.

3.3.5 JVM Variables

A number of environment variables may be set to control the JVM under Windows NT and Windows 2000:

Table 3-1 JVM environment variables for Windows

Variable	Description
DHOST_JVM_USE_VFPRINTF	If set to a value other than “0”, causes the installation of a vfprintf hook function that will write to a log file named “jvm_vfprintf.log” in the temp directory. This will also enable verbose class and JNI messages.
DHOST_JVM_VERBOSE_GC	If set to a value other than “0”, enables verbose garbage collector messages. This is only useful in conjunction with JVM_USE_VFPRINTF.
DHOST_JVM_INITIAL_HEAP	Specifies the value, in decimal number of bytes, of the initial JVM heap size.
DHOST_JVM_MAX_HEAP	Specifies the value, in decimal number of bytes, of maximum JVM heap size.
DHOST_JVM_OPTIONS	Specifies the arguments for the JVM 1.2. For example:

	• -Xnoagent -Xdebug -Xrunjdwp: transport=dt_socket,server=y, address=8000 Each option string is separated by whitespace. If an option string contains whitespace, it must be enclosed in double quotes.
DHOST_JVM_DESTROY	If set to a value other than “0”, causes the JVM loader thread to call DestroyJavaVM() when Identity Manager is unloaded. This is useful to cause the Java 2 hprof profiler to dump its data to a file. *NOTE:* The JVM is unusable after this call and cannot be reloaded. DHOST must be shutdown and restarted to use the JVM again. This means Identity Manager cannot be restarted until DHOST has restarted.

The following options can be used on NetWare:

Table 3-2 JVM environment variables for NetWare

Option	Description
DIRXML_JVM_INITIAL_HEAP	Specifies the value, in bytes, of the initial java heap size. This option works with JVM 1.1.x and 1.2.x. Examples: • DIRXML_JVM_INITIAL_HEAP=4M • DIRXML_JVM_INITIAL_HEAP=4096K
DIRXML_JVM_MAX_HEAP	Specifies the value, in decimal bytes, of the maximum java heap size. This option works with JVM 1.1.x and 1.2.x. Examples: • DIRXML_JVM_MAX_HEAP=36M • DIRXML_JVM_MAX_HEAP=36864K
DIRXML_JVM_VIRTUAL_HEAP	Specifies the value, in decimal bytes, of the virtual java heap size. This option works with JVM 1.1.x and 1.2.x. Examples: • DIRXML_JVM_VIRTUAL_HEAP=36M • DIRXML_JVM_VIRTUAL_HEAP=36864K
DIRXML_JVM_OPTIONS	Specifies the arguments for the JVM 1.2. For example: • -Xnoagent -Xdebug -Xdebugport8080 Each option string is separated by whitespace. If an option string contains whitespace, it must be enclosed in double quotes. This option only works with JVM 1.2.x.
DirXML_JVM_C_STACK_SIZE	Specifies in decimal bytes the Java C process stack size. Examples: • DIRXML_JVM_C_STACK_SIZE=128K • DIRXML_JVM_C_STACK_SIZE=256K Default: 128K

4.0 Introduction to the Rules and Filters

Identity Manager provides two complementary mechanisms for customizing the data synchronization processes: transformation rules and event filters. As a driver developer, you need to understand how the rules and filters work so that you can carefully consider what logic belongs in the driver and what logic belongs in the rules and filters.

4.1 Event Filters

Event filters specify the object classes and the attributes for which the Identity Manager engine processes events. Separate event filters are specified for the subscriber and publisher channels. Event filters only pass events occurring on objects whose base class matches one of those classes specified by the filter. Event filters do not pass events occurring on objects that are a subordinate class of a class specified in the filter unless the subordinate class is also specified.

NOTE: In eDirectory, a base class is the object class that is used to create an entry. You must specify that class in the filter, rather than a super class from which the base class inherits.

For example, if the User class is specified in the event filter with the Surname and Given Name attributes, the Identity Manager engine passes on any changes to these attributes. However, if the entry's Telephone Number attribute is modified, the Identity Manager engine drops this event because the Telephone Number attribute is not in the event filter.

Event filters are configurable by the system administrator and must be configured to include the following:

- Attributes required by the rules
- Attributes that are to be synchronized

The publisher channel in the Identity Manager driver should use the publisher filter to remove events that contain classes and attributes that do not match the filter. Identity Manager provides methods for using the channel filter:

- For Java methods, see the DriverFilter class.
- For C++ methods, see Driver Filter Interface in XML Interfaces C++.

For the format of a filter, see <driver-filter>.

4.2 Transformation Rules

Rules help the Identity Manager engine transform an event from a channel input into a set of commands for a channel output.

- For the subscriber channel, the input event comes from the Identity Vault and the output command goes to the application.
- For the publisher channel, the input event comes from application and the output command goes to the Identity Vault.

Rules are configurable by the system administrator so that the rules can be customized to do whatever an individual installation requires. Rules are processed completely within the Identity Manager engine. As such, the driver writer should carefully consider what logic to put in the driver as opposed to what logic to leave configurable via rules.

Some rules perform a well-defined role in the event-to-command transformation while others allow for more general customization. All rules can be implemented using XSLT style sheets, but rules that perform well-defined roles more commonly use an XML format which is Identity Manager-specific and more easily describes the transformation needed.

In Identity Manager 4.5, the feature called Rule chaining enables multiple rule objects to be chained together to form the total rule processing for a rule step (e.g., Create Rule, Placement Rule, etc.). This allows a stylesheet to supplement the operation of a simple rule without having to completely replace the rule with a stylesheet.

Rules are chained together using the DirXML-NextTransformation attribute on the rule object. Any number of rules may be chained; multiple simple rules and stylesheets may be freely mixed. There may not be any loops in the chain, and any simple rules used must match the rule step. In other words, if rules are being chained in the Create Rule step, any simple rules must use the simple Create Rule syntax.

Rules that use Identity Manager-specific XML formats can usually be created and edited using iManage snap-ins that guide the administrator in creating the rules so that the administrator does not have to see the raw XML that encodes the rule. However, when an XSLT style sheet is used as a rule (either for those rules that can only be implemented using XSLT or as a replacement for one of the more common rules), the XSLT must be programmed manually.

Rules are stored in the XmlData attribute of DirXML-Rule and DirXML-Stylesheet objects. The DirXML-Driver, DirXML-Subscriber, and DirXML-Publisher objects use the following attributes to reference these rule objects.

Table 4-1 Identity Manager attributes for referencing rule objects

Object	Attribute	Rule
DirXML-Driver	DirXML-MappingRule	Schema mapping rules
	DirXML-InputTransform	Input transformation style sheet
	DirXML-OutputTransform	Output transformation style sheet
DirXML-Subscriber	DirXML-EventTransformationRule	Subscriber event transformation style sheet
	DirXML-CommandTransformationRule	Subscriber command transformation style sheet
	DirXML-MatchingRule	Subscriber matching rules
	DirXML-CreateRule	Subscriber create rules
	DirXML-PlacementRule	Subscriber placement rules
	DirXML-CommandTransformationRule	Subscriber placement rules
DirXML-Publisher	DirXML-EventTransformationRule	Publisher event transformation style sheet
	DirXML-CommandTransformationRule	Subscriber command transformation style sheet
	DirXML-MatchingRule	Publisher matching rules
	DirXML-CreateRule	Publisher create rules
	DirXML-PlacementRule	Publisher placement rules

Rules contain either channel-independent transformations or channel-dependent transformations.

- Rules specified on the DirXML-Driver object specify channel-independent transformations.
- Rules specified on the DirXML-Subscriber object or the DirXML-Publisher object specify channel-dependent transformations.

4.3 Channel-Independent Transformations

Channel-independent transformations are applied to all information being passed between the Identity Manager engine and the application driver.

Examples of information that may need to be transformed include class names, attribute names, and attribute value formats.

In addition if your driver wants to use an XML format other than the Identity Manager-specified format (XDS), then the XML documents must be transformed between XDS and the driver-specific format as they are passed between the Identity Manager engine and your driver.

The Identity Manager engine uses three types of channel-independent transformations:

- Schema Mapping Rules
- Input Transformation Style Sheet
- Output Transformation Style Sheet

4.3.1 Schema Mapping Rules

Schema mapping rules map class names and attribute names between the Identity Vault schema and the application schema. The schema mapping rules are configured on the DirXML-Driver object, are applied to both subscriber and publisher channels, and are applied to command documents, event documents, and response documents.

From a driver writer's point of view, the significance of the schema mapping rules is that you do not need to worry about the Identity Vault class names and attribute names. All class names and attribute names in the documents received by your driver are the names from your supported application:

- Before an XDS document is passed to your driver, all names are mapped from the Identity Vault name space into the application name space.
- After your driver passes a document to the Identity Manager engine and the input transformations are applied, but before any further processing takes place, all names are mapped from the application name space to the Identity Vault name space.

NOTE: If a class name or attribute name does not appear in the schema mapping rules, that name will not be mapped.

Schema mapping rules are configurable from iManager and should contain all the object classes and attributes specified in the event filters and the rules.

For more detailed information about schema mapping rules, see Section 8.1, Schema Mapping Elements.

4.3.2 Input Transformation Style Sheet

The purpose of the input transformation is to perform any preliminary transformations necessary on the XML documents passed to the Identity Manager engine from your driver. This transformation is always implemented as an XSLT style sheet.Typically an input transformation style sheet will only be used on an installation-specific basis. A common use for the input transformation is data mapping. For example, suppose your application supplies telephone numbers in the following format:

```
(nnn) nnn-nnnn
```

Suppose that the installation requires that telephone numbers in the Identity Vault have the following format:

```
nnn nnn nnnn
```

An input transformation style sheet can be used to transform the telephone number data from your application's format to the desired format.

NOTE: All schema names in the XML processed by the input transformation style sheet are in the application's name space.

An input transformation style sheet can transform an arbitrary XML format native to the target application to the format expected by Identity Manager. However, this is discouraged because it makes it harder for administrators to customize the transformation for installation-specific needs. The driver should perform this transformation by calling the Novell XSLT processor directly.

For more detailed information about writing style sheets and using the Novell XSLT processor, see Style Sheets.

4.3.3 Output Transformation Style Sheet

The purpose of the output transformation is to perform any final transformations necessary on the XML documents passed to your driver from the Identity Manager engine. This transformation is always implemented as an XSLT style sheet.

Typically, for data mapping purposes, the output transformation is the inverse of the input transformation. See Input Transformation Style Sheet for an example of data mapping.

NOTE: All schema names in the XML processed by the output transformation style sheet are in the application's name space.

An output transformation style sheet can transform the XML format provided by Identity Manager to an arbitrary XML format native to the target application. However, this is discouraged because it makes it harder for administrators to customize the transformation for installation-specific needs. The driver should perform this transformation by calling the Novell XSLT processor directly.

For more detailed information about writing style sheets and using the Novell XSLT processor, see Style Sheets.

4.4 Channel-Dependent Transformations

Channel-dependent transformations are applied to one channel, either the publisher or the subscriber, but not both. For example, on the subscriber channel, the transformation rules are used to match entries in the Identity Vault with corresponding entries in the application and to impose restrictions on creating new entries in the application when a match is not found. On the publisher channel, the transformation rules are used to match entries in the application with corresponding entries in the Identity Vault and to impose restrictions on creating new entries in the Identity Vault when a match is not found.

The Identity Manager engine uses four types of channel-dependent transformations:

- Matching Rules
- Create Rules
- Placement Rules
- Event Transformation Rules

Event transformation rules can be applied to any type of event. Three of these rules (matching, create, and placement) apply only to add events. When an object is added to either the Identity Vault or the application, the following questions must be answered:

1. Does a corresponding object already exist in the target?
2. If not, does the object have sufficient data to enable a successful create operation in the target?
3. If the object can be created, where should the object be placed in the target's hierarchy?

Matching rules answer question #1; create rules, question #2; and placement rules, question #3.

The Identity Manager engine transforms modify events to add events when the modify event does not include data that associates the existing entry in the source with an existing entry in the target.

4.4.1 Matching Rules

Matching rules establish links between an existing entry in the Identity Vault and an existing entry in the external application. The matching rules specify which class and attribute values must match for an entry in the Identity Vault and an entry in the application to be marked as corresponding entries. If a match is successful, an association between the two entries is created. If a match is not successful, the create rules are used.

The Identity Manager engine uses an association to establish and maintain a correspondence between an entry in the Identity Vault and an entry in the application. Once established, the Identity Manager engine uses the association rather than the matching rules to determine the correspondence between two entries. An association is a unique key that the driver supplies from the application that uniquely identifies the entry. The Identity Manager engine stores this key in the Identity Vault. The driver's publisher must supply this key with each event it publishes to the Identity Vault. Once the key is established for an entry, the driver's subscriber receives this key for all commands coming from the Identity Manager engine.

The publisher and the subscriber can have their own matching rules, or they can share the same rule. A matching rule can specify more than one rule. Identity Manager applies the rules in the order they are listed in the file. Identity Manager defines three items for specifying matching criteria:

- Object classes. For example, the entry must belong to the User class.
- Attribute values. For example, the entry must have Surname, Given Name, and Telephone Number attributes that match the target entry's values.
- Placement in the directory's hierarchy. For example, the entry must come from a specified container or its subordinate containers.

Matching rules must be defined such that only a single match results from the specified criteria. In practice, this means the matching criteria must be either a single unique attribute (such as the GUID attribute) or a combination of attributes (such as Surname, Given Name, and Telephone Number).

Any attributes or classes specified in matching rules must also be included in the event filter (see Section 4.1, Event Filters).

A matching rule can be either

- An XML file stored in a DirXML-Rule object (for the syntax, see Section 8.2, Matching Rule Elements)
- An XSLT style sheet stored in a DirXML-Stylesheet object (for more information, see Style Sheets)

4.4.2 Create Rules

The create rules are applied to add events when the matching rules fail to find a match. The create rules specify the minimum set of data that an entry must have before it can be created in the target. Create rules can also perform other modifications to the add event such as

- Supply default values for attributes
- Specify an object to use as a template for creating a new entry

They can also veto an add event if the add event fails the conditions imposed by the create rules. For example, if the create rule requires the entry to have a telephone number and it doesn't have one, the add event fails.

If the add event passes the criteria in the create rules, the placement rules are then applied.

Both the subscriber and the publisher can reference create rules. Create rules can be either

- An XML file stored in a DirXML-Rule object (for the syntax, see Section 8.3, Create Rule Elements)
- An XSLT style sheet stored in a DirXML-Stylesheet object (for more information, see Style Sheets)

4.4.3 Placement Rules

Placement rules specify the entry's name and determine where a new entry is placed in the Identity Vault or in the application. Because placement rules determine location, they also determine the distinguished name of the entry.

Placement rules are applied to the add events that pass the criteria in the create rules.

Both the subscriber and the publisher can reference placement rules. Placement rules can be either

- An XML file stored in a DirXML-Rule object (for the syntax, see Section 8.4, Placement Rule Elements)
- An XSLT style sheet stored in a DirXML-Stylesheet object (for more information, see Style Sheets)

4.4.4 Event Transformation Rules

An event transformation rule can apply just to the subscriber, just to the publisher, or both. Usually the subscriber and the publisher reference different event transformation rules.

These rules perform preliminary transformations on events:

- When referenced by the subscriber, the rules apply to events sent by the Identity Vault to the Identity Manager engine and are processed before any other event processing.
- When referenced by the publisher, the rules apply to events sent by the driver to the Identity Manager engine and are processed after any input transformation style sheet and schema mapping rules but before any other event processing.

Event transformation rules are always implemented using an XSLT style sheet. They are commonly used for the following tasks:

- Custom event filtering.
- Changing event types, for example, transforming an object delete event to a modification event that causes the object to be archived.
- Transforming events to custom commands.
- Generating additional events, for example, changing a delete event to a modify and a move.

For more information and an example, see Section 8.5, Event Transformation Rules.

4.4.5 Command Transformation Rules

A command transformation rule can apply just to the subscriber, just to the publisher, or both. Usually the subscriber and the publisher reference different command transformation rules.

These rules provide final processing on commands before the commands are sent to the Identity Vault or the application:

- When referenced by the subscriber, these rules are executed directly before the Schema Mapping Rule. Both the Schema Mapping Rule and the Output Transformation are executed after the Command Transformation Rule on the Subscriber channel.
- When referenced by the publisher, these rules are executed after all other rules and are executed directly before the Identity Manager engine applies the commands in the command document to the Identity Vault. It is the "last chance" to modify a command before the command is applied to the Identity Vault.

The Command Transformation Rule is always implemented using an XSLT stylesheet. Many applications that had to be performed in the Event Transformation Rule in Identity Manager 1.0 can be more easily performed in the Command Transformation Rule in Identity Manager 4.5. This is because all event-to-command processing performed by the Identity Manager engine has already been performed. They are commonly used for the following tasks:

- Changing the command type (for example, an object delete command might be transformed into a modification that will cause the object to be archived).
- Blocking commands.
- Adding additional commands.
- Controlling the output of the Identity Manager engine’s "merge" process.
- Generating additional events, for example, changing a delete event to a modify and move event.

The Command Transformation Rule is always implemented using an XSLT stylesheet. Many applications that had to be performed in the Event Transformation Rule in Identity Manager 1.0 can be more easily performed in the Command Transformation Rule in Identity Manager 4.5. This is because all event to command processing performed by the DirMXL Engine has already been performed.

Note that the Command Transformation Rule did not exist in Identity Manager 1.0; the Command Transformation Rule was added in Identity Manager 4.1.

4.5 Event Processing

The Identity Manager engine applies rules and event filters to events coming from the Identity Vault and from the application. The order varies according to the channel. The following sections list the order of application. The lists assume that the event passes all the rules. Events which fail are dropped.

4.5.1 Subscriber Channel

When an operation occurs in the Identity Vault that generates an add, move, delete, rename, or modify event, the Identity Manager engine performs the following steps in the order listed.

1. Verifies that the event contains object classes and attributes allowed by the subscriber filter.
2. Applies the event transformation rules.
3. If the event is a modify event on an unassociated object, converts the event to an add event.
4. If the event is an add event, applies the following:
   - Matching rules
   - Create rules
   - Placement rules
5. Applies the command transformation rules.
6. Applies the schema mapping rules.
7. Applies any output transformations.
8. Sends the commands to the driver.

Any of the rules can cause an event to be dropped.

- If event doesn't pass the filter, it is dropped.
- If the matching rules produce multiple matches for the entry, an error is logged and the event is dropped.
- If the entry does not pass the criteria for a create rule, the event is dropped.
- If the entry does not pass the criteria for a placement rule, the event is dropped.

You need to analyze the dropped events to determine if the rules are producing the desired results or if the rules need to be refined.

4.5.2 Publisher Channel

When an event occurs in the application, the publisher channel of the driver sends the engine an XML document to process. The Identity Manager engine then performs the following steps before sending the event to the Identity Vault.

1. Applies the input transformation rules.
2. Applies the schema mapping rules.
3. Applies the event transformation rules.
4. Applies the publisher event filter.
5. If the event is a modify event on an unassociated object, converts the event to an add event.
6. If the event is an add event, applies the following rules:
    - Matching rules
    - Create rules
    - Placement rules
7. Applies the command transformation rule.
8. Sends the event to the Identity Vault.

Any of the rules can cause an event to be dropped.

- If event doesn't pass the filter, it is dropped
- If the matching rules produce multiple matches for the entry, an error is logged and the event is dropped.
- If the entry does not pass the criteria for a create rule, the event is dropped.
- If the entry does not pass the criteria for a placement rule, the event is dropped.

You need to analyze the dropped events to determine if the rules are producing the desired results or if the rules need to be refined.

6.0 Driver Installation

Driver installation must perform the following tasks:

- Copy the driver to the Identity Vault server
- Create the driver objects in the Identity Vault
- Customize the driver for the environment

The following sections describe the details of these tasks. Once you have manually configured your driver, you can export the configuration from the iManager and then ship this file in your product. The administrator then needs instructions on importing this file in iManager and either prompts or instructions on configuring the options that must be customized.

6.1 Copy the Driver

Since Identity Manager drivers can be installed on multiple platforms (NetWare, NT, Solaris, and Linux), your installation program needs to handle all the platforms your driver supports.

- *Win32:* DLLs should be copied to the C:\Novell\nds directory and jar files to the C:\Novell\nds\lib directory.
- *UNIX:* The libcskeldrv.so should be copied to the /usr/lib/nds-modules directory and jar files should be copied to the /usr/lib/dirxml/classes directory.

Before copying the driver to the Identity Vault server, the installation program should verify that the selected server is running eDirectory version 8.8.8 or later.

6.2 Create the Driver Objects

When Identity Manager is installed with eDirectory version 8.8.8, a DirXML-DriverSet object is created in the Identity Vault. When you manually install your driver on the Identity Vault server, you need to create the following objects:

- In the DirXML-DriverSet container object, create a DirXML-Driver object. Required attribute, CN of the object.
- In the DirXML-Driver container, create a DirXML-Subscriber object, a DirXML-Publisher object, a DirXML-Rule object for mapping, and optional style sheet objects. Required attribute for the objects, CN of the object.
- In the DirXML-Subscriber container, create DirXML-Rule objects for placement, matching, and create rules; an optional an object for event transformations. Required attribute for the objects, CN of the object.
- In the DirXML-Publisher container, DirXML-Rule objects for placement, matching, and create rules; an optional an object for event transformations. Required attribute for the objects, CN of the object.

In the DirXML-Driver object, you need to write values for the following attributes:

- DirXML-JavaModule or DirXML-NativeModule. These attributes supply the DirXML engine with the name of the driver's executable module. A Java application uses the DirXML-JavaModule attribute to specify the Java class that needs to be loaded, and a DLL or NLM driver uses the DirXML-NativeModule to specify the library that needs to be loaded.
- Shim authentication attributes (DirXML-ShimAuthID, DirXML-ShimAuthPassword, and DirXML-ShimAuthServer). These attributes allow the driver to authenticate to the external application. The export file should prompt the user to enter values for these attributes.
- DirXML-DriverStartOption. This attribute determines when the driver starts: (1) automatically when eDirectory is initialized, (2) manually when the administrator starts the driver from Designer or iManager, or (3) never because the driver has been disabled.

- DirXML-ShimConfigInfo. This attribute holds the configuration options for the driver and the shims. Depending upon the configuration options you supply, the export file may or may not prompt the user for configuration details. If your configuration options are for optimizing the performance of the driver, the installation program can just copy the file containing the default values to this attribute. Otherwise, the export file should prompt the administrator for the values that must be set and then copy the file to the attribute.

For the DirXML-Rule objects, prototype files need to be copied to the XmlData attribute. Section 8.0, Rule Reference describes all the types of rules, filters, and style sheets that are possible. You need to create the generic rules that are appropriate for the publisher and subscriber, create rule objects for them, and associate them with publisher or subscriber through the appropriate attribute (DirXML-CreateRule, DirXML-DriverFilter, DirXML-MatchingRule, and DirXML-PlacementRule).

Since the rules and filters will require input from the administrator who knows what data needs to be synchronized between the two databases, your driver needs to come with instructions on how to configure these items from Designer or iManager. See the eDirectory Administration Guide for a prototype of sample instructions.

6.3 Exporting the Configuration

Once you have manually configured the driver and have it running, use iManager to export the configuration. When this file is imported on a new system, it creates all the objects required by the driver and configures them with the appropriate options, rules, and style sheets.

6.4 Set Up the Server Environment

The system administrator needs to set up filtered replicas. The server that is going to run Identity Manager drivers should hold a filtered replica of every partition in the eDirectory tree. These filtered replicas should be configured to contain the data for the attributes and classes that eDirectory will share with the application.

7.0 DTD Commands and Events

The eDirectory document type definition file (nds.dtd) defines the schema of the XML documents that the Identity Manager engine can process. XML documents that do not conform to this schema generate errors.

The nds.dtd file defines the following:

- Input and output commands and events (such as add, delete, modify, and rename) that can be performed on entries and the data that must be included with each. For more information, see Section 7.2, Input and Output Elements.
- Driver initialization operations (such as authentication information, driver filter, configuration options, and state) for the driver shim, publisher shim, and subscriber shim and the data that these operations require. For more information, see Section 7.4, Other Elements.
- Schema operations for defining class and attribute definitions. For more information, see <schema-def>.
- Rules for schema mapping, matching, creation, and placement. For more information, see Rule Reference.

