NetIQ® Identity Manager

Understanding Policies

February 2018

Legal Notice

Contents

About this Book and the Library

NetIQ Identity Manager is a data sharing and synchronization service that enables applications, directories, and databases to share information. It links scattered information and enables you to establish policies that govern automatic updates to designated systems when identity changes occur.

Identity Manager provides the foundation for account provisioning, security, single sign-on, user self-service, authentication, authorization, automated workflows, and Web services. It allows you to integrate, manage, and control your distributed identity information so you can securely deliver the right resources to the right people.

Intended Audience

This guide is intended for Identity Manager administrators.

Other Information in the Library

For more information about the library for Identity Manager, see the Identity Manager documentation website.

About NetIQ Corporation

We are a global, enterprise software company, with a focus on the three persistent challenges in your environment: Change, complexity and risk—and how we can help you control them.

Our Viewpoint

Adapting to change and managing complexity and risk are nothing new

In fact, of all the challenges you face, these are perhaps the most prominent variables that deny you the control you need to securely measure, monitor, and manage your physical, virtual, and cloud computing environments.

Enabling critical business services, better and faster

We believe that providing as much control as possible to IT organizations is the only way to enable timelier and cost effective delivery of services. Persistent pressures like change and complexity will only continue to increase as organizations continue to change and the technologies needed to manage them become inherently more complex.

Our Philosophy

Selling intelligent solutions, not just software

In order to provide reliable control, we first make sure we understand the real-world scenarios in which IT organizations like yours operate — day in and day out. That's the only way we can develop practical, intelligent IT solutions that successfully yield proven, measurable results. And that's so much more rewarding than simply selling software.

Driving your success is our passion

We place your success at the heart of how we do business. From product inception to deployment, we understand that you need IT solutions that work well and integrate seamlessly with your existing investments; you need ongoing support and training post-deployment; and you need someone that is truly easy to work with — for a change. Ultimately, when you succeed, we all succeed.

Our Solutions

- Identity & Access Governance
- Access Management
- Security Management
- Systems & Application Management
- Workload Management
- Service Management

Contacting Sales Support

For questions about products, pricing, and capabilities, contact your local partner. If you cannot contact your partner, contact our Sales Support team.

Worldwide:	www.netiq.com/about_netiq/officelocations.asp
United States and Canada:	1-888-323-6768
Email:	info@netiq.com
Web Site:	www.netiq.com

Contacting Technical Support

For specific product issues, contact our Technical Support team.

Worldwide:	www.netiq.com/support/contactinfo.asp
North and South America:	1-713-418-5555
Europe, Middle East, and Africa:	+353 (0) 91-782 677
Email:	support@netiq.com
Web Site:	www.netiq.com/support

Contacting Documentation Support

Our goal is to provide documentation that meets your needs. If you have suggestions for improvements, click **Add Comment** at the bottom of any page in the HTML versions of the documentation posted at www.netiq.com/documentation. You can also email Documentation-Feedback@netiq.com. We value your input and look forward to hearing from you.

Contacting the Online User Community

Qmunity, the NetIQ online community, is a collaborative network connecting you to your peers and NetIQ experts. By providing more immediate information, useful links to helpful resources, and access to NetIQ experts, Qmunity helps ensure you are mastering the knowledge you need to realize the full potential of IT investments upon which you rely. For more information, visit http://community.netiq.com.

1 Overview

Policies are what Identity Manager uses to synchronize data to the different systems. They are the foundation of Identity Manager. Understanding policies and how they work is important to successfully working with Identity Manager.

- "What Are Policies?" on page 9

For administration information about policies, see

- *NetIQ Identity Manager - Using Designer to Create Policies*
- *NetIQ Identity Manager Policies in iManager Guide*
- *NetIQ Identity Manager Credential Provisioning Guide*
- *Identity Manager DTD Reference Documentation*

What Are Policies?

At a high level, a policy is the set of rules that enables you to manage the way Identity Manager sends and receives updates. The driver sends changes from the connected system to the Identity Vault, where policies are used to manipulate the data to achieve the desired results.

As part of understanding how policies work, it is important to understand the components of policies.