Remember the following when reading a DTD file.

Marker	Meaning
?	0 or 1 of these can be included
+	1 or more of these must be included
*	0 or more of these can be included

CDATA	Character data
PCDATA	Parsed character data
<!	Beginning of an element, entity, or attribute definition
>	End of an element, entity, or attribute definition

*NOTE:*Generated DTD reference documentation is included with the Javadoc.

7.1 Top Level Elements

All XML documents sent to the Identity Manager engine or the external application must start with a top-level element. Two of these elements (<nds> and <driver-config>) are described in this chapter:

- <nds>
- <driver-config>

The other top-level elements are rule elements, and they are described in Rule Reference.

<nds>

Specifies the top level element that is used in all documents sent and returned by the Identity Manager interface (for a description of these interfaces, see Writing an Identity Manager Driver).

Description

All XML documents sent as a request operation or returned as a response to that operation must use <nds> as the top level element in the document. In addition, these documents may contain exactly one <input> element or exactly one <output> element, not both. The <source> element is optional.

Definition

```
<!ELEMENT nds (source?, (input | output))>
<!ATTLIST nds
          ndsversion     CDATA          #REQUIRED
          dtdversion     CDATA          #REQUIRED >

<!ELEMENT source (product?, contact?)>

<!ELEMENT product (#PCDATA)>
<!ATTLIST product
          version        CDATA          #IMPLIED
```

```
        asn1id          CDATA          #IMPLIED>
```

```
<!ELEMENT contact (#PCDATA)>
```

Attributes of <nds>

ndsversion

Specifies the current version of eDirectory. It must be set to 8.8.8 or higher.

dtdversion

Specifies the current version of Identity Manager. It must be set to 1.0.

Elements

<source>

Specifies the source that created the XML document.

<input>

Specifies the operations to perform. The document can contain only one <input> element, and if it contains an <input> element, it cannot contain an <output> element.

<output>

Specifies the results of an operation. The document can contain only one <output> element, and if it contains an <output> element, it cannot contain an <input> element.

Parent

- None

Example

The Identity Manager engine sends the following:

```
<nds dtdversion="1.0" ndsversion="8.5">
   <source>
      <product asn1id="2 16 840 1 113719 1 x" version="1.0b3">DirXML</product>
      <contact>Novell, Inc.</contact>
   </source>
   <input>
      <modify class-name="User" event-id="0" src-dn="\ATREE\Users\Julia"
              src-entry-id="33967">
```

```
         <association state="associated">{B43E7155-CDF9-d311-9846-0008C76B
              16C2}</association>
         <modify-attr attr-name="Surname">
            <add-value>
               <value type="string">Gulia</value>
            </add-value>
         </modify-attr>
      </modify>
   </input>
</nds>
```

The driver returns the following

```
<nds dtdversion="1.0" ndsversion="8.5">
   <source>
      <product version="1.0b3">Some Application Driver</product>
         <contact>Nobody in particular</contact>
   </source>
   <output>
      <status event-id="0" level="success"/>
   </output>
</nds>
```

<driver-config>

Specifies driver-specific configuration options.

Description

The <driver-config> element is a top level element. The information specified in this XML document is stored in the attribute of the DirXML-Driver object in the Identity Vault.

In iManager, each driver defined configuration element is displayed as an edit control which allows the system administrator to edit the content of the element. iManager uses the display-name attribute as the name for the element in the edit control. If no display-name is provided, the element tag name is used.

If the options contain a <config-object> element, iManager displays each <config-object> element as a single valued distinguished name control which allows the system administrator to select a distinguished name for the dest-dn attribute of the enclosed <query> element.

Definition

```
<!ELEMENT driver-config (driver-options?,
```

```
                         subscriber-options?,
                         publisher-options?) >
<!ATTLIST driver-config
          name       CDATA        #IMPLIED>
```

Attributes

name

Specifies the name of the Identity Manager driver

Elements

<driver-options>

Specifies configuration options for the DriverShim.

<subscriber-options>

Specifies configuration options for the SubscriptionShim.

<publisher-options>

Specifies configuration options for the PublisherShim.

Parent

- None

Example

```
<driver-config name="Netscape DirXML Driver">
   <driver-options>
      <display-method display-name="Debug Output (0-none,
                 1-Window, 2-DSTrace)">1</display-method>
   </driver-options>
   <subscriber-options>
      <config-object display-name="Super driver
              configuration data">
         <query dest-dn="novell/Driver Set/Super Driver/
                 Config Object" scope="entry"
                 event-id="config1">
```

```
            <read-attr attr-name="Some Attribute"/>
            <read-attr attr-name="XmlData" type="xml"/>
        </query>
     </config-object>
  </subscriber-options>
  <publisher-options>
      <pollRate display-name="Poll rate in seconds">5</
             pollRate>
      <changeLogSuffix display-name="Netscape changelog
             suffix">cn=changelog</changeLogSuffix>
      <changeLogBegin display-name="Starting changelog
             (1-First, 2-New, 3-Continue)">2
             </changeLogBegin>
  </publisher-options>
</driver-config>
```

7.2 Input and Output Elements

The input and output elements control the contents of the XML documents that are sent between the Identity Vault and the external application. They are child elements to the <nds> element.

Identity Manager is designed to handle entry management. As such, it provides commands which use the Identity Vault verbs that list, read, search, add, remove, modify, and move entries. Partition and replica management are not currently available through Identity Manager.

<input>

Specifies an entry operation to be performed by the receiver.

Description

An <input> element is sent in response to an event that occurs in the Identity Vault or the external application such as modifications to an entry's attributes, the creation of a new entry, or the deletion of an entry.

An <input> element can contain one or more of the commands in the list. Some commands are used only by the driver or the engine:

- The Identity Manager driver sends the association commands (add, modify, and remove) to the Identity Manager engine, but the engine never sends these commands to the Identity Manager driver.
- The Identity Manager engine sends the init-params command to the driver when it is first started by the Identity Manager engine.

Although most input documents will contain only one event or command, style sheets can convert the one event into multiple events. Your driver needs to loop through the document and discover all events, rather than assuming that it needs to find only one.

Definition

```
<!ELEMENT input (add|
                 modify|
                 delete|
                 rename|
                 move|
                 query|
                 query-schema|
                 add-association|
                 modify-association|
                 remove-association|
                 init-params|
                 status)* >
```

Elements

<add>

Specifies an input that creates a new entry in the receiving application.

<modify>

Specifies an input that modifies an entry's attributes.

<delete>

Specifies an input that deletes an entry in the receiving application.

<rename>

Specifies an input that renames the entry in the receiving application.

<move>

Specifies an input that moves an entry from one container to another.

<query>

Specifies an input that retrieves additional information about an entry from the receiving application.

<query-schema>

Specifies an input which returns an XML document that describes the schema of the receiving application.

<add-association>

Specifies an input that adds an association value to the specified entry in the Identity Vault database.

<modify-association>

Specifies an input that modifies the association of the specified entry in the Identity Vault database.

<remove-association>

Specifies an input that removes an association from the specified entry in the Identity Vault database.

<init-params>

Specifies an input that contains initialization parameters for the Identity Manager driver.

<status>

Specifies an input that contains the status of the driver.

Parent

- <nds>

<output>

Specifies a response to an entry operation.

Description

An <output> element is sent in response to an input command. A status command must always be returned in response to an <input> element.

An <output> element can contain one or more of the commands in the list. The association commands (add, modify, and remove) are only sent to the Identity Vault. The Identity Manager driver sends the init-params command to the Identity Manager engine when it needs to store state information.

Definition

```
<!ELEMENT output (status|
                  add-association|
                  modify-association|
                  remove-association|
                  instance|
                  schema-def|
                  init-params)* >
```

Elements

<status>

Specifies an output that is the response to an input.

<add-association>

Specifies an output that adds an association value to the specified entry in the Identity Vault database.

<modify-association>

Specifies an output that modifies the association of the specified entry in the Identity Vault database.

<remove-association>

Specifies an output that removes an association from the specified entry in the Identity Vault database.

<instance>

Specifies an output that contains the requested information about an entry.

<schema-def>

Specifies an output that contains the class and attribute definitions in the Identity Vault schema or the external application's schema.

<init-params>

Specifies an output that contains the Identity Manager driver's state information.

Parent

- <nds>

7.3 Command and Event Elements

This section describes the elements that are used to define Identity Manager operations. These are the elements that define the request and reply documents that are exchanged between the Identity Manager engine and the Identity Manager driver.

When the Identity Manager engine sends a document with an <input> element to the driver, the element is a command. When the driver sends a document with an <input> element to the engine, the element is an event. Essentially, an event and a command have the same syntax, but there are subtle differences. The following elements have command and event sections that describe these differences.

When the driver sends an event to the engine, the driver is informing the engine that something occurred in the application. The engine determines, based on the configurable rules, what commands, if any, to send to the Identity Vault.

When the engine sends a command to the driver, the engine is informing the driver that something occurred in the Identity Vault. The engine has already processed the Identity Vault event as input, applied the appropriate rules, determined the changes that need to be made in the application, and transformed these changes into commands.

The following table describes some of the common attributes that are found in the command and event elements.

Table 7-1

Attributes	Description
association-ref	The reference key which uniquely identifies an entry in the database. This is assigned to attribute values when the value is a distinguished name and that distinguished name has formed an association. It is also assigned to structured attributes when one of the components is a distinguished name and that distinguished name has formed an association.
class-name	The base class which the entry belongs to. The class name is mapped in the mapping rules so that Identity Manager is given the Identity Vault name and the external application is given the class name in its name space.
dest-dn	The distinguished name of the entry in the destination database.
dest-entry-id	The entry ID in the destination database.
event-id	A number which is unique to the XML document and which identifies the command.
src-dn	The distinguished name of the entry in the source database (the database that is the originator of the XML document).
src-entry-id	The entry ID in the source database (used internally by Identity Manager).

<add>

Specifies an input that creates a new entry in the receiver.

Description

The <add> element is an input command or an input event. It is used for following tasks:

- The Identity Manager engine sends an add command to the subscriber shim to request that the external application add an entry.
- The publisher shim sends an add event as notification that an entry has been added in the external application. When the add event is used for event notification, it must also contain an association element.

Definition

```
<!ELEMENT add            (association?, add-attr*, password?)>
<!ATTLIST add
          src-dn          CDATA            #IMPLIED
          src-entry-id    CDATA            #IMPLIED
          dest-dn         CDATA            #IMPLIED
          dest-entry-id   CDATA            #IMPLIED
```

```
          class-name       CDATA            #REQUIRED
          template-dn      CDATA            #IMPLIED
          event-id         CDATA            #IMPLIED>
<!ELEMENT add-attr        (value+)>
<!ATTLIST add-attr
          attr-name        CDATA            #REQUIRED>
<!ELEMENT password         (#PCDATA)>
```

Attributes

src-dn

Specifies the distinguished name of the entry to add, in the name space of the sender. When the Identity Manager engine sends the <add> element, the Identity Manager driver should copy the src-dn attribute to the dest-dn attribute of an <add-association> element.

src-entry-id

Specifies the entry ID of the entry that generated the add event or command. It is specified in the name space of the sender. When the Identity Manager engine sends the <add> element, the Identity Manager driver should copy the src-entry-id attribute to the dest-entry-id attribute of an <add-association> element.

dest-dn

Specifies the distinguished name of the entry in the name space of the receiver. For event notifications, it should be left empty. For commands, it is filled in by the placement rules.

dest-entry-id

Specifies the entry ID of the entry in the name space of the receiver. Used internally by the Identity Manager engine and should be ignored by the driver.

class-name

Specifies the base class of the entry being added.

template-dn

Specifies the distinguished name, in the receiver's name space, of the template to use when creating the entry.

event-id

Specifies an identifier used to identify a particular instance of the command or event.

Elements

<association>

Specifies the unique key of the entry in the external application.

<add-attr>

Specifies the attributes to add with the entry.

<password>

Specifies the initial password for the entry.

Request Format

Command

The Identity Manager engine sends the following attributes and elements in the add command to the subscriber shim:

- src-dn
- dest-dn (if generated by the placement rules)
- class-name
- event-id
- template-dn which specifies the distinguished name, in the receiver's name space, of the template to use when creating the entry
- 0 or more <add-attr> elements
- If 1 or more <add-attr> elements, 1 or more <value> elements as children of the <add-attr> element
- <password> (optional, initial password for the entry)

Event

The publisher shim sends the following attributes in the add command to the Identity Manager engine:

- src-dn
- class-name
- event-id (optional)
- 0 or more <add-attr> elements
- If 1 or more <add-attr> elements, 1 or more <value> elements for each <add-attr> element
- <association>

Reply Format

Command

The subscriber shim must return a status command, and if the add succeeded, the subscriber shim must return an add-association command with the key that uniquely identifies the new entry.

If the entry does not contain values for all the attributes defined in the create rules, Identity Manager discards the add command for the entry. When a modify command is received for this entry, Identity Manager queries the Identity Vault for the missing attributes. If all the attributes now have values, Identity Manager changes the modify into an add command.

Event

The Identity Manager engine returns a status command.

If the add event does not contain values for all the attributes defined in the create rules, the add event fails. When a modify event is received for this entry, the Identity Manager engine queries the publisher shim for the missing attributes. The add event succeeds if the required attributes now have values.

Parent

- <input>

Example

The following example shows an add event from a Identity Manager driver.

```
<add class-name="User" src-dn="\Sam">
     <association>1012</association>
     <add-attr attr-name="cn">
          <value>Sam</value>
     </add-attr>
     <add-attr attr-name="Surname">
          <value>Jones</value>
     </add-attr>
     <add-attr attr-name="Given Name">
          <value>Sam</value>
     </add-attr>
     <add-attr attr-name="Telephone Number">
          <value>555-1212</value>
     </add-attr>
</add>
```

<add-association>

Specifies an input or output that adds an association value to the specified entry in the Identity Vault database.

Description

When the Identity Manager engine sends the driver an add command and the driver succeeds with the add, the driver sends back a success status with an <add-association> element.

The <add-association> element is used to return the unique key that identifies an entry in the external application. The Identity Manager engine never sends an <add-association> element to the external application. The Identity Manager driver can send this <element> to the Identity Manager engine either as an input or output.

Definition

```
<!ELEMENT add-association          (#PCDATA) >
<!ATTLIST add-association
          dest-dn          CDATA            #REQUIRED
          dest-entry-id    CDATA            #IMPLIED
          event-id         CDATA            #IMPLIED>
```

Attributes

dest-dn

Specifies the distinguished name of the entry receiving the association. It should be set to the src-dn of the <add> command.

dest-entry-id

Specifies ID of the entry receiving the association. It should be set to the src-entry-id of the <add> command.

event-id

Specifies an identifier used to identify a particular instance of the command or event. It should be set to the event-id of the <add> command.

Request Format

Event

The Identity Manager driver includes the following with this event:

- dest-dn which is the src-dn of the original add command
- dest-entry-id which is the src-entry-id of the original add command (optional, but encouraged)
- event-id which is the event-id of the original add command
- PCDATA which contains the foreign key that uniquely identifies this entry.

Reply Format

If the event is an input event, the Identity Manager engine returns a status command indicating whether the event was processed successfully.

Parent

- <input>
- <output>

Example

The following example shows an <add-association> element as an output from the driver.

```
<add-association dest-dn="\Users\Samuel" dest-entry-id="33974">{BC3E7155-CDF9-d311-
9846-0008C76B16C2}</add-association>
```

<delete>

Specifies an input that deletes an entry in the target application.

Description

The delete command is an input command or event. It is used for the following tasks:

- The Identity Manager engine sends the delete command to the subscriber shim to request that the external application delete an entry. The delete command must contain an association element.

- The publisher shim sends the delete command as an event notification that an entry has been deleted in the external application. When the delete command is used for event notification, it must contain an association element.

Definition

```
<!ELEMENT delete          (association?)>

<!ATTLIST delete

            src-dn          CDATA           #IMPLIED

            src-entry-id    CDATA           #IMPLIED

            dest-dn         CDATA           #IMPLIED

            dest-entry-id   CDATA           #IMPLIED

            class-name      CDATA           #IMPLIED

            event-id        CDATA           #IMPLIED>
```

Attributes

src-dn

Specifies the distinguished name of the entry to delete, in the name space of the sender.

src-entry-id

Specifies the entry ID of the entry that is being deleted. It is used internally by the Identity Manager engine and should be ignored by the driver.

dest-dn

Specifies the distinguished name of the entry in the name space of the receiver. For events, the driver should leave it empty.

dest-entry-id

Specifies the entry ID for the entry in the name space of the receiver. For events, the driver should leave it empty.

class-name

Specifies the base class of the entry being deleted.

event-id

Specifies an identifier used to identify a particular instance of the command or event.

Elements

<association>

Specifies the unique identifier for the entry in the external application.

Request Format

Command

The Identity Manager engine sends the following information to the subscriber shim with the delete command:

- <association>
- src-dn
- event-id

Event

The publisher shim sends the following information to the Identity Manager engine with a delete event:

- <association>
- src-dn (optional, but encouraged)
- class-name (optional, but encouraged)
- event-id (optional)

Reply Format

The receiving application should respond to a <delete> command or event with a <status> element indicating whether the <delete> was processed successfully.

Parent

- <input>

Example

The following example shows a <delete> element as an input from the driver.

```
<delete class-name="User" src-dn="\Sam">
   <association>1012</association>
</delete>
```

<init-params>

Specifies an input or output that contains the initialization parameters.

Description

As input, the Identity Manager engine uses the <init-params> element to send initialization parameters to the DriverShim, PublicationShim, and SubscriptionShim init routines. As a developer of a Identity Manager driver, you determine which of the optional configuration elements your driver requires. All drivers require a <driver-filter> element to be functional.

The Identity Manager driver uses the <init-params> element to send state information to the Identity Manager engine. The driver can include it in any <input> or <output> element.

Definition

```
<!ELEMENT init-params    (authentication-info?,
                          driver-filter?,
                          driver-options?,
                          subscriber-options?,
```

```
publisher-options?,
driver-state?,
subscriber-state?,
publisher-state?)>
```

Elements

<authentication-info>

Specifies the information required to make a connection and log in to the external application.

<driver-filter>

Specifies which object classes and the attributes that the subscriber shim and publisher shim can synchronize with the Identity Vault.

<driver-options>

Specifies any configuration options which the driver shim requires during initialization and which the system administrator needs to supply values for.

<subscriber-options>

Specifies any configuration options which the subscription shim requires during initialization and which the system administrator needs to supply values for.

<publisher-options>

Specifies any configuration options which the publisher shim requires during initialization and which the system administrator needs to supply values for.

<driver-state>

Returns the driver state information that the Identity Manager driver saved when it was shut down.

<subscriber-state>

Returns the subscriber state information that the Identity Manager driver saved when it was shut down.

<publisher-state>

Returns the publisher state information that the Identity Manager driver saved when it was shut down.

Request Format

Command

The command may include any of the following optional elements:

- <authentication-info> which is sent to the DriverShim, PublicationShim, and SubscriptionShim
- <driver-filter> which is sent to the PublicationShim and the SubscriptionShim

- <driver-options> which is sent to the DriverShim init routine
- <subscriber-options> which is sent to the SubscriptionShim init routine
- <publisher-options> which is sent to the PublicationShim init routine
- <driver-state> which is sent to the DriverShim init routine
- <subscriber-state> which is sent to the SubscriptionShim init routine
- <publisher-state> which is sent to the PublicationShim init routine

Reply Format

The DriverShim, PublicationShim, and SubscriptionShim return a <status> element indicating whether the command was processed successfully.

The Identity Manager driver can also send with any <output> element an <init-params> element that contains <driver-state>, <publisher-state>, and <subscriber-state> elements.

Parent

- <input>
- <output>

Example

The following example shows what is sent as an input to the DriverShim, SubscriptionShim, and PublisherShim init methods as well as what is sent to the DriverShim getSchema method.

```
<!-- for DriverShim.init() -->
<init-params>
   <authentication-info>
      <server>localhost</server>
      <user>Fred</user>
      <password>foobar</password>
   </authentication-info>
   <driver-options>
       <!-- some driver defined driver options -->
   </driver-options>
   <driver-state>
       <!-- some driver defined driver state -->
   </driver-state>
</init-params>
```

```
<!-- for SubscriptionShim.init() -->
<init-params>
   <authentication-info>
      <server>localhost</server>
      <user>Fred</user>
      <password>foobar</password>
   </authentication-info>
   <driver-filter type="subscriber">
      <allow-class class-name="User">
         <allow-attr attr-name="Telephone Number"/>
         <allow-attr attr-name="CN"/>
         <allow-attr attr-name="Surname"/>
         <allow-attr attr-name="Given Name"/>
         <allow-attr attr-name="Description"/>
         <allow-attr attr-name="Title"/>

         <allow-attr attr-name="Postal Address"/>
         <allow-attr attr-name="GUID"/>
         <allow-attr attr-name="Full Name"/>
      </allow-class>
      <allow-class class-name="Organizational Unit">
         <allow-attr attr-name="OU"/>
         </allow-class>
         <allow-class class-name="Organizational">
         <allow-attr attr-name="O"/>
      </allow-class>
   </driver-filter>
   <subscriber-options>
      <!-- some driver defined subscriber options -->
   </subscriber-options>
   <subscriber-state>
      <!-- some driver defined subscriber state -->
   </subscriber-state>
</init-params>

<!-- for PublicationShim.init() -->
<init-params>
   <authentication-info>
      <server>localhost</server>
```

```
        <user>Fred</user>
        <password>foobar</password>
    </authentication-info>
    <driver-filter type="publisher">
        <allow-class class-name="User">
            <allow-attr attr-name="Telephone Number"/>
            <allow-attr attr-name="CN"/>
            <allow-attr attr-name="Surname"/>
            <allow-attr attr-name="Given Name"/>
            <allow-attr attr-name="Description"/>
            <allow-attr attr-name="Title"/>
            <allow-attr attr-name="Postal Address"/>
            <allow-attr attr-name="GUID"/>
            <allow-attr attr-name="Full Name"/>
        </allow-class>
        <allow-class class-name="Organizational Unit">
            <allow-attr attr-name="OU"/>
        </allow-class>
        <allow-class class-name="Organizational">
            <allow-attr attr-name="O"/>
        </allow-class>
    </driver-filter>
    <publisher-options>
        <!-- some driver defined publisher options -->
    </publisher-options>
    <publisher-state>
        <!-- some driver defined publisher state -->
    </publisher-state>
</init-params>

<!-- for DriverShim.getSchema() -->
<init-params>
    <authentication-info>
        <server>localhost</server>
        <user>Fred</user>
        <password>foobar</password>
    </authentication-info>
    <driver-filter type="subscriber">
        <allow-class class-name="User">
```

```
            <allow-attr attr-name="Telephone Number"/>
            <allow-attr attr-name="CN"/>
            <allow-attr attr-name="Surname"/>
            <allow-attr attr-name="Given Name"/>
            <allow-attr attr-name="Description"/>
            <allow-attr attr-name="Title"/>
            <allow-attr attr-name="Postal Address"/>
            <allow-attr attr-name="GUID"/>
            <allow-attr attr-name="Full Name"/>
         </allow-class>
         <allow-class class-name="Organizational Unit">
            <allow-attr attr-name="OU"/>
         </allow-class>
         <allow-class class-name="Organizational">
            <allow-attr attr-name="O"/>
         </allow-class>
      </driver-filter>
      <driver-filter type="publisher">
         <allow-class class-name="User">
            <allow-attr attr-name="Telephone Number"/>
            <allow-attr attr-name="CN"/>
            <allow-attr attr-name="Surname"/>
            <allow-attr attr-name="Given Name"/>
            <allow-attr attr-name="Description"/>
            <allow-attr attr-name="Title"/>
            <allow-attr attr-name="Postal Address"/>
            <allow-attr attr-name="GUID"/>
            <allow-attr attr-name="Full Name"/>
         </allow-class>
         <allow-class class-name="Organizational Unit">
            <allow-attr attr-name="OU"/>
         </allow-class>
         <allow-class class-name="Organizational">
            <allow-attr attr-name="O"/>
         </allow-class>
      </driver-filter>
      <driver-options>
         <!-- some driver defined driver options -->
      </driver-options>
```

```
    <subscriber-options>
        <!-- some driver defined subscriber options -->
    </subscriber-options>
    <publisher-options>
        <!-- some driver defined publisher options -->
    </publisher-options>
    <driver-state>
        <!-- some driver defined driver state -->
    </driver-state>
    <subscriber-state>
        <!-- some driver defined subscriber state -->
    </subscriber-state>
    <publisher-state>
        <!-- some driver defined publisher state -->
    </publisher-state>
</init-params>
```

<instance>

Specifies an output that contains the requested information about an entry.

Description

The <instance> element is an output that is sent with the status command as a reply to a query command.

An <instance> element is returned for each entry that matches the search filter of the query.

Definition

```
<!ELEMENT instance      (association?, parent?, attr*)>
<!ATTLIST instance
          src-dn         CDATA            #IMPLIED
          src-entry-id   CDATA            #IMPLIED
          class-name     CDATA            #REQUIRED
          event-id       CDATA            #IMPLIED>
```

Attributes

src-dn

Specifies the distinguished name of the entry being returned in the name space of the sender.

src-entry-id

Specifies the entry ID of the entry being returned in the name space of the sender. It should be ignored by the driver.

class-name

Specifies the base class of the entry being returned.

event-id

Specifies the event-id of the query, if the query contained an <event-id> element.

Elements

<association>

Specifies the unique identifier for the entry being returned.

<parent>

Specifies the parent container of the entry, if requested by the query.

<attr>

Specifies attributes and values, if the query requested them

Reply Format

When the Identity Manager driver returns an <instance> element, the element must contain an <association> element.

When the Identity Manager engine returns an <instance> element, the element contains an <association> element if one has been formed for the entry.

Parent

- <output>

Example

The following example illustrates an <instance> element.

```
<instance class-name="User" src-dn="\Users\Samuel">
   <association>1012</association>
   <attr attr-name="Surname">
      <value>Jones</value>
   </attr>
```

```
   <attr attr-name="cn">
      <value>Samuel</value>
   </attr>
   <attr attr-name="Given Name">
      <value>Samuel</value>
   </attr>
   <attr attr-name="Telephone Number">
      <value>555-1212</value>
      <value>555-1764</value>
   </attr>
</instance>
```

<modify>

Specifies an input that modifies an entry's attributes.

Description

The <modify> element is an input command or event. It is used for the following tasks:

- The Identity Manager engine sends a modify command to the subscriber shim to request that the external application modify an entry. The modify command must contain an association element.
- The publisher shim sends a modify event as notification that an entry has been modified in the external application. When the <modify> element is used for event notification, it must contain an <association> element.

Definition

```
<!ELEMENT modify          (association?, modify-attr+)>
<!ATTLIST modify
           src-dn          CDATA            #IMPLIED
           src-entry-id    CDATA            #IMPLIED
           dest-dn         CDATA            #IMPLIED
           dest-entry-id   CDATA            #IMPLIED
           class-name      CDATA            #IMPLIED
           event-id        CDATA            #IMPLIED>
```

Attributes

src-dn

Specifies the distinguished name of the entry to modify in the name space of the sender.

src-entry-id

Specifies the entry ID in the name space of the sender. It is used internally by the Identity Manager engine and should be ignored by the driver.

dest-dn

Specifies the distinguished name of the entry in the name space of the receiver. It is used internally by the Identity Manager engine and should be ignored by the driver.

dest-entry-id

Specifies the entry ID of the entry in the name space of the receiver. It is used internally by the Identity Manager engine and should be ignored by the driver.

class-name

Specifies the base class of the entry being modified. This attribute is required for modify events.

event-id

Specifies an identifier used to identify a particular instance of the command or event.

Elements

<association>

Specifies the unique identifier for the entry in the external application. This element is required for modify events.

<modify-attr>

Specifies the attributes to modify.

Request Format

Command

The Identity Manager engine sends the following information with a modify command to the subscriber shim:

- src-dn
- class-name
- event-id
- <association>
- <modify-attr> (1 or more)
    - For <remove-value>, 1 or more <value>
    - For <add-value>, 1 or more <value>

    - For <remove-all-values>, no values

Event

The publisher shim sends the following information with a modify event to the Identity Manager engine:

- src-dn
- class-name (required because a modify can turn into an add)
- event-id (optional)
- <association>
- 1 or more <modify-attr> elements
    - For <add-value>, 1 or more <value> elements
    - For <remove-value,> 1 or more <value> elements
    - For <remove-all-values>, no values

Reply Format

The receiving application should respond to a modify command or event with a status command indicating whether the modify was processed successfully.

Parent

- <input>

Example

The following example illustrates a <modify> element.

```
<modify class-name="User" src-dn="\Sam">
   <association>1012</association>
   <modify-attr attr-name="Given Name">
      <remove-all-values/>
      <add-value>
         <value>Samuel</value>
      </add-value>
   </modify-attr>
   <modify-attr attr-name="Telephone Number">
      <remove-value>
```

```
            <value>555-1212</value>
        </remove-value>
        <add-value>
            <value>555-1764</value>
            <value>555-1765</value>
        </add-value>
    </modify-attr>
</modify>
```

<modify-association>

Specifies an input or output that modifies the association of the specified entry.

Description

The modify-association event is used to notify the Identity Manager engine that the entry's unique key in the external application has been modified. The Identity Manager engine never sends a modify-association command to the external application. The Identity Manager driver can send this event to the Identity Manager engine either as an input or output event.

The Identity Manager driver should send this event whenever a foreign key in the external application changes for an entry that passes the event filter for either the subscriber shim or the publisher shim. For example, if the foreign key is the distinguished name of the entry and the entry's dn changes, the driver sends both a rename event and a modify-association event.

Definition

```
<!ELEMENT modify-association    (association, association) >
<!ATTLIST modify-association
           event-id         CDATA            #IMPLIED>
```

Attributes

event-id

Specifies an identifier used to identify a particular instance of the command or event.

Elements

<association>

Specifies the old association value.

<association>

Specifies the new association value.

Request Format

Event

The Identity Manager driver must include the following with this event:

- <association> with the old key
- <association> with the new key
- event-id (optional)

Reply Format

If the event is an input event, the Identity Manager engine returns a status command indicating whether the event was processed successfully.

Parent

- <input>
- <output>

Example

The following example shows a <modify-association> element sent by a driver.

```
<modify-association>
   <association>{BC3E7155-CDF9-d311-9846-0008C76B16C2}</association>
   <association>{CD3F7155-DE09-e311-9846-0008D76C16D2}</association>
</modify-association>
```

<modify-password>

Modifies an attribute password. Added in Identity Manager 1.1.

Description

<modify-password> is used:

- As an event notification from the PublicationShim to Identity Manager that on object password was modified in the application. When used as a notification, an <association> is required.

- As a command from Identity Manager to the SubsciptionShim to modify an object password in the application. When used as a command, an <association> is required and is the unique key of the object to modify.

When the target is NDS, and <old-password> is specified, the modifyPassword API is used to modify the password. If not specified, the GenerateKeyPair API is used. Note that using GenerateKeyPair may invalidate authentication credentials for any existing session authenticated as the target object.

When the target is the application, a driver may or may not implement this functionality, depending on the applicability to the application.

A response to <modify-password> should be a <status> indicating whether or not the <modify-password> was processed successfully.

Note that if the old password is not provided to change the password the Identity Manager driver must have supervisor rights to the object.

Definition

```
<!ELEMENT modify-password  (association ?,
                            old-password ?,
                            password)>
<!ATTLIST modify-attr
           src-dn          CDATA            #IMPLIED>
           src-entry       CDATA            #IMPLIED>
           dest-dn         CDATA            #IMPLIED>
           dest-entry      CDATA            #IMPLIED>
           class-name      CDATA            #IMPLIED>
           event-id        CDATA            #IMPLIED>
           timestamp       CDATA            #IMPLIED>

<!ELEMENT old-password  #PCDATA>
```

Attributes

src-dn

The distinguished name of source object that generated the event in the namespace of the sender.

src-entry-id

The entry id of source object that generated the event in the namespace of the sender. (Reserved. Should be ignored by the driver.)

dest-dn

The distinguished name of the target object in the namespace of the receiver.

dest-entry-id

The entry id of the target object in the namespace of the receiver. (Reserved. Should be ignored by the driver.)

class-name

Required when used as a notification. The name of the base class of the object. The class name is mapped between the application and NDS name spaces by the schema mapping rule so that Identity Manager will see the name in the NDS namespace and a driver will see the name in the application name space.

event-id

An identifier used to tag the results of an event or command.

timestamp

(Reserved. Should be ignored by the driver.)

Elements

<old-password>

Specifies the current password.

Parent

- <input>

<move>

Specifies an input that moves an entry from one container to another.

Description

The move command is an input command or event. It is used for the following tasks:

- The Identity Manager engine sends the move command to the subscriber shim to request that the external application move an entry from one container to another. The move command must contain an <association> element.
- The publisher shim sends a move event as notification that an entry has been moved to a different container in the external application. When the move command is used for event notification, it must contain an <association> element.

Definition

```
<!ELEMENT move             (association?, parent)>
<!ATTLIST move
          src-dn            CDATA            #IMPLIED
          src-entry-id      CDATA            #IMPLIED
          dest-dn           CDATA            #IMPLIED
          dest-entry-id     CDATA            #IMPLIED
          old-src-dn        CDATA            #IMPLIED
```

```
class-name      CDATA           #IMPLIED
event-id        CDATA           #IMPLIED>
```

Attributes

src-dn

Specifies the distinguished name of the entry, after the move, in the name space of the sender.

src-entry-id

Specifies the entry ID of the entry in the name space of the sender. It is used internally by the Identity Manager engine and should be ignored by the driver.

dest-dn

Specifies the distinguished name of the entry in the name space of the receiver. It is used internally by the Identity Manager engine and should be ignored by the driver.

dest-entry-id

Specifies the entry ID of the entry in the name space of the receiver. It is used internally by the Identity Manager engine and should be ignored by the driver.

old-src-name

Specifies the distinguished name of the entry, before the move, in the name space of the sender.

class-name

Specifies the base class of the entry being moved.

event-id

Specifies an identifier used to identify a particular instance of the command or event.

Elements

<association>

Specifies the unique identifier for the entry in the external application.

<parent>

Specifies the new container for the entry.

Request Format

Command

The Identity Manager engine sends the following information with a move command to the subscriber shim:

- src-dn which is the new distinguished name after the move
- old-src-dn which is the name before the move
- class-name
- event-id
- <association>
- <parent>
    - src-dn which is the parent name after the move
    - <association> of the parent if one exists. If one doesn't exist, the subscriber shim should return a status level of warning and not move the entry.

Event

The driver sends the following information with a move event to the Identity Manager engine:

- class-name
- event-id (optional)
- <association> of the entry to move
- <parent>
    - <association> of the parent

Reply Format

The receiving application should respond to the move command with a status command indicating whether the move was processed successfully.

Parent

- <input>

Example

The following example illustrates a move command sent by the driver to the Identity Manager engine.

```
<move class-name="User" src-dn="\Users\Samuel" old-src-dn="\Samuel">
```

```
   <association>1012</association>
   <parent src-dn="\Users\">
      <association>1013</association>
   </parent>
</move>
```

<query>

Specifies an input that retrieves additional information about an entry from the target application.

Description

The query command is an input command or event. Both the publisher and the subscriber must implement all defined possibilities. A query is used to find and read information about entries in the Identity Vault and the external application.

*IMPORTANT:*Full functionality for Identity Manager rules and processing depends upon the full implementation of the query command by the Identity Manager driver.

Definition

```
<!ELEMENT query    (association?,
                      (search-class |
                       search-attr |
                       read-attr |
                       read-parent)*)>
<!ATTLIST query
          dest-dn          CDATA               #IMPLIED
          dest-entry-id    CDATA               #IMPLIED
          class-name       CDATA               #IMPLIED
          scope            (%Search-Scope;)    "subtree"
          event-id         CDATA               #IMPLIED>

<!ELEMENT search-class    EMPTY>
<!ATTLIST search-class
          class-name       CDATA               #REQUIRED>

<!ELEMENT search-attr     (value)+ >
<!ATTLIST search-attr
          attr-name        CDATA               #REQUIRED>
```

```
<!ELEMENT read-attr        EMPTY>
<!ATTLIST read-attr
          attr-name        CDATA              #IMPLIED
          type      (%Read-attr-type;)        "default">

<!ELEMENT read-parent       EMPTY>
```

Attributes

dest-dn

Specifies the distinguished name for the starting point for the search. If both the dest-dn attribute and <association> have values, the <association> value is used as the starting point for the search. If neither have values, the search begins at the root of the directory.

dest-entry-id

Specifies the entry ID in the name space of the receiver. It is used internally by the Identity Manager engine and should be ignored by the driver.

class-name

Specifies the base class of the dest-dn attribute.

scope

Specifies the extent of the search. This attribute supports the following values:

- subtree —indicates to search the base entry and all entries in its branch of the directory tree. If no scope is specified, subtree is used as the default value.

- subordinates —indicates to search the immediate subordinates of the base entry (the base entry is not searched).

- entry— indicates to search just the base entry.

For scopes other than entry, the selected entries can be further limited by the <search-class> and <search-attr> elements. For scopes of entry, the <search-class> and <search-attr> elements are ignored.

event-id

Specifies an identifier used to identify a particular instance of the command or event.

Elements

<association>

Specifies the unique identifier for the entry where the search begins. If both the dest-dn attribute and <association> have values, the <association> value is used as the starting point for the search. If neither have values, the search begins at the root of the directory.

<search-class>

Specifies the search filter for object classes. If the query contains no <search-class> elements, all entries matching the scope and the <search-attr> elements are returned.

<search-attr>

Specifies the search filter for attribute values. If more than one <search-attr> element is specified, the entry must match all attributes to be returned.

<read-attr>

Specifies which attribute values are returned with entries that match the search filters. If no attributes are specified, all attributes are returned. If a <read-attr> element is specified without an attr-name attribute, no attributes are returned.

<read-parent>

Specifies whether the parent of the entry is returned with the entry.

Request Format

A query can include any of the following defined elements and attributes:

- dest-dn or <association> which sets the base, or starting point, for the search.
- dest-entry-id (used internally by Identity Manager)
- scope
- class-name
- event-id
- <search-class>
- <search-attr>
- <read-attr>
- <read-parent>

Whenever the Identity Manager engine sends your driver a query, your driver should be prepared to parse the query for all of the above elements and attributes.

Whenever your driver sends the Identity Manager engine a query, your driver should be prepared to include all of the above element and attributes except the dest-entry-id attribute.

For possible entry points when the Identity Manager engine or the Identity Manager driver sends a query, see Section 4.5, Event Processing.

Reply Format

The receiving application should respond to a query with an instance command for each entry returned. The response should also include a status command indicating whether the query was processed successfully. A query should return a successful status even when no entries exist that match the search criteria.

Parent

- <input>
- <config-object>

Remarks

A query is search operation. When the query is searching for a match, the following elements and attributes are used to specify matching criteria:

- The scope determines whether the search includes just the entry, the entries in one container, or an entire branch of the directory tree.
- The <search-class> element specifies the base class the entry must match. If no <search-class> elements are specified, all entries in the scope are selected as matched.
- The <search-attr> element specifies the attribute and value an entry must match. If more than one value is specified for an attribute, the entry must match all values. If more than one attribute is specified, the entry must match all attributes.

A query is a read operation. When the query is requesting entry and attribute information, the following elements and attributes are used to specify the information returned.

- The scope, <search-class>, and <search-attr> tags determine which entries are returned.
- The <read-attr> element determines which attributes and values are returned. If no <read-attr> elements are specified, all attributes of the entry are returned. If a single <read-attr> element is specified without an attr-name attribute, no attributes are returned.
- The <read-parent> element determines whether the entry's parent is returned.

Example

The following examples illustrate two queries which search different parts of the directory and return different information about the entry.

```
<!-- Example #1                                       -->
<!-- Search the whole application for a User entry with -->
<!-- the Surname of Jones.                            -->
<!-- Don’t read any attributes, but read the parent entry-->
<query class-name="User" event-id="0" scope="subtree">
   <search-class class-name="User"/>


   <search-attr attr-name="Surname">
      <value type="string">Jones</value>
   </search-attr>
   <read-attr/>
   <read-parent/>
</query>
<!-- Example #2                                  -->
<!-- Read the User entry whose foreign key is 1011 -->
<!-- Read the Surname, cn, Given Name and         -->
<!-- Telephone Number attributes                  -->
<query class-name="User" event-id="1" scope="entry">
        <association>1011</association>
```

```
        <read-attr attr-name="Surname"/>
        <read-attr attr-name="cn"/>
        <read-attr attr-name="Given Name"/>
        <read-attr attr-name="Telephone Number"/>
</query>
```

<query-schema>

Specifies an input which returns an XML document that describes the schema of the target application.

Description

The query-schema command is an input command or event. It is used to read the schema definition from the Identity Vault or the external application. It includes only one piece of data, an event-id which identifies the command. Currently this is an optional command, but if it is not implemented, rule configuration is more difficult.

The Identity Manager engine has implemented this command so that

- The Identity Manager driver and iManager can query the Identity Vault and receive back an XML schema-def document that defines the Identity Vault schema.
- The Identity Manager engine and iManager can query the Identity Manager driver and receive back an XML schema-def document that defines the schema of the external application.

Definition

```
<!ELEMENT query-schema       EMPTY>
<!ATTLIST query-schema
          event-id           CDATA          #IMPLIED>
```

Attributes

event-id

Specifies an identifier used to identify a particular instance of the command or event.

Request Format

The request command or event includes the event-id attribute.

Reply Format

The receiving application should respond to a query-schema command with a schema-def document and a status command indicating whether the command was processed successfully.

Parent

- <input>

Remarks

If the Identity Manager driver implements this command, the Identity Manager engine queries the driver for its schema and stores the returned XML document in the DirXML-ApplicationSchema attribute of the driver object. The DriverShim init method should include procedures to update the DirXML-ApplicationSchema attribute when the schema of the external application is modified.

For more information on the format of the schema-def document, see <schema-def>.

Example

```
<query-schema/>
```

<remove-association>

Specifies an input or output that removes an association from the specified entry.

Description

The remove-association command notifies the Identity Manager engine that a particular unique key is not valid. The Identity Manager engine never sends a remove-association command to the external application. The Identity Manager driver sends this command to the Identity Manager engine either as an input or output command.

This command has two main purposes. The first allows an event transformation to change a delete command into a remove-association command. When an entry is deleted in an external application, a system administrator may not want that entry deleted from the Identity Vault. When that is the case, an event transformation rule needs to convert the delete command to a remove-association command.

The other use for this command is for the following condition that shouldn't occur, but might. If the Identity Manager engine sends the driver an association that doesn't match anything in its external application, the driver can return a command to remove the bogus association.

Definition

```
<!ELEMENT remove-association    (#PCDATA) >
<!ATTLIST remove-association
          event-id              CDATA            #IMPLIED>
```

Attributes

event-id

Specifies an identifier used to identify a particular instance of the command or event.

Request Format

Event

The remove-association event must include the following:

- event-id
- PCDATA which contains the key to remove

Reply Format

If the event is an input event, the Identity Manager engine returns a status command indicating whether the event was processed successfully.

Parent

- <input>
- <output>

Example

The following example shows a remove-association event sent by the driver.

```
<remove-association>{BC3E7155-CDF9-d311-9846-0008C76B16C2}</remove-association>
```

<rename>

Specifies an input that renames an entry in the target application.

Description

The rename command is an input command or event. It renames the entry; it cannot move an entry from one container to another in a hierarchical database. It is used for the following tasks:

- The Identity Manager engine sends the rename command to the subscriber shim to request that the external application rename an entry. The rename command must contain an association element.
- The publisher shim sends the rename command as an event notification that an entry has been renamed in the external application. When the rename command is used for event notification, it must contain an association element.

Definition

```
<!ELEMENT rename          (association?, new-name)>
<!ATTLIST rename
          src-dn           CDATA            #IMPLIED
          src-entry-id     CDATA            #IMPLIED
          dest-dn          CDATA            #IMPLIED
          dest-entry-id    CDATA            #IMPLIED
          old-src-dn       CDATA            #IMPLIED
          remove-old-name  (%Boolean;)      "true"
          class-name       CDATA            #IMPLIED
          event-id         CDATA            #IMPLIED>

<!ELEMENT new-name         (#PCDATA)>
```

Attributes

src-dn

Specifies the new distinguished name of the entry in the name space of the sender.

src-entry-id

Specifies the entry ID of the entry in the name space of the sender. It is used internally by the Identity Manager engine and should be ignored by the driver.

dest-dn

Specifies the distinguished name of the entry in the name space of the receiver. It is used internally by the Identity Manager engine and should be ignored by the driver.

dest-entry-id

Specifies the entry ID of the entry in the name space of the receiver. It is used internally by the Identity Manager engine and should be ignored by the driver.

old-src-dn

Specifies the old distinguished name of the entry in the name space of the sender.

remove-old-name

Specifies whether the old relative distinguished name should be deleted or retained. If not specified, defaults to "true" which removes

the old name.

class-name

Specifies the base class of the entry being renamed.

event-id

Specifies an identifier used to identify a particular instance of the command or event.

Elements

<association>

Specifies the unique identifier for the entry in the external application.

<new-name>

Specifies the new relative distinguished name for the entry.

Request Format

Command

The Identity Manager engine sends the following information with the rename command to the subscriber shim:

- <association>
- src-dn which is the new distinguished name after the change
- old-src-dn which is the distinguished name before the change
- remove-old-name which determines whether the old name value is retained. This may not be relevant to a particular external application.
- class-name
- event-id
- <new-name> which is the new relative distinguished name of the entry

Event

The publisher shim sends the following information with a rename command to the Identity Manager engine:

- <association>
- src-dn which is the new distinguished name after the change
- remove-old-name which determines whether the old name value is retained. If not specified, defaults to removing the old name value.

- class-name
- event-id (optional)
- <new-name> which is the new relative distinguished name of the entry

Reply Format

The receiving application should respond to a rename command with a status command indicating whether the rename was processed successfully.

Parent

- <input>

Example

```
<rename class-name="User" src-dn="\Samuel" old-src-dn="\Sam">
   <association>1012</association>
   <new-name>Samuel</new-name>
</rename>
```

<schema-def>

Specifies an output that contains the class and attribute definitions in the Identity Vault schema or the external application's schema.

Description

The <schema-def> element is part of the reply to a <query-schema> element and to the DriverShim getSchema method. The reply includes the <status> and then the <schema-def> element.

The <schema-def> element defines the available classes and attributes in the database. iManager uses this file to display the schema definitions to administrators so that they can create mapping rules by clicking on the displayed classes and attributes.

Definition

```
<!ELEMENT schema-def       (class-def)* >
<!ATTLIST schema-def
          hierarchical     (%Boolean;)      "true">
          application-name  CDATA           #IMPLIED>
```

```
<!ELEMENT class-def        (attr-def)* >
<!ATTLIST class-def
          class-name       CDATA            #REQUIRED
          asn1id           CDATA            #IMPLIED
          container        (%Boolean;)      "false">

<!ELEMENT attr-def         EMPTY>
<!ATTLIST attr-def
          attr-name        CDATA            #REQUIRED
          asn1id           CDATA            #IMPLIED
          type             (%Attr-type;)    "string"
          required         (%Boolean;)      "false"
          naming           (%Boolean;)      "false"
          multi-valued     (%Boolean;)      "true"
          case-sensitive   (%Boolean;)      "false"
          read-only        (%Boolean;)      "false">
```

Attributes

hierarchical

Specifies whether the data is stored in a hierarchical structure. If not specified, defaults to a hierarchical structure.

application-name

Specifies the name of the application that uses the schema.

Elements

<class-def>

Specifies a class definition for the originating schema.

<attr-def>

Specifies an attribute definition for the parent <class-def> element.

Required Elements

The following elements and attributes are required to define an object class.

```
<schema-def>
   <class-def class-name="xxx">
```

```
        <attr-def attr-name="aaa"/>
    </class-def>
</schema-def>
```

All other class-def and attr-def attributes are optional. The XML documents for schema definitions can become very large. Therefore, if the default values for the XML attributes match the attribute or class schema definition, do not specify them in the XML document.