- Policies are made up of rules.
- A rule is a set of conditions, see "Conditions" that must be met before a defined action, see "Actions" occurs.
- Actions can have dynamic arguments that derive from tokens that are expanded at run time.
- Tokens are broken up into two classifications: nouns and verbs.
  - Noun tokens, see "Noun Tokens" expand to values that are derived from the current operation, the source or destination data stores, or some external source.
  - Verb tokens, see "Verb Tokens" modify the concatenated results of other tokens that are subordinate to them.
- Regular expressions (see "Regular Expressions" on page 49) and XPath 1.0 expressions (see "XPath 1.0 Expressions" on page 50) are commonly used in the rules to create the desired results for the policies.
- A policy operates on an XDS document and its primary purpose is to examine and modify that document.
- An operation is any element in the XDS document that is a child of the input element and the output element. The elements are part of the NetIQ `nds.dtd`; for more information, see Identity Manager DTD Reference Documentation (http://www.netiq.com/documentation/identity-manager-developer/dtd-documentation.html).
- An operation usually represents an event, a command, or a status.

- The policy is applied separately to each operation. As the policy is applied to each operation in turn, that operation becomes the current operation. Each rule is applied sequentially to the current operation. All of the rules are applied to the current operation unless an action is executed by a prior rule that causes subsequent rules to no longer be applied.
- A policy can also get additional context from outside of the document and cause side effects that are not reflected in the result document.

For detailed information, see the following sections in this guide:

- Chapter 2, “Upgrading Identity Manager Policies,” on page 11
- Chapter 3, “Understanding Types of Policies,” on page 17
- Chapter 4, “Understanding Policy Components,” on page 45
- Chapter 6, “Defining Policies by Using XSLT Style Sheets,” on page 57

2 Upgrading Identity Manager Policies

If you have a prior version of Identity Manager installed, continue with this section. If you have installed Identity Manager for the first time, skip to Chapter 3, “Understanding Types of Policies,” on page 17.

Methods for Upgrading the Driver Configuration File

There are multiple ways of upgrading an existing driver and its policies. There is no simple method, because there is no merge process in Identity Manager to merge customized policies. When a driver is upgraded, any policy that has the same name as a policy in the new driver is over written. If the policies have been customized, they are overwritten and the customization is lost.

There are many different ways of upgrading to address this issue, but this section discusses two of the upgrade methods. There are pros and cons to each upgrade method.

Installing a New Driver and Moving the Existing Policies from the Old Driver

The pros to this method are:

- Any existing policies are not overwritten.

The cons to this method are:

- All associations for synchronized objects are lost and must be re-created, expanded, and reloaded.
- The amount of time it takes to make the associations again. If you have a policy that depends upon a specific association, that policy does not work.
- Complexity of making sure policies and rules are restored correctly.

Overlay the New Driver Configuration File Over an Existing Driver

The impact of this method depends upon how your policies are configured.

The pros are:

- If your policies have different names than the policies in the driver configuration file, they are not overwritten.
- The associations for the synchronized objects stay the same and do not need to be re-created.

The cons are:

- If your policies have the same name as policies in the driver configuration file, they are overwritten.

This is the recommended upgrade option. However, in order for this upgrade method to work, some methodology needs to be in place for creating policies.

- You should follow the same procedures when developing policies as when you upgrade the policies.
- Existing NetIQ policies or rules should never be modified.
- If you do not use a default policy, disable the policy, but do not delete it.
- Create new policies or rules to achieve the desired result for your business needs.
- Use a standard naming model for naming the policies in your company.
- Name your policies with a prefix of the policy set where the policy is stored. This allows you to know which policy set to attach the policy to.

If you have these methodologies in place, use “Recommended Driver Configuration Upgrade Procedure” on page 12, to upgrade the driver configuration.

Recommended Driver Configuration Upgrade Procedure

This is NetIQ’s recommended driver configuration upgrade procedure. Make sure you do the procedure in a lab environment. The procedure can be performed in Designer or iManager.

Upgrading the Driver Configuration in Designer

The upgrade procedure has three different tasks that need to be completed:

Creating an Export of the Driver

Creating an export of the driver makes a backup of your current configuration. Make sure you have a backup before upgrading.

1. Verify that your project in Designer has the most current version of your driver. For instructions, see the *NetIQ Designer for Identity Manager Administration Guide*.
2. In the Modeler, right-click the driver line of the driver you are upgrading.
3. Select **Export to a Configuration File**.
4. Browse to a location to save the configuration file, then click **Save**.
5. Click **OK** on the results page.

Overlay the New Driver Configuration File Over the Existing Driver

1. In the Modeler, right-click the driver line of the driver you are upgrading.
2. Select **Run Configuration Wizard**.
3. Click **Yes** on the warning page.

   The warning is informing you that all of the driver setting and policies are reset.

   IMPORTANT: Make sure that your customized policies have different names, from the default policies, so you do not lose any data.

   ---
4. Browse to and select the driver configuration for the driver are upgrading, then click **Run**.
5. Specify the information for the driver, then click **Next**.

   There might be more than one page of information to specify.
6. Click **OK** on the results page.

Restoring Custom Policies and Rules to the Driver

You can add policies into the policy set in two different ways:

- “Adding a Customized Policy Through the Outline View” on page 13
- “Adding a Customized Policy Through the Show Policy Flow View” on page 13

Adding a Customized Policy Through the Outline View

1. In the **Outline** view, select the upgraded driver to display the **Policy Set** view.
2. Right-click the policy set where you need to restore the customized policy to the driver, then select **New > From Copy**.
3. Browse to and select the customized policy, then click **OK**.
4. Specify the name of the customized policy, then click **OK**.
5. Click **Yes** in the file conflict message to save your project.
6. After the Policy Builder opens the policy, verify that the information is correct in the copied policy.
7. Repeat Step 2 through Step 6 for each customized policy you need to restore to the driver.
8. Start the driver and test the driver.
9. After you verify that the policies work, move the driver to the production environment.

Adding a Customized Policy Through the Show Policy Flow View

1. In the **Outline** view, select the upgraded driver, then click the **Show Policy Flow** icon.
2. Right-click the policy set where you need to restore the customized policy to the driver, then select **Add Policy > Copy Existing**.
3. Browse to and select the customized policy, then click **OK**.
4. Specify the name of the customized policy, then click **OK**.
5. Click **Yes** in the file conflict message to save your project.
6. After the Policy Builder opens the policy, verify that the information is correct in the copied policy.
7. Repeat Step 2 through Step 6 for each customized policy you need to restore to the driver.

8 Start the driver and test the driver.

9 After you verify that the policies work, move the driver to the production environment.

Upgrading the Driver Configuration in iManager

The upgrade procedure has three different tasks that need to be completed:

- “Creating an Export of the Driver” on page 14
- “Overlaying the New Driver Configuration File Over the Existing Driver” on page 14
- “Restoring Custom Policies and Rules Back to the Driver” on page 14

Creating an Export of the Driver

Creating an export of the driver makes a backup of your current configuration. Make sure you have a backup before upgrading.

1 In iManager, select **Identity Manager > Identity Manager Overview**.

2 Click **Search** to find the Driver Set object that holds the driver you want to upgrade.

3 Click the driver you want to upgrade, then click **Export**.

4 Click **Next**, then select **Export all contained policies, linked to the configuration or not**.

5 Click **Next**, then click **Save As**.

6 Select **Save to Disk**, then click **OK**.

7 Click **Finish**.

Overlaying the New Driver Configuration File Over the Existing Driver

1 In iManager, select **Identity Manager > Identity Manager Overview**.

2 Click **Add Driver**, then click **Next** on the New Driver Wizard page.

3 Select the driver configuration you want to overlay, then click **Next**.

4 In the **Existing drivers** field, browse to and select the driver you want to upgrade.

5 Specify the information for the driver, the click **Next**.

6 On the summary page, select **Update everything about that driver and policy libraries**.

IMPORTANT: Make sure that any customized policies have a different name from the default, so you do not lose any data.