Parent

- <output>

Remarks

The schema for the external application is not read dynamically. During initial set up, Identity Manager sends a schema read operation to your driver and stores the returned XML document. If the schema in your application changes, your driver will need to be stopped and the new XML schema document needs to be sent to the Identity Manager engine during initialization.

The <schema-def> element does not need to include all class and attribute definitions; it needs to include all class definitions for the entries that the Identity Vault and the external application will synchronize. The class definition does not need to include all attribute definitions; it needs to include the attribute definitions for the attributes that the Identity Vault and the external application will synchronize.

Example

The following example displays a definition for four classes (Organization, Organizational Unit, User, and Bogus). Most of the <attr-def> elements have more information than fits on one line, so the lines are wrapped and indented. Such wrapped lines do not begin with a <.

```
schema-def hierarchical="true">
   <class-def class-name="Organization" container="true">
      <attr-def attr-name="Name" case-sensitive="false" multi-valued="false"
             naming="true" read-only="false" required="false" type="string"/>
      <attr-def attr-name="Object Path" case-sensitive="false"
             multi-valued="false" naming="false" read-only="false"
             required="true" type="string"/>
      <attr-def attr-name="Unique Id" case-sensitive="false"
             multi-valued="false" naming="false" read-only="false"
             required="true" type="string"/>
   </class-def>

   <class-def class-name="Organizational Unit" container="true">
      <attr-def attr-name="Name" case-sensitive="false" multi-valued="false"
             naming="true" read-only="false" required="false" type="string"/>
      <attr-def attr-name="Object Path" case-sensitive="false"
             multi-valued="false" naming="false" read-only="false"
```

```
            required="true" type="string"/>
        <attr-def attr-name="Unique Id" case-sensitive="false"
                multi-valued="false" naming="false" read-only="false"
                required="true" type="string"/>
    </class-def>

    <class-def class-name="User" container="false">
        <attr-def attr-name="cn" case-sensitive="false" multi-valued="false"
                naming="true" read-only="false" required="true" type="string"/>
        <attr-def attr-name="Surname" case-sensitive="false" multi-valued="false"
                naming="false" read-only="false" required="false"
                type="string"/>
        <attr-def attr-name="Given Name" case-sensitive="false"
                multi-valued="false" naming="false" read-only="false"
                required="false" type="string"/>
        <attr-def attr-name="Telephone Number" case-sensitive="false"
                multi-valued="true" naming="false" read-only="false"
                required="false" type="string"/>
        <attr-def attr-name="Object Path" case-sensitive="false"
                multi-valued="false" naming="false" read-only="false"
                required="true" type="string"/>
        <attr-def attr-name="Unique Id" case-sensitive="false"
                multi-valued="false" naming="false" read-only="false"
                required="true" type="string"/>
    </class-def>

    <class-def class-name="Bogus" container="false">
        <attr-def attr-name="Whatever" case-sensitive="false" multi-valued="true"
                naming="true" read-only="false" required="false" type="string"/>
        <attr-def attr-name="Object Path" case-sensitive="false"
                multi-valued="false" naming="false" read-only="false"
                required="true" type="string"/>
        <attr-def attr-name="Unique Id" case-sensitive="false"
                multi-valued="false" naming="false" read-only="false"
                required="true" type="string"/>
    </class-def>
</schema-def>
```

A more complete document for the Identity Vault schema is found in the schema.xml file. Because the Identity Vault schema is extensible, this file does not include all possible attributes for a class. iManager reads the Identity Vault schema dynamically whenever it displays the classes and attributes.

<status>

Specifies either an output that is the response to an input or an input that contains the status of the driver.

Description

The <status> element can be sent under the following conditions:

- As a reply to a command or event. More than one <status> element can be returned for a command or event.
- As a log message in an input event. The Identity Manager driver sends the status to the Identity Manager engine whenever the driver wants to log the driver's status in the Identity Manager log.

Definition

```
<!ELEMENT status          ANY >
<!ATTLIST status
          level           (%Status-Level;)    #REQUIRED
          event-id         CDATA              #IMPLIED>
```

Attributes

level

Specifies the level of success of the command or event. It uses the following flags:

• fatal—indicates that the driver should be shutdown

• error—indicates that the operation did not succeed

• warning—indicates that the operation succeeded but that an warning was logged.

• success—indicates the operation succeeded

• retry—indicates that the application was not responding and therefore the operation should be rescheduled for another time

event-id

If the level is error or warning, specifies the event-id of the input command and the <status> element contains text that explains the warning or error.

Parent

- <input>
- <output>

Example

The following example illustrates two <status> elements.

```
<!-- Example #1                                    -->

<status event-id="0" level="success"/>


<!-- Example #2                                        -->

<status event-id="0" level="warning">Objects in the rear view mirror may appear
closer than they are!</status>
```

7.4 Other Elements

The elements described in this section are child elements of the command and event elements, the nds element, or the rule elements.

<add-attr>

Specifies the attributes and values to add.

Definition

```
<!ELEMENT add-attr        (value+)>

<!ATTLIST add-attr

          attr-name        CDATA            #REQUIRED>
```

Attributes

attr-name

Specifies the name of the attribute.

Elements

<value>

Specifies the attribute's value.

Parent

<add>

<add-value>

Specifies the values to add.

Definition

```
<!ELEMENT add-value          (value+)>
```

Elements

<value>

Specifies the attribute value to add.

Parent

<modify-attr>

<allow-attr>

Specifies which attributes in the class are allowed in the event filter.

Definition

```
<!ELEMENT allow-attr      EMPTY>
<!ATTLIST allow-attr
          attr-name       CDATA              #REQUIRED>
```

Attributes

attr-name

Specifies the name of the attribute.

Parent

<allow-class>

<allow-class>

Specifies the classes that are allowed by the event filter.

Definition

```
<!ELEMENT allow-class     (allow-attr)* >
<!ATTLIST allow-class
          class-name      CDATA              #REQUIRED>
```

Attributes

class-name

Specifies the name of the class.

Parent

<driver-filter>

<association>

Specifies the key which uniquely identifies an entry in the external application.

Description

The association can be any character string such as the entry's distinguished name or a number. The Identity Manager engine stores the association in the DirXML-Associations attribute of the Identity Vault entry. It is used for the following tasks:

- The publisher shim sends the <association> element to the Identity Manager engine as an event notification when an entry in the external application has been modified, moved, renamed, deleted, or added.
- The Identity Manager engine sends the <association> element to the subscriber shim with a add, delete, modify, move, or rename command.
- Both the Identity Manager engine and the Identity Manager driver send the <association> element to specify the base object of a

query command.

- Both the Identity Manager engine and the Identity Manager driver send the <association> element when returning <instance> elements that match a query.

Definition

```
<!ELEMENT association      (#PCDATA)>
<!ATTLIST association
          state            (%Assoc-State;)      #IMPLIED
```

Attributes

state

Reserved, used internally by the Identity Manager engine.

Parent

- <add>
- <delete>
- <instance>
- <modify>
- <modify-association>
- <move>
- <query>
- <parent>
- <rename>

<attr>

Specifies the attributes and values requested by the query. Each attribute should contain at least one value.

Definition

```
<!ELEMENT attr            (value*)>
<!ATTLIST attr
          attr-name        CDATA       #REQUIRED>
```

Attributes

attr-name

Specifies the attribute's name. The engine specifies the name in the Identity Vault name space and the driver specifies the name in the

application's name space.

Elements

<value>

Specifies the attribute's value.

Parent

<instance>

<attr-def>

Specifies an attribute definition.

Definition

```
<!ELEMENT attr-def          EMPTY>
<!ATTLIST attr-def
          attr-name         CDATA             #REQUIRED
          asn1id            CDATA             #IMPLIED
          type              (%Attr-type;)     "string"
          required          (%Boolean;)       "false"
          naming            (%Boolean;)       "false"
          multi-valued      (%Boolean;)       "true"
          case-sensitive    (%Boolean;)       "false"
          read-only         (%Boolean;)       "false">
```

Attributes

attr-name

Specifies the attribute definition name used in the originating schema. This name is mapped between the application and the Identity Vault through the schema mapping rules so that the Identity Vault sees the attribute name in its name space and the external application sees the attribute name in its name space.

asn1id

Specifies the object ID (OID) for the attribute (optional).

type

Specifies the type of data contained in the attribute's value. Optional if the attribute contains string data. For other possible values for type, see <value>.

required

Specifies whether the attribute is mandatory for the class. If mandatory, the attribute requires a value in order to create an entry. The default value is "false".

naming

Specifies whether the attribute is used as part of the relative distinguished name of the entry for this base class. Default value is "false."

multi-valued

Specifies whether the attribute can have multiple values. Default value is "true".

case-sensitive

Specifies whether the value is case sensitive. Default value is "false".

read-only

Specifies whether the value is read-only. Default value is "false".

Parent

<class-def>

Remarks

The XML documents for schema definitions can become very large. Therefore, if the default values for the XML attributes match the attribute definition, do not specify them in the XML document.

<authentication-info>

Supplies the information that the driver needs to connect and log in to the external application.

Definition

```
<!ELEMENT authentication-info   (server?, user?, password?) >

<!ELEMENT server     (#PCDATA)>

<!ELEMENT user       (#PCDATA)>

<!ELEMENT password   (#PCDATA)>
```

Elements

<server>

Specifies the name of the server hosting the external application in any format that works for the external application. The name is stored in the DirXML-ShimAuthServer attribute on the DirXML-Driver object.

<user>

Specifies the user the Identity Manager driver uses to log in to the external application in any format that works for the external application. The name is stored in the DirXML-ShimAuthID on the DirXML-Driver object.

<password>

Specifies the user's password in any format that works for the external application. The password is stored securely in the DirXML-ShimAuthPassword on the DirXML-Driver object.

Parent

- <init-params>

Remarks

All of the elements in the <authentication-info> element are optional, so you can select the elements that apply to the external application.

<class-def>

Specifies a class definition for the originating schema.

Definition

```
<!ELEMENT class-def        (attr-def)* >
<!ATTLIST class-def
          class-name       CDATA            #REQUIRED
          asn1id           CDATA            #IMPLIED
          container        (%Boolean;)      "false">
```

Attributes

class-name

Specifies the class definition name used in the originating schema. This name is mapped between the application and the Identity Vault through the schema mapping rules so that the Identity Vault sees the class name in its name space and the external application sees the class name in its name space.

asn1id

Specifies the object ID (OID) for the class (optional).

Elements

<attr-def>

Specifies an attribute definition for the class specified in the <class-def> element.

Parent

<schema-def>

<component>

Specifies an individual field of a structured value.

Definition

```
<!ELEMENT component          (#PCDATA)>
<!ATTLIST component
          name               CDATA            #REQUIRED
          association-ref    CDATA            #IMPLIED>
```

Attributes

name

Specifies the name of the component and is specific to the individual attribute syntax.

association-ref

When present, marks the component as referential. When a component is marked referential, the component's value is the association of an existing entry in the Identity Vault. The Identity Vault validates these values, modifies them if the association changes, and deletes them if the association is deleted.

Parent

- <value>

<config-object>

Specifies an object or objects where additional configuration information can be obtained.

Description

During driver initialization, the Identity Manager engine processes the <query> element, replaces the <config-object> in the <init-params> with the <instance> elements returned, and passes the <init-params> to the DriverShim, SubscriptionShim, and PublicationShim init methods.

Definition

```
<!ELEMENT config-object    (query) >
<!ATTLIST config-object
          display-name      CDATA          #IMPLIED>
```

Attributes

display-name

Specifies the name to display in iManager.

Elements

<query>

Specifies where to look for the configuration objects.

Parent

- <driver-options>
- <publisher-options>
- <subscriber-options>

<driver-filter>

Specifies the object classes and attributes for which the publisher can send modifications and for which the subscriber can receive

modifications.

Description

The filter controls the information that the Identity Vault synchronizes with the external application and helps designate the authoritative source of the information. A driver which publishes changes to an attribute but does not subscribe for changes to that attribute is set up in Identity Manager to be the authoritative source for that attribute. (The other step is to modify the attribute's ACL in the Identity Vault so that the driver object is the only object with rights to modify the attribute). If no authoritative sources are being set up, the subscription and publication filters can be the same. The driver filter is stored in the DirXML-DriverFilter attribute of the DirXML-Publisher and DirXML-Subscriber objects in the Identity Vault.

The Identity Manager engine processes the filter according to the channel:

- For the subscriber channel which sends events from the Identity Vault to the driver, the engine applies the subscriber filter to the the Identity Vault events before applying the event transformation rules (see Section 8.5, Event Transformation Rules).
- For the publisher channel which sends events from the driver to the engine, the engine applies the filter after applying input transformation rules (see Section 8.7, Input Transformation Style Sheets) and the schema mapping rules (see <attr-name-map>).

The attributes and the classes specified in the filters must complement the attributes and classes specified in the rules. For example, if the create rule for the subscriber specifies that a User object must have a Surname attribute, then the subscriber filter should include the User object and its Surname attribute.

Definition

```
<!ELEMENT driver-filter   (allow-class)* >
<!ATTLIST driver-filter
          type            (publisher|subscriber)   #IMPLIED>

<!ELEMENT allow-class     (allow-attr)* >
<!ATTLIST allow-class
          class-name      CDATA              #REQUIRED>

<!ELEMENT allow-attr      EMPTY>
<!ATTLIST allow-attr
          attr-name       CDATA              #REQUIRED>
```

Attributes

type

Specifies whether publisher or subscriber should use the filter

Elements

<allow-class>

Specifies the classes that are allowed by the filter

Format

A driver-filter contains the following:

- type attribute which specifies whether the publisher or the subscriber should use the filter. If type is not specified, both use the filter.
- <allow-class> (0 or more) which specifies the classes that are allowed by the filter
    - class-name attribute which specifies the class name
    - <allow-attr> (0 or more) which specifies the attributes of the enclosing class that are allowed by the filter. This element contains an attr-name attribute.

Parent

- <init-params>

Sample Filter

The VR Test driver can be configured to use the following sample filter for the subscriber shim. This sample starts with the <input> element. The filter allows the subscriber to receive modifications for four attributes of the User object class.

```
...
<input>
   <init-params>
      <driver-filter>
         <allow-class class-name="User">
            <allow-attr attr-name="Surname"/>
            <allow-attr attr-name="CN"/>
            <allow-attr attr-name="Given Name"/>
            <allow-attr attr-name="Telephone Number"/>
         </allow-class>
      </driver-filter>
...
```

<driver-options>

Specifies configuration options for the DriverShim.

Definition

```
<!ELEMENT driver-options (ANY | config-object)* >
```

Elements

<config-object>

Specifies an object or objects where additional configuration information can be obtained.

Parent

- <init-params>

Remarks

The <driver-options> element accepts any valid XML element and attribute tags.You are free to define tags for whatever configuration options an administrator needs to make your driver function in a specific environment. The elements and attributes can contain only text.

For example, if the driver is using LDAP to communicate with the external application, the administrator might need an option to configure the port. You could use a <port> tag with numeric text to specify the port.

Sample Option Tags

The VRTest driver uses the following options.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<driver-config name="VRTestServer Driver">
   <driver-options>
      <server-id display-name="Server Instance Id">02</server-id>
   </driver-options>
   <subscriber-options>
   </subscriber-options>
   <publisher-options>
      <allow-loopback display-name="Allow Loopback">no</allow-loopback>
      <use-filter display-name="Use Filter">yes</use-filter>
      <log-input display-name="Log Input to NDS">no</log-input>
```

```
      <save-state-each-event display-name="Save state with each event">no</save-
state-each-event>
   </publisher-options>
</driver-config>
```

The tags under the driver-options and publisher-options are defined by the VR Test driver. The one exception is the display-name attribute. iManager uses this tag with its beginning and ending quotation marks to parse the file and display each of the configuration options for the driver, the publisher, and subscriber. If you do not include the display-name attribute, iManager displays the element name for the configuration parameter.

<driver-state>

Allows the DriverShim to save any required state information that it needs when it is initialized again.

Definition

```
<!ELEMENT driver-state ANY >
```

Parent

- <init-params>

Remarks

The ANY element allows you to add tags for whatever state information your driver needs to determine whether a transaction completed successfully. The Identity Manager engine keeps a queue of pending transactions and does not delete a transaction until the Identity Manager driver sends back the state of the transaction.

A state element can be included in any output or input command which the driver sends to the Identity Manager engine.The Identity Manager engine stores the information in the DirXML-DriverStorage attribute of the DirXML-Driver object and returns the information in the init-params command when the driver shim, subscriber shim, and publisher shim are started.

Sample State Tags

The VR Test driver saves the state information for the DriverShim. This information consists of a timestamp and a count. This sample XML starts with the <input> element.

```
...
<input>
   <init-params>
      ...
      <driver-state>
         <time-stamp>2000-02-18 10:06:52.610</time-stamp>
         <run-count>51</run-count>
      </driver-state>
```

```
    </init-params>
</input>
...
```

If you have binary data to save as state information, you will need to encode it as base64.

<modify-attr>

Specifies the attribute values to modify for the modify command or event.

Description

Each <modify-attr> element must contain at least one <add-value>, <remove-all-values> or <remove-value> element.

Definition

```
<!ELEMENT modify-attr       (remove-value |
                             remove-all-values |
                             add-value)+>
<!ATTLIST modify-attr
           attr-name         CDATA            #REQUIRED>

<!ELEMENT remove-all-values  EMPTY>

<!ELEMENT remove-value       (value+)>

<!ELEMENT add-value          (value+)>
```

Attributes

attr-name

Specifies the name of the attribute to modify.

Elements

<remove-value>

Specifies the value to remove. This element may be included multiple times. If the value doesn't exist, the driver should ignore the discrepancy and return success.

<remove-all-values>

Specifies to remove all values of the attribute.

<add-value>

Specifies the value to add to the attribute. This element may be included multiple times. If the value already exists, the driver should ignore the discrepancy and return success.

Parent

- <modify>

<old-password>

Specifies the existing password in a <modify-password> event. Added in Identity Manager 1.1.

Definition

```
<!ELEMENT old-password  (#PCDATA)>
```

Elements

<old-password>

Specifies the current password.

Parent

- <modify-password>

<parent>

Specifies the entry's parent container.

Description

The <parent> element specifies

- The container of an entry in an instance command or event
- The destination container for a move event or move command

Definition

```
<!ELEMENT parent (association?)>
<!ATTLIST parent
          src-dn          CDATA           #IMPLIED
          src-entry-id    CDATA           #IMPLIED
          dest-dn         CDATA           #IMPLIED
          dest-entry-id   CDATA           #IMPLIED>
```

Attributes

src-dn

Specifies the distinguished name of the parent entry.

src-entry-id

Reserved, used internally by the Identity Manager engine.

dest-dn

Reserved, used internally by the Identity Manager engine.

dest-entry-id

Reserved, used internally by the Identity Manager engine.

Elements

<association>

Specifies the parent's unique key in the external application.

Parent

- <instance>
- <move>

Remarks

When the Identity Manager engines sends the <parent> element to the Identity Manager driver in a <move> or <instance> element, the <association> element is included if an association has been established for the parent.

When the Identity Manager driver sends the <parent> element to the Identity Manager engine in a <move> or <instance> element, the <parent> element must contain an <association> element.

<publisher-options>

Specifies configuration options for the PublisherShim.

Definition

```
<!ELEMENT publisher-options (ANY | config-object)* >
```

Elements

<config-object>

Specifies an object or objects where additional configuration information can be obtained.

Parent

- <init-params>

Remarks

The <publisher-options> element accepts any valid XML element and attribute tags. You are free to define tags for whatever configuration options an administrator needs to make your driver function in a specific environment. The elements and attributes can contain only text.

For example, if the driver is using LDAP to communicate with the external application, the administrator might need an option to configure the port. You could use a <port> tag with numeric text to specify the port.

Sample Option Tags

The VRTest driver uses the following options.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<driver-config name="VRTestServer Driver">
   <driver-options>
      <server-id display-name="Server Instance Id">02</server-id>
   </driver-options>
   <subscriber-options>
   </subscriber-options>
   <publisher-options>
      <allow-loopback display-name="Allow Loopback">no</allow-loopback>
      <use-filter display-name="Use Filter">yes</use-filter>
      <log-input display-name="Log Input to NDS">no</log-input>
```

```
      <save-state-each-event display-name="Save state with each event">no</save-state-each-event>

   </publisher-options>

</driver-config>
```

The tags under the driver-options and publisher-options are defined by the VR Test driver. The one exception is the display-name attribute. IManager uses this tag with its beginning and ending quotation marks to parse the file and display each of the configuration options for the driver, the publisher, and subscriber. If you do not include the display-name attribute, the IManager snapin displays the element name for the configuration parameter.

<publisher-state>

Allows the PublisherShim to save any required state information that it needs when it is initialized again.

Definition

```
<!ELEMENT publisher-state ANY >
```

Parent

- <init-params>

Remarks

The ANY element allows you to add tags for whatever state information your driver needs to determine whether a transaction completed successfully. The Identity Manager engine keeps a queue of pending transactions and does not delete a transaction until the Identity Manager driver sends back the state of the transaction.

A state element can be included in any output or input command which the driver sends to the DirXML engine. The Identity Manager engine stores the information in the DirXML-DriverStorage attribute of the DirXML-Driver object and returns the information in the init-params command when the driver shim, subscriber shim, and publisher shim are started.

Sample State Tags

The VR Test driver saves the state information for the Driver Shim. This information consists of a timestamp and a count. This sample XML starts with the <input> element.

```
...

<input>

   <init-params>

      ...

      <publisher-state>

         <time-stamp>2000-02-18 10:06:52.610</time-stamp>

         <run-count>51</run-count>

      </publisher-state>
```

```
   </init-params>
</input>
...
```

If you have binary data to save as state information, you will need to encode it as base64.

<read-attr>

Specifies which attribute values are returned with entries that match the search filter. If no attributes are specified, all attributes are returned. If a <read-attr> element is specified without an attr-name attribute, no attributes are returned.

Definition

```
<!ELEMENT read-attr       EMPTY>
<!ATTLIST read-attr
          attr-name       CDATA                #IMPLIED
          type      (%Read-attr-type;)        "default">
```

Attributes

attr-name

Specifies the name of the attribute that should be returned with its value for all matching entries. The attribute's name is specified in the name space of the sender.

type

Specifies how to parse the attribute's value. It supports two values: "default" and "xml". If type="xml", then the attribute value will be parsed as XML and returned as such.

Parent

<query>

<remove-value>

Specifies the values to remove.

Definition

```
<!ELEMENT remove-value       (value+)>
```

Elements

<value>

Specifies the attribute value to remove.

Parent

<modify-attr>

<search-attr>

Specifies the search filter for attribute values. If more than one <search-attr> element is specified, the entry must match all attributes to be returned.

Definition

```
<!ELEMENT search-attr     (value)+ >
<!ATTLIST search-attr
          attr-name       CDATA              #REQUIRED>
```

Attributes

attr-name

Specifies the attribute that entry must have to be returned.

Elements

<value>

Specifies the attribute value the entry must have to be returned. If more than one value is specified for the attribute, the entry must match all values to be returned.

Parent

<query>

<search-class>

Specifies the search filter for object classes. If the query contains no <search-class> elements, all entries matching the scope and the <search-attr> elements are returned.

Definition

```
<!ELEMENT search-class    EMPTY>
<!ATTLIST search-class
          class-name      CDATA              #REQUIRED>
```

Attributes

class-name

Specifies the base class of the entries to search for. An entry must match one of the specified base classes to be returned.

Parent

<query>

<source>

Specifies the source that created the XML document.

Definition

```
<!ELEMENT source (product?, contact?)>

<!ELEMENT product (#PCDATA)>
<!ATTLIST product
          version        CDATA          #IMPLIED
          asn1id         CDATA          #IMPLIED>

<!ELEMENT contact (#PCDATA)>
```

Elements

product

Specifies the name of the product that produced the document.

contact

Specifies a point of contact for the product such as a telephone number or a company.

Attributes of <product>

version

Specifies the version of the product.

asn1id

Specifies the ASN1 ID or OID assigned to the product.

Parent

<nds>

<subscriber-options>

Specifies configuration options for the SubscriptionShim.

Definition

```
<!ELEMENT subscriber-options (ANY | config-object)* >
```

Elements

<config-object>

Specifies an object or objects where additional configuration information can be obtained.

Parent

- <init-params>

Remarks

The <subscriber-options> element accepts any valid XML element and attribute tags.You are free to define tags for whatever configuration options an administrator needs to make your driver function in a specific environment. The elements and attributes can contain only text.

For example, if the driver is using LDAP to communicate with the external application, the administrator might need an option to configure the port. You could use a <port> tag with numeric text to specify the port.

Sample Option Tags

The VRTest driver uses the following options.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<driver-config name="VRTestServer Driver">
   <driver-options>
      <server-id display-name="Server Instance Id">02</server-id>
   </driver-options>
   <subscriber-options>
   </subscriber-options>
   <publisher-options>
      <allow-loopback display-name="Allow Loopback">no</allow-loopback>
      <use-filter display-name="Use Filter">yes</use-filter>
      <log-input display-name="Log Input to NDS">no</log-input>
      <save-state-each-event display-name="Save state with each event">no</save-
state-each-event>
   </publisher-options>
</driver-config>
```

The tags under the driver-options and publisher-options are defined by the VR Test driver. The one exception is the display-name attribute. iManager uses this tag with its beginning and ending quotation marks to parse the file and display each of the configuration options for the driver, the publisher, and subscriber. If you do not include the display-name attribute, the iManager displays the element name for the configuration parameter.

<subscriber-state>

Allows the SubscriberShim to save any required state information that it needs when it is initialized again.

Definition

```
<!ELEMENT subscriber-state ANY >
```

Parent

- <init-params>

Remarks

The ANY element allows you to add tags for whatever state information your driver needs to determine whether a transaction completed successfully. The Identity Manager engine keeps a queue of pending transactions and does not delete a transaction until the Identity Manager driver sends back the state of the transaction.

A state element can be included in any output or input command which the driver sends to the Identity Manager engine. The Identity Manager engine stores the information in the DirXML-DriverStorage attribute of the DirXML-Driver object and returns the information in the init-params command when the driver shim, subscriber shim, and publisher shim are started.

Sample State Tags

The VR Test driver saves the state information for the DriverShim. This information consists of a timestamp and a count. This sample XML starts with the <input> element.

```
...
<input>
   <init-params>
      ...
      <subscriber-state>
         <time-stamp>2000-02-18 10:06:52.610</time-stamp>
         <run-count>51</run-count>
      </subscriber-state>
   </init-params>
</input>
...
```

If you have binary data to save as state information, you will need to encode it as base64.

<value>

Specifies an attribute's value.

Definition

```
<!ELEMENT value          (#PCDATA | component)*>
<!ATTLIST value
          type              (%Attr-type;)    #IMPLIED
```

```
association-ref   CDATA           #IMPLIED
naming            (%Boolean;)     "false">
```

Attributes

type

Specifies the syntax of the directory attribute.

• If the type is structured, the <value> element contains at least one <component> element.

• If the type is octet, the <value> element contains base64 encoded binary data.

• If the type is neither structured or octet, the <value> element contains text.

association-ref

When present, marks the value as referential and contains the association value (see <association>). When a syntax is marked referential, the attribute's value must be the distinguished name of an existing entry in the Identity Vault. The Identity Vault validates these values, modifies them if the entry's name changes, and deletes them if the entry is deleted.

naming

Reserved; used internally by the Identity Manager engine to mark the attribute as a directory attribute that names the entry. Such marked attributes require a rename command rather than a modify command to modify the value.

Elements

<component>

Specifies an individual field of a structured attribute.

Parent

The following elements are parents of the <value> element:

- <add-attr>
- <add-value>
- <attr>
- <match-attr>
- <remove-value>
- <required-attr>
- <search-attr>

Remarks

eDirectory supports 28 syntax definitions that can be used for an attribute's type. Identity Manager supports only 10. The following table shows how the 28 definitions have been mapped to the 10 definitions and the component names that have been supplied for the various structured syntax definitions.

Table 7-2 Mapping eDirectory syntax to Identity Manager syntax

eDirectory Syntax Name	Identity Manager Type	Components and Notes
SYN_UNKOWN	octet	base64 encoded data
SYN_DIST_NAME	dn	referential
SYN_CE_STRING	string	
SYN_CI_STRING	string	
SYN_PR_STRING	string	
SYN_NU_STRING	string	
SYN_CI_LIST	structured	one or more string components
SYN_BOOLEAN	state	"true" or "false"
SYN_INTEGER	int	
SYN_OCTET_STRING	octet	base64 encoded data
SYN_TEL_NUMBER	teleNumber	
SYN_FAX_NUMBER	structured	three components: • faxNumber • faxBitCount • faxParameters base64 encoded data
SYN_NET_ADDRESS	structured	two components • netAddrType • netAddr base64 encoded data
SYN_OCTET_LIST	structured	one or more octet components. base64 encoded data
SYN_EMAIL_ADDRESS	structured	two components: • eMailType • eMailAddr

SYN_PATH	structured	three components: • nameSpace • volume (referential) • path
SYN_REPLICA_POINTER	structured	four components • server (referential) • replicaType • replicaNumber • count
SYN_OBJECT_ACL	structured	three components • protectedName • trustee (referential) • privileges
SYN_PO_ADDRESS	structured	6 string components
SYN_TIMESTAMP	structured	three components: • seconds • replicaNumber • eventID
SYN_CLASS_NAME	className	
SYN_STREAM	octet	64base encoded data
SYN_COUNTER	counter	
SYN_BACK_LINK	structured	two components: • serverDn (referential) • remoteId
SYN_TIME	time	
SYN_TYPED_NAME	structured	three components: • dn (referential) • level

		• interval
SYN_HOLD	structured	two components: • holdEntryDn (referential) • holdAmount
SYN_INTERVAL	interval	

For more information about the eDirectory syntax definitions, see the NDS Schema Reference.

8.0 Rule Reference

Rules define what data is transferred and how the data is synchronized between the two database. The nds.dtd defines the XML syntax for the rules, and the administrator uses Designer or iManager to create a rule file. The file is then stored in the Identity Vault as XML data in a stream attribute.

In addition to the XML-formatted rules, a driver can use XSLT style sheets for the following:

- Event transformations rules
- Input transformation rules
- Output transformation rules
- Matching, create, and placement rules

Whether an XML-formatted rule or an XSLT style sheet, rules transform an input XML document by changing the data or the XML structure to form a result, or output, document. For example, a create rule blocks the creation of an entry by transforming the input document containing the <add> element into an output document that has no <add> element.

Style sheets can be used to generate attribute values for the XML rules. For example, the Identity Vault does not require that a CN be unique in the Identity Vault, but only unique for a particular container. A style sheet in a create rule can ensure that a CN is unique for the Identity Vault.

This chapter describes the transformations caused by the rules, the XML format of the rules (matching, create, and placement), and style sheet restrictions for event transformations, input transformations, and output transformations. Section 9.0, Style Sheets describes implementation details for style sheets.

For an overview of the rules, see Section 4.0, Introduction to the Rules and Filters.

The top rule elements in the nds.dtd are described in the following sections:

- <attr-name-map>
- <matching-rules>
- <create-rules>
- <placement-rules>

8.1 Schema Mapping Elements

The <attr-name-map> element is the top level element for the schema mapping rules. Mapping rules determine how the Identity Vault schema interacts with the external database schema. They map the Identity Vault class definitions and attributes with corresponding

attribute and class definitions in the external application.

Mapping rules can be set up for attribute names or class names:

- An attribute mapping is specified using the <attr-name> element. The rule must specify the Identity Vault name of the attribute and the corresponding name in the application. Optionally, it can include a classname attribute which specifies the Identity Vault class for which the rule applies. If a classname is not specified, the attribute mapping applies to all classes with that attribute.

- A class mapping is specified using the <class-name> element. The rule must specify the Identity Vault name of the class and the corresponding name in the application.

The Identity Manager engine uses the schema mapping rules to map the class-name and attr-name attributes of all elements in the document. A class can be mapped to only one class in the other application.

The rules are applied whenever XML is sent to and returned from the following methods:

- SubscriptionShim.execute

- XmlQueryProcessor.query

- XmlCommandProcessor.execute

The rules are also applied whenever XML is sent to (but not the XML returned from) the following methods:

- SubscriptionShim.init

- PublicationShim.init

The rules are applied before output transformation rules when the Identity Manager engine issues a document to the driver and before input transformation rules when the driver issues a document to the Identity Manager engine.

The schema mapping rules should complement the other rules. For example, if the matching rules depend upon the User class and the Surname, Given Name, and Telephone Attributes, the schema mapping rules should contain this class and these attributes.

<attr-name-map>

Determines how the Identity Vault schema interacts with the application's schema.

Description

The <attr-name-map> element is the top level element for the schema mapping rules. All other elements in the document are subordinate to this element.

Mapping rules map specified the Identity Vault class definitions and attributes with corresponding attribute and class definitions in the external application. Mapping rules are kept in an DirXML-Rule object and associated with your driver through the DirXML-MappingRule attribute of your driver's DirXML-Driver object.

Definition

```
<!ELEMENT attr-name-map (attr-name | class-name)*>
```

```
<!ELEMENT attr-name (nds-name, app-name)>
```

```
<!ATTLIST attr-name
          class-name    CDATA    #IMPLIED>

<!ELEMENT class-name (nds-name, app-name)>

<!ELEMENT nds-name (#PCDATA)>

<!ELEMENT app-name (#PCDATA)>
```

Attributes

class-name

Specifies the class definition which the attribute belongs to in the <attr-name> element.

Elements

<attr-name>

Specifies an attribute mapping between the Identity Vault name space and the application name space.

<class-name>

Specifies a class mapping between the Identity Vault name space and the application name space.

<nds-name>

Specifies either the attribute or class name in the Identity Vault name space. The attribute names specified must be unique to the class. The class names specified must be unique in the rules.

<app-name>

Specifies either the attribute or class name in the application name space. The attribute names specified must be unique to the class. The class names specified must be unique in the rules.

Parent

- Top element

Examples

The following example contains three mapping rules.

```
<attr-name-map>
<!-- map NDS class User application class inetOrgPerson -->
```

```
   <class-name>
      <nds-name>User</nds-name>
      <app-name>inetOrgPerson</app-name>
   </class-name>

<!-- map NDS attribute Given Name to application attribute-->
<!-- givenName for class User                            -->

   <attr-name class-name="User">
      <nds-name>Given Name</nds-name>
      <app-name>givenName</app-name>
   </attr-name>

<!-- map NDS attribute Surname to application attribute  -->
<!-- sn for all classes that don’t have a                -->
<!-- class-specific mapping                              -->

   <attr-name>
      <nds-name>Surname</nds-name>
      <app-name>sn</app-name>
   </attr-name>
</attr-name-map>
```

For another example, see the mapping_rule.xml file used by the VRTest driver.

8.2 Matching Rule Elements

Matching rules establish links between an entry in the Identity Vault and an entry in the external application. If a match is successful, an association between the two entries is created.

The <matching-rules> element can contain more than one matching rule. When multiple rules are specified, the Identity Manager engine applies the rules in the order listed. The rules produce one of the following results:

- One match. If exactly one match is found, the entry in the source is associated with the entry in the destination. The Identity Manager engine reconciles attribute values for the attributes included in the driver filter.
- Multiple matches. If more than one match is found, the Identity Manager engine signals an error condition. The system administrator must either manually associate the entry or modify the matching rules to be more specific.
- No match. If no match is found, the Identity Manager engine continues to process the add operation by applying the create rules.

The names of the attributes and classes in the matching rules are in the Identity Vault name space.

The Identity Manager engine queries the destination for entries that match the subject of an <add> element.

- For the subscriber channel, the source is Identity Vault and the destination is the external application.

- For the publisher channel, the source is the external application and the destination is Identity Vault.

The input document is transformed so that the Identity Manager engine can inspect the results of the query. Each channel has its own transformation process.

Subscriber Channel Transformation. If the results of a query indicate one or more successful matches with application entries, the transformation adds an <association> element as a child of the <add> element for each matched application entry. Each <association> element contains the unique key for the application entry, a key that the Identity Manager driver supplies.

If the results of the query indicate no successful match was found, the absence of an <association> element in the output document signals the Identity Manager engine that no match was found.

If the engine discovers multiple <association> elements as children of an <add> element after it has applied the matching rules, it signals an error and does not allow the add operation to proceed.

Publisher Channel Transformation. If the results of the query indicate a single successful match, the transformation adds a dest-dn attribute to the <add> element. The dest-dn attribute's value is the distinguished name of the matching Identity Vault object. If one of the following error conditions occur, the transformation places a Unicode character in the dest-dn attribute:

- If multiple matches are found, 0xfffd is placed in the attribute.

- If a match is found with an Identity Vault object that is already associated with an object in the application, 0xfffc is placed in the attribute.

If the engine discovers either special Unicode character in the dest-dn attribute after the matching rules have been applied, the engine signals an error and the add operation is aborted.

The Unicode characters 0xfffd and 0xfffc are represented in an XML style sheet using XML character references (� and ￼, respectively).

XML Matching Rule Syntax. Matching rules use the following elements to determine whether entries match:

- *<match-class>*. Specifies that the entry must match the specified base class. For example, the entry must belong to the User class.

- *<match-attr>*. Specifies that the entry must have values for the specified attributes, and if values are specified, the attributes must have the specified values. For example, the entry must have Surname, Given Name, and Telephone Number attributes that match the target entry's values.

- *<match_path>*. Specifies that a portion of the entry's DN must match the specified path in the prefix attribute. For example, the entry must come from the provo.novell container when the source directory contains the following containers: sales.provo.novell, dev.provo.novell, hr.orem.novell.

-

<matching-rules>

Specify which class and attribute values must match for an entry in the Identity Vault to match an entry in the external application.

Description

Matching rules establish links between an entry in the Identity Vault and an entry in the external application. If a match is successful, an association between the two entries is created.

The Identity Manager engine uses the matching rules when it receives an add operation from either the publisher or the subscriber. The Identity Manager engine changes a modify operation to an add operation when an entry does not have an association. Matching rules are applied before the create rules.

Definition

```
<!ELEMENT matching-rules (matching-rule*)>

<!ELEMENT matching-rule (match-class*,
                         match-path?,
                         match-attr*)>
<!ATTLIST matching-rule
          description   CDATA   #IMPLIED>

<!ELEMENT match-class   EMPTY>
<!ATTLIST match-class
          class-name    CDATA   #REQUIRED>

<!ELEMENT match-path    EMPTY>
<!ATTLIST match-path
          prefix         CDATA   #REQUIRED>

<!ELEMENT match-attr (value)+ >
<!ATTLIST match-attr
          attr-name      CDATA    #REQUIRED>
```

Elements

<matching-rule>

Specifies the criteria for finding a matching entry in the destination.

<match-attr>

Specifies an attribute value that the entry must match.

<match-class>

Specifies the base class the entry must match before the rule can be used.

<match-path>

Specifies the root container of the directory that the entry must match.

Parent

- Top element

Examples

The following sample has three rules. The first rule requires User entries to match on both the Surname and Given Name attributes. The second rule, which is applied if the first rule fails, requires that the entries match on the Surname attribute. If both these rules fail, entries are match on their CN attribute.

The following example illustrates three matching rules.

```
<matching-rules>

<!-- for Users, first try to match on Surname, Given Name -->
<!-- and Location                                         -->

   <matching-rule>
      <match-class class-name="User"/>
      <match-attr attr-name="Surname"/>
      <match-attr attr-name="Given Name"/>
      <match-attr attr-name="Location"/>
   </matching-rule>

<!-- for Users, then try to match on Surname only in     -->
<!-- the o=novell subtree                                 -->

   <matching-rule>
      <match-class class-name="User"/>
      <match-path prefix="o=novell"/>
      <match-attr attr-name="Surname"/>
   </matching-rule>

<!-- for all classes try to match on CN only              -->

   <matching-rule>
      <match-attr attr-name="CN"/>
   </matching-rule>
</matching-rules>
```

For another example, see the sub_matching_rule.xml file used by the subscriber for the VRTest driver.

<matching-rule>

Specifies the criteria for finding a matching entry in the destination.

Description

The Identity Manager engine uses the following algorithm to determine if the add operation can use a particular matching rule.

1. If the rule contains <match-class> elements, the class-name attribute in the add operation must match one of the class-names specified in the <match-class> elements.

2. If the rule contains <match-attr> elements, the add operation must contain an attribute value for each of the attributes specified in the <match-attr> elements.

If a rule cannot be used, it is skipped, and the Identity Manager engine moves to the next rule.

Once a suitable rule is found, the Identity Manager engine queries the destination for entries that have the attributes specified by the <match-attr> elements in the matching rule and by the <add-attr> elements in the add operation. If the rule contains a <match-path> element, the Identity Manager engine uses that value to set the dest-dn attribute in the <query> element (this attribute specifies the starting point for the search).

The destination returns <instance> elements for each entry that matches.

Definition

```
<!ELEMENT matching-rule (match-class*,
                         match-path?,
                         match-attr*)>
<!ATTLIST matching-rule
          description   CDATA   #IMPLIED>
```

Attributes

description

Provides a description of the rule for display in the IManager snapin.

Elements

<match-attr>

Specifies an attribute value that the entry must match.

<match-class>

Specifies the base class the entry must match before the rule can be used.

<match-path>

Specifies the root container of the directory that the entry must match.

Parent

- <matching-rules>

<match-attr>

Specifies an attribute for create, placement, and matching rules.

Description

When the <match-attr> element is specified in a <create-rule> or <placement-rule> element, at least one value must be specified.

When the <match-attr> element is specified in a <matching-rule> element, it must not contain a value.

Definition

```
<!ELEMENT match-attr       (value)+ >
<!ATTLIST match-attr
          attr-name        CDATA            #REQUIRED>
```

Attributes

attr-name

Specifies the name of the attribute in the Identity Vault name space.

Elements

<value>

Specifies the value for the attribute.

Parent

- <create-rule>
- <matching-rule>
- <placement-rule>

<match-class>

Specifies the base class that an entry must match

Definition

```
<!ELEMENT match-class      EMPTY>

<!ATTLIST match-class

          class-name       CDATA             #REQUIRED>
```

Attributes

class-name

Specifies the base class in the Identity Vault name space.

Parent

- <matching-rule>
- <placement-rule>

<match-path>

Specifies a subtree in the directory that is used in matching the entry.

Description

When the <match-path> element is specified in a <matching-rule> element, the prefix attribute is used to specify the starting point of the <query> element that is generated from the matching rule. The name space of the prefix is the destination of the add operation.

The format of the prefix is dependent upon the name space.

- For the Identity Vault, the format is slash, for example, \treename\container\container. If the leading slash is omitted, the path is assumed to be relative to the tree root.

- For the external application, the format is application dependent and must be documented by the driver writer for the system

administrator.

Definition

```
<!ELEMENT match-path        EMPTY>
<!ATTLIST match-path
          prefix            CDATA             #REQUIRED>
```

Attributes

prefix

Specifies the distinguished name of a subtree in the directory.

Parent

- <matching-rule>
- <placement-rule>

8.3 Create Rule Elements

Create rules perform transformations only on <add> elements. They are only applied if a match is not found by the matching rules. The transformation can remove (veto) elements or add data.

Vetoing elements. The transformation compares the <add> elements against criteria and removes the <add> elements that don't meet the criteria.

Adding data. The transformation can synthesize attribute values from another attribute. For example, the Identity Vault Surname and Given Name attributes can be synthesized from an application's Full Name attribute.

Also, a create rule can set an initial password for an entry. To set the initial password, the create rule style sheet adds a <password> element as a child of the <add> element. The PCDATA content of the <password> element is the initial password value for the entry. The <password> element can only be added through a style sheet.

The Identity Manager engine supports the <password> element for entries created in the Identity Vault from the publisher channel. Identity Manager drivers are encouraged to support the <password> element on the subscriber channel. For an example, see Section 9.7, Creating a Password Example: Create Rule.

XML Create Rule Syntax. Create rules use the following elements to veto <add> elements or to add data.

- *<required-attr>*. Specifies that the add command must have values for all of the listed attributes, or the add fails. The rule can supply a default value for a required attribute. If a default value is supplied, the transformation uses the following process:
    - If the add command does not have a value for such an attribute, the entry is given the default value.
    - If the add command has a value for such an attribute, the default value is ignored and the entry is given the value specified in the add command.

- *<match-attr>*. Specifies that the add command must contain an attribute value for each of the attributes specified, or the add fails.
- *<template>*. Specifies the distinguished name of a template object in the destination directory which supplies default values for a group of attributes. For example, the template can ensure that an entry which is being created in the Identity Vault from an application has the same password restrictions as other Identity Vault entries.
-

You can have multiple create-rule elements in the file. Each rule is processed in the order that it appears. If no applicable rule is found, the add is allowed.

<create-rules>

Determine whether a new entry can be created in the destination as a result of an add operation in the source.

Description

The create rules specify the conditions for creating a new entry in either the Identity Vault or external application. The create rules are kept in an DirXML-Rule object and associated with your driver through the DirXML-CreateRule attribute of your driver's DirXML-Publisher and DirXML-Subscriber objects. Both the subscriber and publisher can have their own rules, or they can share the same rules.

For the subscriber rules, the source is the Identity Vault and the destination is the external application. For the publisher rules, the source is the external application and the destination is the Identity Vault.

The Identity Manager engine applies create rules only after applying any existing matching rules, and these matching rules fail to find a matching entry in the destination.

Create rules should complement the matching rules. Usually a create rule should require all the attributes used by a matching rule. This defers the creation of an entry until enough is known about an entry to perform a reasonable match in the receiving application.

Definition

```
<!ELEMENT create-rules (create-rule)*>


<!ELEMENT create-rule (match-attr*,
                        required-attr*,
                        template?) >
<!ATTLIST create-rule
          class-name    CDATA   #IMPLIED
          description   CDATA   #IMPLIED>


<!ELEMENT match-attr    (value)+ >
<!ATTLIST match-attr
          attr-name      CDATA    #REQUIRED>


<!ELEMENT required-attr (value)*>
```

```
<!ATTLIST required-attr
          attr-name     CDATA   #REQUIRED>

<!ELEMENT template EMPTY>
<!ATTLIST template
          template-dn   CDATA   #REQUIRED>
```

Elements

<create-rule>

Specifies the criteria for creating a new entry in the destination.

<match-attr>

Specifies attribute values the entry must have to be created.

<required-attr>

Specifies the attributes that the entry must have to be created.

<template>

Specifies the distinguished name of the template to use when creating the entry.

Parent

- Top element

Sample Create Rules

The following example illustrates two create rules.

```
<create-rules>
<!-- For all Users in the Defense organization require -->
<!-- Given Name, Surname, and Security Clearance.      -->
<!-- Create using the templates\Secure User template.  -->

   <create-rule class-name="User">
      <match-attr attr-name="OU">
         <value>Defense</value>
      </match-attr>
      <required-attr attr-name="Given Name"/>
```

```
      <required-attr attr-name="Surname"/>
      <required-attr attr-name="Security Clearance"/>



      <template template-dn="templates\Secure User"/>
   </create-rule>

<!-- For all other Users require Given Name and Surname. -->
<!-- Default the value of Security Clearance to None     -->
<!-- Don't use a template for creation                   -->

   <create-rule class-name="User">
      <match-attr attr-name="OU">
         <value>Defense</value>
      </match-attr>
      <required-attr attr-name="Given Name"/>
      <required-attr attr-name="Surname"/>
      <required-attr attr-name="Security Clearance">
         <value>None</value>
      </required-attr>
   </create-rule>
</create-rules>
```

For another example, see the sub_create_rule.xml file used by the subscriber in the VRTest driver.

<create-rule>

Specifies the criteria for creating a new entry in the destination of an add operation.

Description

The Identity Manager engine uses the following algorithm to determine if the add operation can use a particular create rule:

1. If a class-name attribute is specified in the <create-rule> element, the class-name attribute in the add operation must match the class-name attribute in the <create-rule> element.
2. If <match-attr> elements are specified in the <create-rule> element, the add operation must contain an attribute value for each of the attributes specified in the <match-attr> elements.

If a rule cannot be used, it is skipped, and the Identity Manager engine moves to the next rule.

Once a suitable rule is found, the Identity Manager engine evaluates the add operation to see if it has values for all the <required-attr> elements that do not have default values.

- If the add operation does not have values for all the <required-attr> elements and the create rule does not supply default values for these attributes, the add operation fails.
- If the add operation has all the required values but is missing some of the default values, the Identity Manager engine fills in the missing default values.

If a <template> element is specified, the template-dn attribute of the <add> element is filled in.

If not applicable rule is found, the add is allowed.

Definition

```
<!ELEMENT create-rule (match-attr*,
                       required-attr*,
                       template?) >
<!ATTLIST create-rule
          class-name    CDATA   #IMPLIED
          description   CDATA   #IMPLIED>
```

Attributes

class-name

Specifies the base class that this rule applies to. If no base class is specified, the rule applies to all classes. The name is specified in the Identity Vault name space.

description

Provides a description of the rule for display in iManager.

Elements

<match-attr>

Specifies an attribute value that the entry must match in order for the rule to be selected.

<required-attr>

Specifies the attributes that the entry must have. Default values for the attributes may be specified.

<template>

Specifies the distinguished name of template to use in creating the entry. The name is specified in the destination name space.

Parent

- <create-rules>

<match-attr>

Specifies an attribute for create, placement, and matching rules.

Description

When the <match-attr> element is specified in a <create-rule>, at least one value must be specified.

Definition

```
<!ELEMENT match-attr       (value)+ >
<!ATTLIST match-attr
          attr-name        CDATA            #REQUIRED>
```

Attributes

attr-name

Specifies the name of the attribute in the Identity Vault name space.

Elements

<value>

Specifies the value for the attribute.

Parent

- <create-rule>
- <matching-rule>
- <placement-rule>

<required-attr>

Specifies an attribute that is required to create an entry.

Description

The <required-attr> element may contain one or more <value> elements.

- When the <required-attr> element contains one or more <value> elements and the <add> element does not specify that attribute, the values are used as default values.
- When the <required-attr> element contains no values, the <add> element must contain one or more value for the specified attribute.

Definition

```
<!ELEMENT required-attr (value)*>
<!ATTLIST required-attr
          attr-name     CDATA   #REQUIRED>
```

Attributes

attr-name

Specifies the name of the required attribute. The name is specified in the Identity Vault name space.

Elements

<value>

Specifies a default value for the attribute.

Parent

- <create-rule>

<template>

Specifies the template to use when creating the entry.

Description

A template is used to specify default attribute values for all entries created in a container in the directory. For example, an Identity Vault template usually specifies such attributes as minimum password length and password expiration interval.

Definition

```
<!ELEMENT template EMPTY>
<!ATTLIST template
          template-dn   CDATA   #REQUIRED>
```

Attributes

template-dn

Specifies the distinguished name of the template in the destination name space.

Parent

- <create-rule>

8.4 Placement Rule Elements

Placement rules transform only <add> elements, and are only applied if the <add> element was not vetoed by the create rules. Placement rule transformations fill in the dest-dn attribute of the <add> element, placing the entry in the directory hierarchy and supplying the entry's distinguished name.

Publisher Channel Transformation. Placement rules are required to create an entry in the Identity Vault. They generate the distinguished name for the dest-dn attribute. If they fail to generate a value for the dest-dn attribute, the entry is not created in the Identity Vault.

The publisher channel supports two formats for the dest-dn attribute:

- Slash—for example, \tree_name\container\entry
- Qualified slash—for example, \T=tree name \O=container \OU=container \CN=entry

If the leading backslash is omitted, the distinguished name is considered to be relative to the root of the Identity Vault tree that is hosting Identity Manager.

Subscriber Channel Transformation. Placement rules may or may not be required to create entries in the external application. If the external application has a hierarchical format, placement rules should probably be required. The driver and the application determine the format of the dest-dn attribute.

XML Placement Rule Syntax. Placement rules support the following elements to determine whether the rule should be used to place an entry:

- *<match-class>* If the rule contains any match class elements, the base class of the entry must match the class-name attribute in the rule. If the match fails, the placement rule is not used for that record.
- *<match-attr>* If the rule contains any match attribute elements, the record must contain an attribute value for each of the attributes specified in the match attribute element. If the match fails, the placement rule is not used for that record.
- *<match-path>* If the rule contains any match path elements, a portion of the record's DN must match the prefix attribute specified in the match path element. If the match fails, the placement rule is not used for that record.

•

The <placement> element in the rule specifies where to place the entry.

You can have multiple placement-rule elements in the file. Each rule is processed in the order that it appears. If an entry does not match any of the rules, the dest-dn attribute of the add operation is left blank.

<placement-rules>

Generate distinguished names for new entries.

Description

The placement rules specify where a new entry is created in the destination. Both the subscriber and publisher can have their own rules, or they can share the same rules. The placement rules are kept in an DirXML-Rule object and associated with your driver through the DirXML-PlacementRule attribute of your driver's DirXML-Publisher and DirXML-Subscriber objects.

Definition

```
<!ELEMENT placement-rules (placement-rule*)>
<!ATTLIST placement-rules
          src-dn-format    (%dn-format;)   "slash"
          dest-dn-format   (%dn-format;)   "slash"
          src-dn-delims    CDATA           #IMPLIED
          dest-dn-delims   CDATA           #IMPLIED>

<!ELEMENT placement-rule (match-class*,
                         match-path*,
                         match-attr*,
                         placement)>
<!ATTLIST placement-rule
          description      CDATA            #IMPLIED>

<!ELEMENT match-class      EMPTY>
<!ATTLIST match-class
          class-name       CDATA            #REQUIRED>

<!ELEMENT match-path       EMPTY>
<!ATTLIST match-path
          prefix           CDATA            #REQUIRED>
```

```
<!ELEMENT match-attr       (value)+ >
<!ATTLIST match-attr
          attr-name        CDATA            #REQUIRED>

<!ELEMENT placement        (#PCDATA |
                           copy-name |
                           copy-attr |
                           copy-path |
                            copy-path-suffix)* >
```

Attributes

dest-dn-delims

Specifies the format of the destination dn when the dest-dn-format is empty. See the table below for the defined character set.

dest-dn-format

Specifies the format of the destination dn (dot, qualified dot, slash, qualified-slash, or ldap) when the des-dn-delims attribute is empty.

src-dn-delims

Specifies the format of the source dn when the src-dn-format is empty. See the table below for the defined character set.

src-dn-format

Specifies the format of the source dn (dot, qualified dot, slash, qualified-slash, or ldap) when the src-dn-delims attribute is empty.

Only one of the attributes for formats can specify a value, either dest-dn-delims or dest-dn-format, but not both.

The format for the Identity Vault is always slash.

The format for the external application must be what the application uses. Most applications should be able to use one of the values specified for the *-dn-format attributes. If these are not adequate, the *-dn-delims attributes accept up to eight characters with the following meanings:

Character Number	Description
#1	Typed Name Boolean Flag: 0 means names are not typed, 1 means names are typed.
#2	Unicode No-Map Character Boolean Flag: 0 means don't output or interpret unmappable unicode characters as escaped hex digit strings, for example, \FEFF. The Identity Vault does not accept the following unicode characters: 0xFEFF, 0xFFFE, 0xFFFD, and 0xFFFF.
#3	Relative RDN delimiter.
#4	RDN delimiter.

#5	Name divider.
#6	Name value delimiter.
#7	Wildcard character.
#8	Escape character.

If the RDN delimiter and the Relative RDN delimiter are the same character, the orientation of the name is root right; otherwise, the orientation is root left.

Elements

<placement-rule>

Specifies the criteria for selecting an entry and for generating the distinguished name.

<match-attr>

Specifies attribute values the entry must match to use this placement rule.

<match-class>

Specifies the base class the entry must match to use this placement rule.

<match-path>

Specifies the root container the entry's dn must match to use this placement rule.

<placement>

Specifies where the entry is placed when it matches all the placement criteria.

Parent

- Top element

Sample Placement Rules

The following example shows three placement rules.

```
<placement-rules src-dn-format="slash" dest-dn-format="ldap" >
<!-- for Users coming from the subtree \Tree\novell in NDS -->
<!-- place them in the same relative hierarchy           -->
<!-- under o=novell                                       -->

   <placement-rule>
      <match-class class-name="User"/>
```

```
      <match-path prefix="\TREE\novell"/>
      <placement><copy-path-suffix/>,o=novell</placement>
   </placement-rule>

<!-- for all other users and groups                      -->
<!-- place them in the department container under novell -->

   <placement-rule>
      <match-class class-name="User"/>
      <match-class class-name="Group"/>
      <placement>cn=<copy-name/>,ou=<copy-attr
          attr-name="OU"/>,o=novell</placement>
   </placement-rule>

<!-- for everything else, try to mirror the hierarchy -->

   <placement-rule>
      <placement><copy-path/></placement>
   </placement-rule>
</placement-rules>
```

For another example, see the pub_placement_rule.xml file used by the publisher in the VRTest Driver.

<placement-rule>

Specifies the criteria for generating a distinguished name for a new entry.

Description

The Identity Manager engine uses the following algorithm to determine if the add operation can use a particular placement rule.

1. If the rule contains any <match-class> elements, the class-name attribute in the add operation must match one of the class-names specified in the <match-class> elements.

2. If the rule contains any <match-attr> elements, the add operation must contain an attribute value for each of the attributes specified in the <match-attr> elements.

3. If the rule contains a <match-path> element, the src-dn of the add operation must match one of the subtrees specified by the <match-path> element.

If a rule cannot be used, it is skipped, and the Identity Manager engine moves to the next rule.

Once a suitable rule is found, the Identity Manager engine uses the <placement> element to generate a value for the dest-dn attribute for the add operation. If no suitable rule is found, the dest-dn attribute of the add operation is left blank.

Definition

```
<!ELEMENT placement-rule (match-class*,
                          match-path*,
                          match-attr*,
                          placement)>
<!ATTLIST placement-rule
          description     CDATA            #IMPLIED>
```

Attributes

description

Provides a description of this placement rule for display by iManager.

Elements

<match-attr>

Specifies attribute values the entry must match to use this placement rule.

<match-class>

Specifies the base class the entry must match to use this placement rule.

<match-path>

Specifies the root container the entry's dn must match to use this placement rule.

<placement>

Specifies where the entry is placed when it matches all the placement criteria.

Parent

- <placement-rules>

<match-attr>

Specifies an attribute for create, placement, and matching rules.

Description

When the <match-attr> element is specified in a <placement-rule> element, at least one value must be specified.

Definition

```
<!ELEMENT match-attr       (value)+ >
<!ATTLIST match-attr
          attr-name        CDATA            #REQUIRED>
```

Attributes

attr-name

Specifies the name of the attribute in the Identity Vault name space.

Elements

<value>

Specifies the value for the attribute.

Parent

- <create-rule>
- <matching-rule>
- <placement-rule>

<match-class>

Specifies the base class that an entry must match.

Definition

```
<!ELEMENT match-class      EMPTY>
<!ATTLIST match-class
          class-name       CDATA            #REQUIRED>
```

Attributes

class-name

Specifies the base class in the Identity Vault name space.

Parent

- <matching-rule>
- <placement-rule>

<match-path>

Specifies a subtree in the directory that is used in matching the entry.

Description

When the <match-path> element is specified in a <placement-rule> element, the prefix attribute is compared to the src-dn attribute of the <add> element. The entry matches if the entry is in the subtree specified by the prefix. The name space of the prefix is the source of the add operation.

The format of the prefix is dependent upon the name space.

- For the Identity Vault, the format is slash, for example, \treename\container\container. If the leading slash is omitted, the path is assumed to be relative to the tree root of the Identity Vault server that is hosting Identity Manager.
- For the external application, the format is application dependent and must be documented by the driver writer for the system administrator.

Definition

```
<!ELEMENT match-path       EMPTY>
<!ATTLIST match-path
          prefix           CDATA             #REQUIRED>
```

Attributes

prefix

Specifies the distinguished name of a subtree in the directory.

Parent

- <matching-rule>
- <placement-rule>

<placement>

Specifies the distinguished name for the new entry.

Description

The dn is generated by concatenating together, in order, the text and the specified <elements>. Any leading or trailing white space is removed unless it is enclosed by a CDATA section.

Definition

```
<!ELEMENT placement        (#PCDATA |
                            copy-name |
                            copy-attr |
                            copy-path |
                            copy-path-suffix)* >

<!ELEMENT copy-attr         EMPTY>
<!ATTLIST copy-attr
          attr-name         CDATA      #REQUIRED>

<!ELEMENT copy-name         EMPTY>

<!ELEMENT copy-path         EMPTY>

<!ELEMENT copy-path-suffix EMPTY>
```

Attributes for <copy-attr>

attr-name

Specifies the attribute name.

Elements

<copy-name>

Specifies that the rdn of the src-dn attribute from the <add> element is copied and used as part of the dn. If the <copy-name> element is specified and the src-dn in the <add> element is empty, the placement rule is skipped.

<copy-attr>

Specifies that the first value of the attribute specified by attr-name is copied from the <add> element and used as part of the destination dn. If the attribute does not exist in the <add> element, the placement rule is skipped.

Structured attribute types are not supported.

<copy-path>

Specifies that the src-dn attribute from the <add> element is copied to be used as the destination dn. The Identity Manager engine used the src-dn-format and dest-dn-format attributes to convert the name to the proper format. Conversion from typeless format to a typed format is unsupported unless the source is the Identity Vault.

<copy-path-suffix>

Specifies that only a portion of the src-dn attribute from the <add> element is copied to be used as the destination dn. The portion of the src-dn attribute that matches the <match-path> element is stripped from the name. If no <match-path> element is specified, the whole src-dn is copied.

The Identity Manager engine used the src-dn-format and dest-dn-format attributes to convert the name to the proper format. Conversion from typeless format to a typed format is unsupported unless the source is the Identity Vault.

Parent

- <placement-rule>

8.5 Event Transformation Rules

Event transformation rules change an event from one type to another. They are processed in a slightly different order according to the channel, but the names of the attributes and classes are in the Identity Vault name space.

There are no required transformations that an event transformation style sheet must perform. They can be used for the following:

- Another XML element. For example, it can change a <delete> element to a <remove-association> element.
- An XML element to an external application element. For example, if the external application has an XML interface, the <add> element could be converted to the external application's XML <create> element. This type of transformation can only be done for the subscriber as commands come from the Identity Manager engine and the driver passes them to the external application.
- An external application element to an Identity Vault XML element. For example, if the external application has an XML interface, the external application's XML <create> element could be converted to the <add> element. This type of transformation can only be done for the publisher as events come from the external application and the driver passes them to the Identity Manager engine.
- Additional elements. For example, an <add> element could be split into an <add> and a <modify> element.
- Remove elements. For example, <add> and <modify> elements could be combined into just an <add> element.

- Custom command. If the external application does not have an XML interface, the XML elements could be transformed directly to a function or method from the application's interface.

The Identity Manager engine processes the event transformation rule according to the channel:

- For the subscriber channel which sends events from the Identity Vault to the driver, the engine applies the event transformation rules before any of the other rules or output style sheets.

- For the publisher channel which sends events from the driver to the engine, the engine applies the event transformation after the input transformation rules and the schema mapping rules, but before the matching, create, and placement rules.

The event transformation rules are kept in an DirXML-Rule object and associated with your driver through the DirXML-EventTransformationRule attribute of your driver's DirXML-Publisher and DirXML-Subscriber objects.

8.5.1 Sample Event Transformation Rule

The following sample is and event transformation for the subscriber channel that limits the scope of events.

```
<?xml version="1.0" encoding="UTF-8"?><!--
     This stylesheet is and event transformation for the
     subscriber channel that limits the scope of events.
-->
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

     <xsl:template match="input/*[@src-dn]">
          <xsl:variable name="src-dn" select="string(@src-dn)"/>
          <xsl:if test=  "starts-with($src-dn,'\TREE\novell\Provo\') or
                   starts-with($src-dn,'\TREE\novell\San Jose\') or
                               starts-with($src-dn,'\TREE\novell\Orem\') or
                               starts-with($src-dn,'\TREE\novell\Unknown Location\')
">
              <xsl:copy>
                <xsl:apply-templates select="@*|node()"/>
              </xsl:copy>
              </xsl:if>
     </xsl:template>

<!-- identity template for everything we don't specifically handle -->
     <xsl:template match="node()|@*">
              <xsl:copy>
                        <xsl:apply-templates select="@*|node()"/>
              </xsl:copy>
       </xsl:template>

</xsl:transform>
```

8.6 Command Transformation Rules

Command Transformation Rules provide final processing on commands before the commands are sent to the Identity Vault or the application

There are no required transformations that a command transformation style sheet must perform. They can be used for the following:

- Another XML element. For example, it can change a <delete> element to a <remove-association> element.
- An XML element to an external application element. For example, if the external application has an XML interface, the <add> element could be converted to the external application's XML <create> element. This type of transformation can only be done for the subscriber as commands come from the Identity Manager engine and the driver passes them to the external application.
- An external application element to an Identity Vault XML element. For example, if the external application has an XML interface, the external application's XML <create> element could be converted to the <add> element. This type of transformation can only be done for the publisher as events come from the external application and the driver passes them to the Identity Manager engine.
- Additional elements. For example, an <add> element could be split into an <add> and a <modify> element.
- Remove elements. For example, <add> and <modify> elements could be combined into just an <add> element.
- Custom command. If the external application does not have an XML interface, the XML elements could be transformed directly to a function or method from the application's interface.

The Identity Manager engine processes the command transformation rule according to the channel:

- For the subscriber channel which sends events from the Identity Vault to the driver, the engine applies the command transformation rules directly before the Schema Mapping Rule. Both the Schema Mapping Rule and the Output Transformation are executed after the Command Transformation Rule on the Subscriber channel.
- For the publisher channel which sends events from the driver to the engine, the engine applies the command transformation after all other rules, directly before the Identity Manager engine applies the commands in the command document to the Identity Vault.

The command transformation rules are kept in an DirXML-Rule object and associated with your driver through the DirXML-CommandTransformationRule attribute of your driver's DirXML-Publisher and DirXML-Subscriber objects.

8.6.1 Sample Command Transformation Rules

The following sample is a command transformation for the publisher channel that causes a change of location in the application to trigger a move in the Identity Vault.

```
<?xml version="1.0"?>
<!--
       This stylesheet is a command transformation for the publisher
     channel that causes a change of location in the application
         to trigger a move in eDirectory.
-->

<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

       <xsl:template match="modify[modify-attr[@attr-name = ’L’]]">
              <xsl:copy>
```

```
                <xsl:apply-templates select="@*|node()"/>
            </xsl:copy>
            <xsl:variable name="location" select="string(modify-attr[@attr-name
= 'L']/add-value/value)" />
            <xsl:variable name="target">
                <xsl:choose>
                    <xsl:when test="$location='San Jose' or
                                        $location='Orem' or
                                         $location='Provo'">
                     <xsl:value-of select="$location"/>
                    </xsl:when>
                    <xsl:otherwise>
                         <xsl:value-of select="'Unknown Location'"/>
                     </xsl:otherwise>
                 </xsl:choose>
             </xsl:variable>
             <move dest-dn="{@dest-dn}">
                  <parent dest-dn="novell\{$target}"/>
             </move>
        </xsl:template>

<!-- identity template for everything we don't specifically handle -->

     <xsl:template match="node()|@*">
            <xsl:copy>
                <xsl:apply-templates select="@*|node()"/>
            </xsl:copy>
        </xsl:template>

</xsl:transform>
```

8.7 Input Transformation Style Sheets

Input transformation rules perform any pre-processing of XML issued from the driver to the Identity Manager engine. This is a good place to perform data format mapping, for example changing a 15.2.1965 format for a birth date to a 2/15/65 format.

The input transformation rules are applied before the schema mapping transformation so the names of the attributes and classes are in the application's name space.

The style sheet can also be used for the following:

- Changing event types

- Generating extra events
- Removing events

There are no required transformations that an input transformation style sheet must perform.

The input transformation rule is stored in a DirXML-StyleSheet object and is linked to the driver through the DirXML-InputTransform attribute of the driver's DirXML-Driver object.

8.8 Output Transformation Style Sheets

Output transformation rules perform any post-processing of XML issued from the Identity Manager engine to the driver. This is a good place to perform data format mapping, for example changing a 2/15/65 format for a birth date to a 15.2.1965 format.

The output transformation rules are applied after the schema mapping transformation so the names of the attributes and classes are in the application's name space.

The style sheet can also be used for the following:

- Changing event types
- Generating extra events
- Removing events

There are no required transformations that an output transformation style sheet must perform.

The output transformation rule is stored in a DirXML-StyleSheet object and is linked to the driver through the DirXML-OutputTransform attribute of the driver's DirXML-Driver object.

9.0 Style Sheets

Style sheets define XSLT transformation rules. The XSLT processor in the Identity Manager engine is compliant with the 16 November 1999 W3C Recommendation. For the specifications, see the following:

- XSL Transformations (XSLT)
- XML Path Language (XPath)

Style sheets can be used in the following places:

- Input transformation rules
- Output transformation rules
- Event transformation rules
- Matching, create, or placement rules
- Mapping rules

The following sections describe the implementation specifics of using style sheets with Identity Manager.

- Section 9.1, Restrictions
- Section 9.2, Starting with an Identity Transformation

- Section 9.3, Using the Parameters that Identity Manager Passes
- Section 9.4, Using Extension Functions
- Section 9.5, Testing Style Sheets Outside of Identity Manager
- Section 9.6, Invoking the Novell XSLT Processor Directly
- Section 9.7, Creating a Password Example: Create Rule
- Section 9.8, Creating an Identity Vault User Example: Create Rule

9.1 Restrictions

Three of the rule types (matching, create, and placement) can be also be XML documents. When these rules are written as style sheets, they are subject to the following restrictions.

9.1.1 Matching Rule Restrictions

When matching rules are written as an XSLT style sheet, they are subject to the following restrictions:

- Use the special value of a single Unicode character 0xFFFD to signal that multiple matches were found.
- Can operate only on add events.
    - On the subscriber channel, the Identity Manager driver must add an <association> element for any matches that are found in the application.
    - On the publisher channel, the Identity Manager driver must fill in the dest-dn attribute of the <add> element if a match is found in the Identity Vault.
- Can remove events
- Cannot generate extra events
- Cannot change event types

The names of the attributes and classes are in the Identity Vault name space.

9.1.2 Create Rule Restrictions

When create rules are written as an XSLT style sheet, they are subject to the following restrictions:

- Can operate only on add events.
- Can add attributes and values to the <add> element.
- Can remove events (this is how an add event is vetoed).

The names of the attributes and classes are in the Identity Vault name space.

9.1.3 Placement Rule Restrictions

When placement rules are written as an XSLT style sheet, they are subject to the following restrictions:

- Can operate only on add events.
- Must fill in the dest-dn attribute of the <add> element.
- Can remove events.

The names of the attributes and classes are in the Identity Vault name space.

9.2 Starting with an Identity Transformation

Unless you are translating to or from an XML format that is completely different from the Identity Manager format, you will want to start your style sheet with templates that implement the identity transformation. These templates allow the events in the document that you don't specifically try to intercept and change to pass through without any modifications.

The following two templates together implement the identity transformation:

```
<xsl:template match="/" >
    <xsl:apply-templates select="node()|@*"/>
</xsl:template>

<xsl:template match="node()|@*" >
    <xsl:copy>
       <xsl:apply-templates select="node()|@*"/>
    </xsl:copy>
</xsl:template>
```

9.3 Using the Parameters that Identity Manager Passes

The Identity Manager engine passes the rule style sheets the following parameters that the style sheet can use. Note that with Identity Manager 4.5, the query processor parameters are now passed to the schema mapping rules and the input and output transformation rules. The command processor parameters are passed to all rules.

- fromNds—This is a boolean value that is true if the rule is being processed by the subscriber channel and false if the rule is being processed by the publisher channel.
- srcQueryProcessor—This is a Java object that implements the XdsQueryProcessor interface. This allows the style sheet to query the event source for more information.
- destQueryProcessor—This is a Java object that implements the XdsQueryProcessor interface. This allows the style sheet to query the event target for more information.
- srcCommandProcessor—This is a java object that implements the XdsCommandProcessor interface. This allows the stylesheet to "write-back" a command to the event source. Not available in Identity Manager 1.0.
- destCommandProcessor-This is a java object that implements the XdsCommandProcessor interface.This allows the stylesheet to issue a command to the command destination directly, bypassing most other rules. Not available in Identity Manager 1.0.

To use these parameters include the following in your style sheet:

- <xsl:param name="fromNds"/>
- <xsl:param name="srcQueryProcessor"/>
- <xsl:param name="destQueryProcessor"/>
- <xsl:param name="srcCommandProcessor"/>
- <xsl:param name="destCommandProcessor"/>

With Identity Manager 4.5, processors will accept a query or command element as the top level element and will wrap it in <input> and <nds> if necessary.

When using the query and command parameters with the schema mapping rules, input transformation rules, and output transformation rules the following limitations apply:

1. Queries issued to the application shim must be in the form expected by the application shim. In other words, schema names must be in the application namespace and the query must conform to whatever XML vocabulary is used natively by the shim. No association refs will be added to the query.

2. Responses from the application shim will be in the form returned by the shim with no modification or schema mapping performed and no resolution of association refs.

3. Queries issued to NDS must be in the form expect by NDS. In other words schema names must be in the NDS namespace and the query must be XDS. Association refs will not be resolved.

4. Responses from the application shim will be in the form returned by the shim with no modification or schema mapping performed.

Query Processors

Use of the query processors depends on the Novell XSLT implementation of extension functions. To make a query, you need to declare a name space for the XdsQueryProcessor interface. This is done by adding the following to the <xsl:stylesheet> or <xsl:transform> element of the style sheet.

```
xmlns:query="http://www.novell.com/nxsl/java/com.novell.nds.dirxml.driver.XdsQueryP
rocessor"
```

The following example uses one of the query processors (the extra long lines are wrapped and do not begin with a <):

```
<!-- Query object name queries NDS for the passed object -->

<!-- name. Ideally, this would not depend on "CN": to do -->

<!-- this, add another parameter that is the name of the   -->

<!-- naming attribute.                                     -->


<xsl:template name="query-object-name">

   <xsl:param name="object-name"/>


<!-- build an xds query as a result tree fragment -->

   <xsl:variable name="query">

      <nds ndsversion="8.5" dtdversion="1.0">

         <input>

            <query>

               <search-class class-name="{ancestor-or-self:

                     :add/@class-name}"/>
```

```
<!-- NOTE: depends on CN being the naming attribute -->
               <search-attr attr-name="CN">
                   <value><xsl:value-of select="$object-name"/
                        ></value>
               </search-attr>
<!-- put an empty read attribute in so that we don't get -->
<!-- the whole object back                               -->
               <read-attr/>
            </query>
         </input>
      </nds>
   </xsl:variable>

<!-- query NDS -->
<xsl:variable name="result" select="query:query($destQuery
     Processor,$query)"/>

<!-- return an empty or non-empty result tree fragment -->
<!-- depending on result of query                      -->
   <xsl:value-of select="$result//instance"/>
</xsl:template>
```

Command Parameters

In order to allow channel write-back for default attributes added by a create rule, a new XML attribute called write-back was added to the <required-attr> element of the Create Rule. If present and set to true, the create rule will call the srcCommandProcessor with a modify command to write the default value back to the source.

The following example uses command parameters to perform a write back operation.

```
<?xml version="1.0"?>
<xsl:transform
           version="1.0"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
       xmlns:cmd="http://www.novell.com/nxsl/java
       com.novell.nds.dirxml.driver.XdsCommandProcessor"
>
<xsl:param name="srcCommandProcessor"/>

<xsl:template match="node()|@*">
     <xsl:copy>
              <xsl:apply-templates select="@*|node()"/>
```

```
        </xsl:copy>
</xsl:template>

<xsl:template match="add">
        <xsl:copy>
                <xsl:apply-templates select="@*|node()"/>
        </xsl:copy>

  <!-- on a user add, add Engineering department to the source object -->
  <xsl:variable name="dummy">
         <modify class-name="{@class-name} "dest-dn="{@src-dn}">
                <xsl-copy-of select="association"/>
                <modify-attr attr-name="OU">
                     <add-value>
                              <value type="string">Engineering</value>
                        </add-value>
                </modify-attr>
    </modify>
  </xsl:variable>
        <xsl:variable name="dummy2"
          select="cmd:execute($srcCommandProcessor, $dummy)"/>
</xsl:template>

</xsl:transform>
```

9.4 Using Extension Functions

XSLT is an excellent tool for performing some kinds of transformations and a rather poor tool for other types of transformations such as non-trivial string manipulation and iterative processes. Fortunately the Novell XSLT processor implements extension functions which allow the style sheet to call a function implemented in Java, and by extension, any other language that can be accessed through JNI.

For specific examples, see the above example using the query processor, and the following example that illustrates using Java for string manipulation (the extra long lines are wrapped and do not begin with a <).

```
<!-- get-dn-prefix places the part of the passed dn that -->
<!-- precedes the last occurrence of '\' in the passed dn -->
<!-- in a result tree fragment meaning that it can be    -->
<!-- used to assign a variable value                     -->

<xsl:template name="get-dn-prefix" xmlns:jstring="http://
     www.novell.com/nxsl/java/java.lang.String">
```

```
   <xsl:param name="src-dn"/>

<!-- use java string stuff to make this much easier -->
   <xsl:variable name="dn" select="jstring:new($src-dn)"/>
   <xsl:variable name="index" select="jstring:lastIndexOf
       ($dn,'\')"/>
   <xsl:if test="$index != -1">
      <xsl:value-of select="jstring:substring($dn,0,$index)
          "/>
   </xsl:if>
</xsl:template>
```

9.5 Testing Style Sheets Outside of Identity Manager

The XSLT process in the Identity Manager engine may be invoked from the command line and can be used to test stylesheets in a more controlled environment before installing them into Identity Manager.

The following batch file may be used to invoke the XSLT processor on NT or Windows 2000.

```
@echo off
setlocal
rem TODO - edit the following line to point to directory where NDS and DirXML are
installed

set DIRXML_HOME=c:\novell\nds
set COMMON_JARS=%DIRXML_HOME%\lib%DIRXML_HOME%\jre\bin\java -
classpath%COMMON_JARS%\xp.jar; %COMMON_JARS%\collections.jar; %COMMON_JARS%\nxsl.ja
r com.novell.xsl.nxsl %1 %2 %3 %4 %5 %6 %7 %8 %9

endlocal
```

Invoking the processor without any arguments prints out the latest information on the command syntax for the processor.

To get sample XML to use as input to test your style sheet, use the tracing options (see Section 3.1, Using DSTrace and the Identity Manager Trace Log) and paste the appropriate XML into a text file.

Since you are running outside of Identity Manager, the srcQueryProcessor and destQueryProcessor will not be available. To get around this limitation, you can temporarily comment out code that uses the query processor and replace it with an explicit assignment of the reply you might expect from the query. For example:

```
<!-- query NDS -->
<!-- <xsl:variable name="result" select="query:query($destQueryProcessor, $query)"/>
-->

<!-- simulate query results -->

<xsl:variable name="result">
```

```
    <nds dtdversion="1.0" ndsversion="8.5">
       <output>
         <instance class-name="User" src-dn="\MY_TREE \MY_ORG\Fred"/>
         <status event-id="" level="success"></status>
       </output>
    </nds>
<xsl:variable>
```

9.6 Invoking the Novell XSLT Processor Directly

Drivers can invoke the Novell XSLT processor directly by using the com.novell.xsl.StyleSheet class. There are various ways in which this can be set up and invoked, but a typical invocation is illustrated by the following code fragment.

```
import com.novell.nds.dirxml.driver.*;
import com.novell.xsl.*;
import com.novell.xsl.result.*;
import org.w3c.dom.*;
.
.
.
   try
   {
      XmlDocument inputDoc = new XmlDocument();
      XmlDocument stylesheetDoc = new XmlDocument();

      // load the input and stylesheet documents from a file
      inputDoc.readDocument(new FileInputStream("input.xml"));
      stylesheetDoc.readDocument(new FileInputStream("stylesheet.xsl"));

      // create the stylesheet processor and give it the stylesheet
      Stylesheet  styleSheet = new Stylesheet();
      styleSheet.load(stylesheetDoc.getDocument());

      // pass in any stylesheet parameters that the stylesheet might need
      styleSheet.setParameter("fromNds", new Boolean(fromNds));

      // setup a result handler to get a DOM tree
      Document resultDoc = com.novell.xml.dom.DocumentFactory.newDocument();
      DOMResultHandler resultHandler = new DOMResultHandler(resultDoc);
      styleSheet.setResultHandler(resultHandler);
```

```
      // apply the stylesheet
      styleSheet.process(doc, null);

      // result will be in resultDoc

   }
   catch( XSLException xsle )
   {
      // handle any exception thrown
   }
```

9.7 Creating a Password Example: Create Rule

The following style sheet can be used for a create rule. It creates a user, generates a password for the user from the user's Surname and CN attributes, and performs an identity transform (which passes through everything in the document except the events you are trying to intercept and transform).

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!--   This stylesheet has an example of how to replace a create rule with
        an XSLT stylesheet and supply an initial password for "User" objects. -->

<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform
     "version="1.0">

<!-- ensure we have required NDS attributes -->
<xsl:template match="add">
   <xsl:if test="add-attr[@attr-name='Surname'] and
                 add-attr[@attr-name='CN']">
      <!-- copy the add through -->
      <xsl:copy>
         <xsl:apply-templates select="@*|node()"/>
         <!-- add a <password> element -->
         <xsl:call-template name="create-password"/>
      </xsl:copy>
   </xsl:if>

<!-- if the xsl:if fails, we don't have all the required attributes
      so we won't copy the add through, and the create rule will veto the add -->
```

```
</xsl:template>

<xsl:template name="create-password">
   <password>
      <xsl:value-of select="concat(add-attr[@attr-name='Surname']/value,
           '-',add-attr[@attr-name='CN']/value)"/>
   </password>
</xsl:template>

<!-- identity transform for everything we don't want to change -->

<xsl:template match="@*|node()">
    <xsl:copy>
        <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
</xsl:template>

</xsl:transform>
```

9.8 Creating an Identity Vault User Example: Create Rule

This style sheet can be used for a create rule. It shows how to create an Identity Vault user from an entry created in an external application. This example is based on the idea that a newly hired person is first created in the Human Resources database and then on the network. It takes the user's first name and last name and generates a unique CN in the Identity Vault. Although Identity Vault requires the CN to be unique in only the container, this style sheet ensures that it is unique across all containers in the Identity Vault.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!--  This stylesheet is an example of how to replace a create rule with an
      XSLT stylesheet and that creates the User name from the Surname and
      given Name attributes -->

<xsl:transform
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
   xmlns:query="http://www.novell.com/nxsl/java/com.novell.nds.dirxml.driver.
         XdsQueryProcessor"
   >

<!-- This is for testing the stylesheet outside of DirXML so things
     are pretty to look at -->
<xsl:strip-space elements="*"/>
```

```
<xsl:preserve-space elements="value,component"/>
<xsl:output method="xml" indent="yes"/>

<!-- dirxml always passes two stylesheet parameters to an XSLT rule:
     an inbound and outbound query processor -->
<xsl:param name="srcQueryProcessor"/>
<xsl:param name="destQueryProcessor"/>

<!-- match <add> elements -->
<xsl:template match="add">

   <!-- ensure we have required NDS attributes we need for the name -->
   <xsl:if test="add-attr[@attr-name='Surname'] and
                 add-attr[@attr-name='Given Name']">

      <!-- copy the add through -->
      <xsl:copy>
         <!-- copy any attributes through except for the src-dn -->
         <!-- we'll construct the src-dn below so that the placement rule will work
-->
         <xsl:apply-templates select="@*[string(.) != 'src-dn']"/>

         <!-- call a template to construct the object name and place the result in
a variable -->
         <xsl:variable name="object-name">
            <xsl:call-template name="create-object-name"/>
         </xsl:variable>

         <!-- now create the src-dn attribute with the created name -->
         <xsl:attribute name="src-dn">
            <xsl:variable name="prefix">
               <xsl:call-template name="get-dn-prefix">
                  <xsl:with-param name="src-dn" select="string(@src-dn)"/>
               </xsl:call-template>
            </xsl:variable>
            <xsl:value-of select="concat($prefix,'\',$object-name)"/>
         </xsl:attribute>

         <!-- if we have a "CN" attribute, set it to the constructed name -->
         <xsl:if test="./add-attr[@attr-name='CN']">
```

```
            <add-attr attr-name="CN">
               <value type="string"><xsl:value-of select="$object-name"/></value>
            </add-attr>
         </xsl:if>

         <!-- copy the rest of the stuff through, except for what we have already
copied -->
         <xsl:apply-templates select="*[name() != 'add-attr' or @attr-name != 'CN']
|
                                      comment() |
                                       processing-instruction() |
                                        text()"/>

         <!-- add a <password> element -->
         <xsl:call-template name="create-password"/>

      </xsl:copy>
   </xsl:if>
   <!-- if the xsl:if fails, it means we don't have all the required attributes
        so we won't copy the add through, and the create rule will veto the add -->
</xsl:template>

<!-- get-dn-prefix places the part of the passed dn that precedes the   -->
<!-- last occurrance of '\' in the passed dn in a result tree fragment   -->
<!-- meaning that it can be used to assign a variable value               -->
<xsl:template name="get-dn-prefix"
xmlns:jstring="http://www.novell.com/nxsl/java/java.lang.String">
   <xsl:param name="src-dn"/>

   <!-- use java string stuff to make this much easier -->
   <xsl:variable name="dn" select="jstring:new($src-dn)"/>
   <xsl:variable name="index" select="jstring:lastIndexOf($dn,'\')"/>
   <xsl:if test="$index != -1">
      <xsl:value-of select="jstring:substring($dn,0,$index)"/>
   </xsl:if>
</xsl:template>

<!-- create-object-name creates a name for the user object and places the   -->
<!-- result in a result tree fragment                                        -->
<xsl:template name="create-object-name">
```

```
   <!-- first try is first initial followed by surname -->
   <xsl:variable name="given-name" select="add-attr[@attr-name='Given
Name']/value"/>
   <xsl:variable name="surname" select="add-attr[@attr-name='Surname']/value"/>
   <xsl:variable name="prefix" select="substring($given-name,1,1)"/>
   <xsl:variable name="object-name" select="concat($prefix,$surname)"/>

   <!-- then see if name already exists in NDS -->
   <xsl:variable name="exists">
      <xsl:call-template name="query-object-name">
         <xsl:with-param name="object-name" select="$object-name"/>
      </xsl:call-template>
   </xsl:variable>

   <!-- if exists, then try 1st fallback, else return result -->
   <xsl:choose>
      <xsl:when test="$exists != ''">
         <xsl:call-template name="create-object-name-2"/>
      </xsl:when>
      <xsl:otherwise>
         <xsl:value-of select="$object-name"/>
      </xsl:otherwise>
   </xsl:choose>

</xsl:template>

<!-- create-object-name-2 is the first fallback if the name created by      -->
<!-- create-object-name already exists                                       -->
<xsl:template name="create-object-name-2">

   <!-- first try is first name followed by surname -->
   <xsl:variable name="given-name" select="add-attr[@attr-name='Given
Name']/value"/>
   <xsl:variable name="surname" select="add-attr[@attr-name='Surname']/value"/>
   <xsl:variable name="object-name" select="concat($given-name,$surname)"/>

   <!-- then see if name already exists in NDS -->
   <xsl:variable name="exists">
      <xsl:call-template name="query-object-name">
```

```
      <xsl:with-param name="object-name" select="$object-name"/>
    </xsl:call-template>
  </xsl:variable>

  <!-- if exists, then try last fallback, else return result -->
  <xsl:choose>
    <xsl:when test="$exists != ''">
      <xsl:call-template name="create-object-name-fallback"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="$object-name"/>
    </xsl:otherwise>
  </xsl:choose>

</xsl:template>

<!-- create-object-name-fallback recursively tries a name created by      -->
<!-- concatenating the surname and a count until NDS doesn't find         -->
<!-- the name. There is a danger of infinite recursion, but only if      -->
<!-- there is a bug in NDS                                                -->
<xsl:template name="create-object-name-fallback">
  <xsl:param name="count" select="1"/>

  <!-- construct the a name based on the surname and a count -->
  <xsl:variable name="surname" select="add-attr[@attr-name='Surname']/value"/>
  <xsl:variable name="object-name" select="concat($surname,'-',$count)"/>

  <!-- see if it exists in NDS -->
  <xsl:variable name="exists">
    <xsl:call-template name="query-object-name">
      <xsl:with-param name="object-name" select="$object-name"/>
    </xsl:call-template>
  </xsl:variable>

  <!-- if exists, then try again recursively, else return result -->
  <xsl:choose>
    <xsl:when test="$exists != ''">
      <xsl:call-template name="create-object-name-fallback">
        <xsl:with-param name="count" select="$count + 1"/>
```

```
            </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
            <xsl:value-of select="$object-name"/>
        </xsl:otherwise>
    </xsl:choose>

</xsl:template>

<!-- query object name queries NDS for the passed object-name. Ideally, this would
-->
<!-- not depend on "CN": to do this, add another parameter that is the name of the
-->
<!-- naming attribute.
-->
<xsl:template name="query-object-name">
    <xsl:param name="object-name"/>

    <!-- build an xds query as a result tree fragment -->
    <xsl:variable name="query">
        <nds ndsversion="8.5" dtdversion="1.0">
            <input>
                <query>
                    <search-class class-name="{ancestor-or-self::add/@class-name}"/>
                    <!-- NOTE: depends on CN being the naming attribute -->
                    <search-attr attr-name="CN">
                        <value><xsl:value-of select="$object-name"/></value>
                    </search-attr>
                    <!-- put an empty read attribute in so that we don't get the whole
object back -->
                    <read-attr/>
                </query>
            </input>
        </nds>
    </xsl:variable>

    <!-- query NDS -->
    <xsl:variable name="result" select="query:query($destQueryProcessor,$query)"/>

    <!-- return an empty or non-empty result tree fragment depending on result of
query -->
```

```
   <xsl:value-of select="$result//instance"/>
</xsl:template>

<!-- create an initial password -->
<xsl:template name="create-password">
   <password>
      <xsl:value-of select="concat(add-attr[@attr-name='Surname']/value,'-',add-attr[@attr-name='CN']/value)"/>
   </password>
</xsl:template>

<!-- identity transform for everything we don't want to mess with -->
<xsl:template match="@*|node()">
    <xsl:copy>
        <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
</xsl:template>

</xsl:transform>
```

10.0 Identity Manager Error Codes

The following is a list of Identity Manager specific error codes:

Code	Error
-300	VR_ERR_NO_JRE
-299	VR_ERR_JRE_LOAD_FAIL
-298	VR_ERR_JVM_INIT_FAIL
-297	VR_ERR_JVM_CREATE_FAIL
-296	VR_ERR_VRDRIVER_MISSING
-295	VR_ERR_VRDRIVER_CREATE_FAIL
-294	VR_ERR_VRDRIVER_INTERFACE_MISMATCH
-293	VR_ERR_MEMORY_ERR
-292	VR_ERR_FATAL
-291	VR_ERR_TIMEOUT

-290	VR_ERR_INVALID_COUNT
-289	VR_ERR_ALREADY_EXISTS
-288	VR_ERR_OPEN_COMM
-287	VR_ERR_CLOSE_COMM
-286	VR_ERR_PREEMPT_COMM

11.0 Javadoc, FAQs, and DTD Reference

The following link opens the Identity Manager Driver Developer Kit reference material: Identity Manager Reference

The Identity Manager Driver Developer Kit reference material contains five sections:

- *Identity Manager Javadoc.* This section contains Javadoc for the Identity Manager Java APIs.
- *C++ Reference.* This section contains C++ Reference for the Identity Manager C++ APIs.
- *XDS Libraries Javadoc.* This section contains Javadoc for the XDS Libraries.
- *XDS Libraries C++ Reference.* This section contains C++ Reference for the XDS Libraries.
- *XDS DTD and HTML Reference.* This section contains a copy of the XDS DTD and HTML DTD reference, which enables you to view DTD descriptions and hierarchy in HTML format.
-

A.0 VRTest Application

The VRTest application was created to test the Identity Manager infrastructure and to provide an example of a Identity Manager driver. It consists of four components:

- VRTestServer—a configurable server that acts as the external application. Configuration files have been set up so that it can simulate either a flat or a hierarchical directory.
- VRTestClient—a client application that allows you to view and manipulate server data. From this application, you can create, delete, and modify records. You can then watch these modifications as they are synchronized into Identity Vault.
- VRTestAPI—the programming interface for communicating with the VRTestServer application. Both vrtest_driver and VRTestClient use this interface. You can compare this interface to the interface available from your external application to determine if your application's interface contains the functionality required for a Identity Manager driver.
- vrtest_driver—the Identity Manager driver that connects Identity Vault and the VRTestServer and allows data to synchronize between the two. The vrtest_driver is available as sample code in C++ and Java.

A.1 Requirements and Installation

The VRTestServer application requires a Win32 client or server running eDirectory 8.8.8 or higher. To install the application and driver, follow the instructions in the VRTest section contained in Section 11.0, Javadoc, FAQs, and DTD Reference. For general information on installing drivers see the current Identity Manager documentation.

B.0 Identity Manager Definitions for the Schema

This appendix defines the attribute and object class definitions that are created in the Identity Vault schema to enable Identity Manager drivers.

B.1 Identity Manager Object Class Definitions

To enable communication between the Identity Manager engine and the Identity Manager driver, three objects must be created in Identity Vault within the DirXML-DriverSet object when the driver is installed. If the Identity Manager driver is the first Identity Manager driver installed in the Identity Vault, you will need to create the DirXML-DriverSet object. This object holds the DirXML-Driver object which holds the objects for the publisher and the subscriber shims: DirXML-Subscriber and DirXML-Publisher.

By creating three objects for each Identity Manager driver, the administrator is allowed complete flexibility in setting up the interaction between the Identity Vault and the external application. The following figure illustrates the possible placement of these objects.

Figure B-1 Possible placement of Identity Manager objects

The figure shows the typical configuration with rules supplied for both shims. However, the shims can share the same rules. The driver is associated with a rule object by storing the distinguished name of the rule in one of the driver's attributes (one exists for each type of rule). Shims share the same rule by having the same distinguished name of the rule object in the attribute for that type of rule.

The shims can also have rules which transform events from one event to another. The event rules on the subscriber transform an Identity Vault event into an event for the external application. The event rules on the publisher transform an external application event into an Identity Vault event.

Style sheets can be placed on the driver object. The InputTransform style sheet uses XSLT processing to transform external application events into Identity Vault events. The OutputTransform style sheet using XSLT process to transform an Identity Vault event into an external application event.

The Identity Manager engine performs all the processing of the rules and style sheets. It knows how to obtain the information from the Identity Vault objects and attributes. The driver developer needs to supply values for these rules and the optional style sheets.

Identity Manager-Driver

Contains the configuration attributes and objects for a single Identity Manager driver.

ASN.1 ID

- 2.16.840.1.113719.1.14.6.1.2

Class Flags

Class Flags	Setting
Container	On

Effective	On
Nonremovable	On
Ambiguous Naming	Off
Ambiguous Container	Off
Auxiliary Class	Off

Class Structure

Rule	Class/Attribute	Defined For
Super Classes	Top	DirXML-Driver
Containment	DirXML-DriverSet	DirXML-Driver
Named By	CN (Common Name)	DirXML-Driver

Mandatory Attributes

Identity Manager-Driver	Inherited from Top
CN	Object Class

Optional Attributes

Identity Manager-Driver

- DirXML-ApplicationSchema
- DirXML-DriverCacheLimit
- DirXML-DriverStartOption
- DirXML-DriverStorage
- DirXML-InputTransform
- DirXML-JavaModule
- DirXML-MappingRule
- DirXML-NativeModule
- DirXML-OutputTransform
- DirXML-ShimAuthID
- DirXML-ShimAuthPassword
- DirXML-ShimAuthServer
- DirXML-ShimConfigInfo
- DirXML-State
- Private Key
- Public Key
- Security Equals

Inherited from Top

- ACL
- Audit:File Link
- Authority Revocation
- Back Link (Attribute)
- Bindery Property
- CA Private Key
- CA Public Key
- Certificate Revocation
- Certificate Validity Interval
- Equivalent To Me
- GUID
- Last Referenced Time
- MASV:Authorized Range
- MASV:Default Range
- MASV:Proposed Label
- Obituary
- Other GUID
- Reference

- Cross Certificate Pair
- DirXML-Association
- Revision
- Used By

Default ACL Template

Object Name	Default Rights	Affected Attributes	Class Defined For
[Creator]	Supervisor	[Entry Rights]	Top

Remarks

This object can contain two application shim objects, DirXML-Subscriber and DirXML-Publisher. These objects are associated with the Identity Manager object by their containment. If they are placed in another directory, the driver looses its association with them.

If the subscriber or publisher objects are created with iManager, iManager places them under the driver the administrator is configuring.

Identity Manager-DriverSet

Contains all the drivers that are applicable for a given server.

ASN.1 ID

- 2.16.840.1.113719.1.14.6.1.1

Class Flags

Class Flags	Setting
Container	On
Effective	On
Nonremovable	On
Ambiguous Naming	Off
Ambiguous Container	Off
Auxiliary Class	Off

Class Structure

Rule	Class/Attribute	Defined For
Super Classes	Top	Identity Manager-DriverSet
Containment	Country	Identity Manager-DriverSet
	domain	Identity Manager-DriverSet
	Locality	Identity Manager-DriverSet
	Organization	Identity Manager-DriverSet
	Organizational Unit	Identity Manager-DriverSet
Named By	CN (Common Name)	Identity Manager-DriverSet

Mandatory Attributes

Identity Manager-DriverSet	Inherited from Top
CN	Object Class

Optional Attributes

DirXML-DriverSet

- DirXML-DriverTraceLevel
- DirXML-JavaDebugPort
- DirXML-JavaTraceFile
- DirXML-ServerList
- DirXML-XSLTraceLevel

Inherited from Top

- ACL
- Audit:File Link
- Authority Revocation
- Back Link (Attribute)
- Bindery Property
- CA Private Key
- CA Public Key
- Certificate Revocation
- Certificate Validity Interval
- Cross Certificate Pair
- DirXML-Association
- Equivalent To Me
- GUID
- Last Referenced Time
- MASV:Authorized Range
- MASV:Default Range
- MASV:Proposed Label
- Obituary
- Other GUID
- Reference
- Revision
- Used By

Default ACL Template

Object Name	Default Rights	Affected Attributes	Class Defined For
[Creator]	Supervisor	[Entry Rights]	Top

Remarks

This object should be defined as its own partition, so that a replica can be placed on the Identity Vault servers that are using the driver set. With this configuration, servers not using the driver set are not involved with synchronizing the data in the Identity Manager partition.

DirXML-Publisher

Contains the required information that allows an external application to synchronized selected data with the Identity Vault.

ASN.1 ID

- 2.16.840.1.113719.1.14.6.1.3

Class Flags

Class Flags	Setting
Container	On
Effective	On
Nonremovable	On
Ambiguous Naming	Off
Ambiguous Container	Off
Auxiliary Class	Off

Class Structure

Rule	Class/Attribute	Defined For
Super Classes	Top	DirXML-Publisher
Containment	DirXML-Driver	DirXML-Publisher
Named By	CN (Common Name)	DirXML-Publisher

Mandatory Attributes

DirXML-Publisher	Inherited from Top
CN	Object Class

Optional Attributes

DirXML-Publisher

- DirXML-CreateRule
- DirXML-DriverFilter
- DirXML-EventTransformationRule
- DirXML-MatchingRule
- DirXML-PlacementRule
- Private Key
- Public Key

Inherited from Top

- ACL
- Audit:File Link
- Authority Revocation
- Back Link (Attribute)
- Bindery Property
- CA Private Key
- CA Public Key
- Certificate Revocation
- Certificate Validity Interval
- Cross Certificate Pair
- DirXML-Association
- Equivalent To Me
- GUID
- Last Referenced Time
- MASV:Authorized Range
- MASV:Default Range
- MASV:Proposed Label
- Obituary
- Other GUID
- Reference
- Revision
- Used By

Default ACL Template

Object Name	Default Rights	Affected Attributes	Class Defined For
[Creator]	Supervisor	[Entry Rights]	Top

Remarks

This object can contain rule and style sheet objects which further define the format and rules that Identity Manager engine enforces when it sends data from the external application to the Identity Vault.

DirXML-Rule

Contains rule information which controls the behavior of the DirXML-Driver, DirXML-Publisher, or DirXML-Subscriber.

ASN.1 ID

- 2.16.840.1.113719.1.14.6.1.7

Class Flags

Class Flags	Setting
Container	Off
Effective	On
Nonremovable	On
Ambiguous Naming	Off
Ambiguous Container	Off
Auxiliary Class	Off

Class Structure

Rule	Class/Attribute	Defined For
Super Classes	Top	DirXML-Rule
Containment	Identity Manager-Driver	DirXML-Rule
	DirXML-Publisher	DirXML-Rule
	DirXML-Subscriber	DirXML-Rule
Named By	CN (Common Name)	DirXML-Rule

Mandatory Attributes

DirXML-Rule	Inherited from Top
CN	Object Class

Optional Attributes

DirXML-Rule

- XmlData
-

Inherited from Top

- ACL
- Audit:File Link
- Equivalent To Me
- GUID

- Authority Revocation
- Back Link (Attribute)
- Bindery Property
- CA Private Key
- CA Public Key
- Certificate Revocation
- Certificate Validity Interval
- Cross Certificate Pair
- DirXML-Association
- Last Referenced Time
- MASV:Authorized Range
- MASV:Default Range
- MASV:Proposed Label
- Obituary
- Other GUID
- Reference
- Revision
- Used By

Default ACL Template

Object Name	Default Rights	Affected Attributes	Class Defined For
[Creator]	Supervisor	[Entry Rights]	Top

Remarks

A rule object contains a particular type of rule in an XDS format defined for that rule type. Rule objects can contain mapping, matching, create, and placement rules. XDS is a subset of XML and is defined in the nds.dtd which specifies the formats for the rules.

DirXML-StyleSheet

Contains the XSL data that can transform data from one format to another, for example from XDS format to the external application format.

ASN.1 ID

- 2.16.840.1.113719.1.14.6.1.6

Class Flags

Class Flags	Setting
Container	Off
Effective	On
Nonremovable	On
Ambiguous Naming	Off
Ambiguous Container	Off
Auxiliary Class	Off

Class Structure

Rule	Class/Attribute	Defined For
Super Classes	StyleSheet	DirXML-StyleSheet
Containment	DirXML-Driver	DirXML-StyleSheet
	DirXML-Publisher	DirXML-StyleSheet
	DirXML-Subscriber	DirXML-StyleSheet
Named By	CN (Common Name)	StyleSheet

Mandatory Attributes

DirXML-StyleSheet	Inherited from StyleSheet	Inherited from Top
(None)	CN	Object Class

Optional Attributes

DirXML-StyleSheet

- (None)

Inherited from StyleSheet

- XmlData

Inherited from Top

- ACL
- Audit:File Link
- Authority Revocation
- Back Link (Attribute)
- Bindery Property
- CA Private Key
- CA Public Key
- Certificate Revocation
- Certificate Validity Interval
- Cross Certificate Pair
- DirXML-Association
- Equivalent To Me
- GUID
- Last Referenced Time
- MASV:Authorized Range
- MASV:Default Range
- MASV:Proposed Label
- Obituary
- Other GUID
- Reference
- Revision
- Used By

Default ACL Template

Object Name	Default Rights	Affected Attributes	Class Defined For
[Creator]	Supervisor	[Entry Rights]	Top

Remarks

Attributes associate a particular style sheet with a driver object. The DirXML-Driver contains two attributes (DirXML-InputTransform and DirXML-OutputTransform) which can associate both a input and output style sheet with the driver. The DirXML-Publisher and DirXML-Subscriber objects use the DirXML-EventTransformationRule attribute to form an association with a style sheet.

DirXML-Subscriber

Contains the required information that allows Identity Vault to synchronize selected data to an external application.

ASN.1 ID

- 2.16.840.1.113719.1.14.6.1.4

Class Flags

Class Flags	Setting
Container	On
Effective	On
Nonremovable	On
Ambiguous Naming	Off
Ambiguous Container	Off
Auxiliary Class	Off

Class Structure

Rule	Class/Attribute	Defined For
Super Classes	Top	DirXML-Subscriber
Containment	DirXML-Driver	DirXML-Subscriber
Named By	CN (Common Name)	DirXML-Subscriber

Mandatory Attributes

DirXML-Subscriber	Inherited from Top
CN	Object Class

Optional Attributes

DirXML-Subscriber

- DirXML-CreateRule
- DirXML-DriverFilter
- DirXML-EventTransformationRule
- DirXML-MatchingRule
- DirXML-PlacementRule
- DirXML-Timestamp

Inherited from Top

- ACL
- Audit:File Link
- Authority Revocation
- Back Link (Attribute)
- Bindery Property
- CA Private Key
- CA Public Key
- Certificate Revocation
- Certificate Validity Interval
- Cross Certificate Pair
- DirXML-Association
- Equivalent To Me
- GUID
- Last Referenced Time
- MASV:Authorized Range
- MASV:Default Range
- MASV:Proposed Label
- Obituary
- Other GUID
- Reference
- Revision
- Used By

Default ACL Template

Object Name	Default Rights	Affected Attributes	Class Defined For
[Creator]	Supervisor	[Entry Rights]	Top

Remarks

This object can contain rule and style sheet objects which further define the format and rules that the Identity Manager engine enforces when it sends data from the Identity Vault to the external application.

StyleSheet

Contains general XML styling information.

ASN.1 ID

- 2.16.840.1.113719.1.14.6.1.5

Class Flags

Class Flags	Setting
Container	Off
Effective	On
Nonremovable	On
Ambiguous Naming	Off
Ambiguous Container	Off
Auxiliary Class	Off

Class Structure

Rule	Class/Attribute	Defined For
Super Classes	Top	StyleSheet
Containment	Country	StyleSheet
	domain	StyleSheet
	Locality	StyleSheet
	Organization	StyleSheet
	Organizational Unit	StyleSheet
Named By	CN (Common Name)	StyleSheet

Mandatory Attributes

StyleSheet	Inherited from Top
CN	Object Class

Optional Attributes

StyleSheet

- XmlData
-

Inherited from Top

- ACL
- Audit:File Link
- Authority Revocation
- Back Link (Attribute)
- Bindery Property
- CA Private Key
- CA Public Key
- Certificate Revocation
- Certificate Validity Interval
- Cross Certificate Pair
- DirXML-Association
- Equivalent To Me
- GUID
- Last Referenced Time
- MASV:Authorized Range
- MASV:Default Range
- MASV:Proposed Label
- Obituary
- Other GUID
- Reference
- Revision
- Used By

Default ACL Template

Object Name	Default Rights	Affected Attributes	Class Defined For
[Creator]	Supervisor	[Entry Rights]	Top

B.2 DirXML Attribute Definitions

The following attributes have been defined for Identity Manager drivers and their associated objects.

DirXML-ApplicationSchema

Holds the XML file that describes the external application's schema.

Syntax

- Stream

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.29

Used In

- Identity Manager-Driver

Remarks

The schema-def element in the nds.dtd defines the format of this document. See <schema-def>.

DirXML-Associations

Holds the information that links an Identity Vault object with an object in an external application.

Syntax

- Path

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.4

Used In

- Top

Remarks

This is a multi-valued attribute, so objects can have associations with multiple Identity Manager drivers. However, an object cannot have multiple associations with a single driver.

The Path syntax contains three fields. Identity Manager uses the volume field to hold distinguished name of the Identity Manager driver. The name space (integer) field holds the state of the association. The path field (string) contains a identifier used by the external application that uniquely identifies an object in that application.

Since all object in the Identity Vault database inherit from Top, all objects have the potential for an association with an object in an external application.

DirXML-CreateRule

Contains the distinguished name of the rule object which contains the create rules.

Syntax

- Distinguished Name

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.22

Used In

- DirXML-Publisher
- DirXML-Subscriber

Remarks

Create rules specify the conditions under which a new object can be created in the Identity Vault or the external application.

The referenced rule object can be a DirXML-Rule object with XML data or a DirXML-StyleSheet object with XSLT data.

DirXML-DriverCacheLimit

Contains the maximum size, in kilobytes, of the driver's cache file.

Syntax

- Integer

Constraints

- DS_NONREMOVABLE_ATTR
- DS_PER_REPLICA
- DS_PUBLIC_READ
- DS_READ_ONLY_ATTR
- DS_SCHEDULE_SYNC_NEVER
- DS_SINGLE_VALUED_ATTR

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.24

Used In

- Identity Manager-Driver

Remarks

A value of zero allows an unlimited size.

DirXML-DriverFilter

Contains the filter specifying what data will pass to and from the Identity Vault and the external application.

Syntax

- Stream

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.11

Used In

- DirXML-Publisher
- DirXML-Subscriber

DirXML-DriverSetDN

References the container that holds the driver's definitions that are to be run on any given server.

Syntax

- Distinguished Name

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.12

DirXML-DriverStartOption

Indicates how the driver should be initialized.

Syntax

- Integer

Constraints

- DS_NONREMOVABLE_ATTR
- DS_PER_REPLICA
- DS_PUBLIC_READ
- DS_READ_ONLY_ATTR
- DS_SCHEDULE_SYNC_NEVER
- DS_SINGLE_VALUED_ATTR

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.13

Used In

- Identity Manager-Driver

Remarks

Currently, three values are defined for the drivers.

Value	Description
1	Auto. Start automatically when the Identity Vault is initialized.
2	Manual. Start manually through Designer or iManager.
3	Disabled. Cannot start until set to Manual or Auto.

DirXML-DriverStorage

Holds any required information in XML format that a driver may need between invocations.

Syntax

- Stream

Constraints

- DS_NONREMOVABLE_ATTR
- DS_PER_REPLICA
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.30

Used In

- Identity Manager-Driver

Remarks

The driver is responsible for storing and retrieving this information in an XML format that it understands.

DirXML-DriverTraceLevel

Contains the maximum level of Identity Manager trace message to output for drivers in the Driver Set.

Syntax

- Integer

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.25

Used In

- Identity Manager-DriverSet

Remarks

This attribute supports the following values:

Value	Description
0	No tracing
1	Displays informational messages about Identity Manager

DirXML-EventTransformationRule

Contains the distinguished name of the DirXML-StyleSheet object which contains the event transformation rules.

Syntax

- Distinguished Name

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.23

Used In

- DirXML-Publisher
- DirXML-Subscriber

Remarks

Event transformation rules convert events from one type to another. For example, they can be used to convert a <delete> element to a <remove-association> element.

This is an optional method for converting between systems. An Identity Manager driver is not required to supply such a rule.

DirXML-InputTransform

Contains the distinguished name of the DirXML-StyleSheet object which contains transformation rules for data sent by the Identity Manager driver to the Identity Manager engine.

Syntax

- Distinguished Name

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.17

Used In

- Identity Manager-Driver

Remarks

DirXML-StyleSheet object contains XSL commands for XSLT processing. These commands can be used for data format mapping such as changing a 15.2.1965 date format to a 2/15/65 format.

This is an optional method for converting between systems. An Identity Manager driver is not required to supply such a style sheet.

DirXML-JavaDebugPort

Contains whether the driver is using a debugging port on the Java Virtual Machine.

Syntax

- Integer

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.27

Used In

- Identity Manager-DriverSet

Remarks

This attribute supports the following values:

Value	Description
0	Don't use a debugging port
-1	Auto select a debugging port

DirXML-JavaModule

Holds the name of the Java class that must be loaded with the Identity Manager driver.

Syntax

- Case Ignore String

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.3

Used In

- Identity Manager-Driver

Remarks

The Java class must implement the com.novell.nds.dirxml.driver.SubscriptionShim interface.

DirXML-JavaTraceFile

Specifies where all Java System.out and System.err output can be logged.

Syntax

- Case Ignore String

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.28

Used In

- Identity Manager-DriverSet

Remarks

This attribute contains the name and path of a file that logs the output.

DirXML-MappingRule

Contains the distinguished name of the object which contains the mapping rules.

Syntax

- Distinguished Name

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.19

Used In

- Identity Manager-Driver

Remarks

Mapping rules map the Identity Vault class and attribute names to their corresponding names in the external application.

The referenced rule object can be a DirXML-Rule object with XML data or a DirXML-StyleSheet object with XSLT data.

DirXML-MatchingRule

Contains the distinguished name of a rule object which contains the matching rules.

Syntax

- Distinguished Name

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.21

Used In

- DirXML-Publisher
- DirXML-Subscriber

Remarks

Matching rules specify how the Identity Vault and the external application discover that an Identity Vault object corresponds to an external application entry. The rules specify which attribute values must match to create an association.

The referenced rule object can be a DirXML-Rule object with XML data or a DirXML-StyleSheet object with XSLT data.

DirXML-NativeModule

Holds the name of the DLL, NLM, or shared library that should be loaded with the driver.

Syntax

- Case Ignore String

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.9

Used In

- Identity Manager-Driver

Remarks

The subscriber channel of the driver must supply a number of well defined entry points for the Identity Manager engine.

DirXML-OutputTransform

Contains the distinguished name of the DirXML-StyleSheet object which contains transformation rules for data send from the Identity Manager engine to the Identity Manager driver.

Syntax

- Distinguished Name

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.18

Used In

- Identity Manager-Driver

Remarks

DirXML-StyleSheet object contains XSL commands for XSLT processing. These commands can be used for data format mapping such as changing a 15.2.1965 date format to a 2/15/65 format.

This is an optional method for converting between systems. An Identity Manager driver is not required to supply such a style sheet.

DirXML-PlacementRule

Contains the distinguished name of an object which contains the placement rules.

Syntax

- Distinguished Name

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.20

Used In

- DirXML-Publisher
- DirXML-Subscriber

Remarks

Placement rules define where newly created objects are located. For the publisher, the rules define in which the Identity Vault containers objects can be created. For the subscriber, the rules define where newly created objects are located in the external application.

The referenced rule object can be a DirXML-Rule object with XML data or a DirXML-StyleSheet object with XSLT data.

DirXML-ShimAuthID

Holds the identity that the Identity Manager driver passes to the external application for authentication.

Syntax

- Case Ignore String

Constraints

- DS_NONREMOVABLE_ATTR
- DS_PER_REPLICA
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.5

Used In

- Identity Manager-Driver

Remarks

The identity can be a name or a unique identifying number, in the format required by the external application to identify an administrator of the application.

DirXML-ShimAuthPassword

Holds the password that the Identity Manager driver passes to the external application for authentication.

Syntax

- Octet String

Constraints

- DS_NONREMOVABLE_ATTR
- DS_PER_REPLICA
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.6

Used In

- Identity Manager-Driver

DirXML-ShimAuthServer

Holds information about the server hosting the external application which the Identity Manager driver requires for authentication.

Syntax

- Case Ignore String

Constraints

- DS_NONREMOVABLE_ATTR

- DS_PER_REPLICA
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.7

Used In

- Identity Manager-Driver

Remarks

This is an optional attribute. If all you need to authenticate to the external application is a name and a password, then this attribute does not need a value. If the external application requires a server name or server address, this information should be store in this attribute.

DirXML-ShimConfigInfo

Holds configuration information for the Identity Manager driver.

Syntax

- Stream

Constraints

- DS_NONREMOVABLE_ATTR
- DS_PER_REPLICA
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.8

Used In

- Identity Manager-Driver

Remarks

All configuration information is optional. This attribute allows drivers to have configuration options for the driver, the publisher, and the subscriber. The information is read when the Identity Manager engine initializes the driver.

DirXML-ServerList

Contains the distinguished names of the servers that are using the driver set.

Syntax

- Distinguished Name

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.15

Used In

- Identity Manager-DriverSet

DirXML-State

Contains the current status of the driver or the modification state of the publisher or subscriber filter.

Syntax

- Integer

Constraints

- DS_NONREMOVABLE_ATTR
- DS_PER_REPLICA
- DS_PUBLIC_READ

- DS_READ_ONLY_ATTR
- DS_SCHEDULE_SYNC_NEVER
- DS_SINGLE_VALUED_ATTR

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.14

Used In

- Identity Manager-Driver
- DirXML-Publisher
- DirXML-Subscriber

Remarks

DirXML-Driver objects support the following states for the driver.

State	Description
0	Stopped
1	Starting
2	Running
3	Shutdown pending
11	Driver get schema

DirXML-Subscriber and DirXML-Publisher objects support the following states for the filter.

State	Description
0	Current
1	Modified

DirXML-Timestamp

Contains a timestamp that allows a driver to know when the external application was last synchronized with the Identity Vault.

Syntax

- Timestamp

Constraints

- DS_NONREMOVABLE_ATTR
- DS_PUBLIC_READ
- DS_PER_REPLICA
- DS_READ_ONLY_ATTR
- DS_SCHEDULE_SYNC_NEVER
- DS_SINGLE_VALUED_ATTR

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.16

Used In

- DirXML-Subscriber

DirXML-XSLTraceLevel

Contains the maximum level of XSL trace messages to output for the XSL processor.

Syntax

- Integer

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.26

Used In

- Identity Manager-DriverSet

Remarks

This attribute supports the following values:

Value	Description
0	No tracing
1	Displays source node processing
2	Displays the above messages and rule instantiation messages
3	Displays the above messages and temple instantiation messages
4	Displays the above messages and rule matching and select expression messages

XmlData

Holds XML encoded data.

Syntax

- Stream

Constraints

- DS_NONREMOVABLE_ATTR
- DS_SINGLE_VALUED_ATTR
- DS_SYNC_IMMEDIATE

ASN.1 ID

- 2.16.840.1.113719.1.14.4.1.3

Used In

- DirXML-Rule
- StyleSheet

Remarks

C.0 Revision History

The following table lists changes made to the Identity Manager Driver Kit documentation:

June 10, 2015	Updated the documentation for 4.5 release.
August 1, 2012	Made minor technical edits.
March 1, 2006	Made minor technical edits.
October 5, 2005	Transitioned to revised Novell documentation standards. Updated the description of Remote Communication in Section 1.5, Designing the Driver.
April 2005	Added information on the Identity Manager driver for exteNd composer.
February 2004	Updated NDS.DTD to the version 2 release, corresponding to the release of Identity Manager 2. Added DirXML script DTD and reference. Updated XDS libraries with support for new elements in NDS.DTD.
June 2003	Added additional information on the XDS Libraries and updated introductory material.
March 2003	Added documentation on the XDS Libraries.
March 2002	Updated the documentation to Identity Manager 1.1. Added the following: • Command transformation rule • Rule chaining • Expanded use of the query parameters to the schema mapping rules, input transformation rules, and output transformation rules • Command style sheet parameters • Automatic handling of auxiliary classes • Enhanced move operations • Attribute password modify • Dynamic loading of jar files
June 2001	Added information on using the Identity Manager Driver Kit for the Solaris and Linux platforms.
February 2001	Revised the rule chapters and added a style sheet chapter. Added Identity Manager

	error codes.
September 2000	Moved to the NDK as an Early Access component.
May 2000	Published as a Leading Edge component

D.0 Legal Notices

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 make changes to any and all parts of Novell software, at any time, without any obligation to notify any person or entity of such changes.

Any products or technical information provided under this Agreement may be subject to U.S. export controls and the trade laws of other countries. You agree to comply with all export control regulations and to obtain any required licenses or classification to export, re-export or import deliverables. You agree not to export or re-export to entities on the current U.S. export exclusion lists or to any embargoed or terrorist countries as specified in the U.S. export laws. You agree to not use deliverables for prohibited nuclear, missile, or chemical biological weaponry end uses. See the Novell International Trade Services Web page for more information on exporting Novell software. Novell assumes no responsibility for your failure to obtain any necessary export approvals.

Copyright © 2008-2012 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.

Novell, Inc.
1800 South Novell Place
Provo, UT 84606
U.S.A.
www.novell.com

Online Documentation: To access the latest online documentation for this and other Novell products, see the Novell Documentation Web page.

Novell Trademarks

For Novell trademarks, see the Novell Trademark and Service Mark list.

Third-Party Materials

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