---

7 Click **Next**, then click **Finish** on the Summary page.

Restoring Custom Policies and Rules Back to the Driver

1 In iManager, select **Identity Manager > Identity Manager Overview**.

2 Click **Search** to find the Driver Set object, then click the upgraded driver.

3 Select the policy set where you need to restore the customized policy to the driver, then click **Insert**.

4 Select **Use an existing policy**, then browse to and select the custom policy.

5 Click **OK**, then click **Close**.

6 Repeat Step 3 through Step 5 for each custom policy you need to restore to the driver.

7 Start the driver and test the driver.

8 After you verify that the policies work, move the driver to the production environment.

3 Understanding Types of Policies

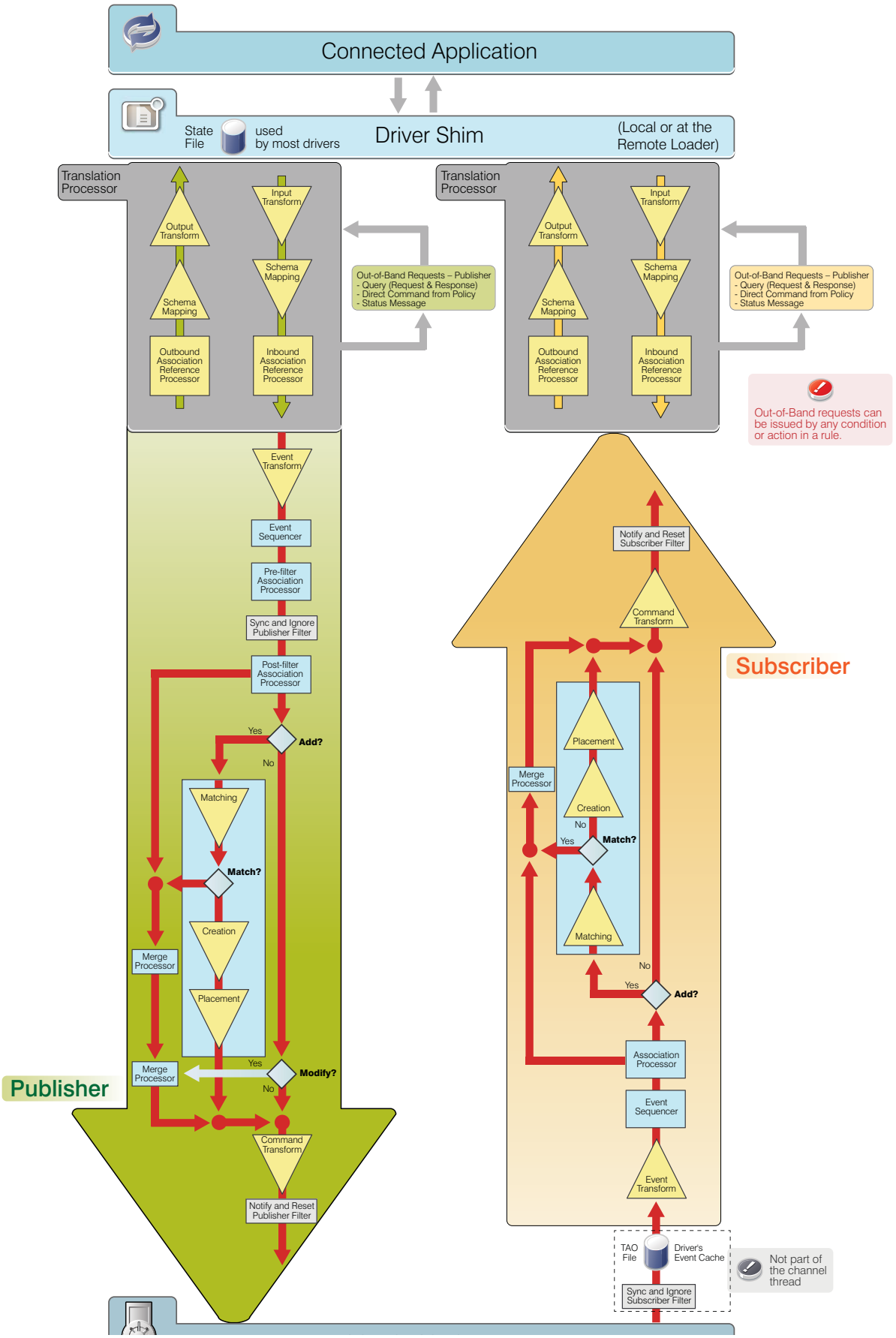
This section contains an overview of policies and filters, and their function in an Identity Manager environment. The following topics are covered:

Identity Manager Architecture in Relation to Policies

Identity Manager provides for the clean movement of data between the Identity Vault and any application, directory, or database. To accomplish this, Identity Manager has a well-defined interface that translates the Identity Vault data and events into XML format. This interface allows the data to flow in and out of the directory.

The following figure illustrates the basic Identity Manager components and their relationships.

Figure 3-1 *Identity Manager Components*


Connected Application
State File
used by most drivers
Driver Shim
(Local or at the Remote Loader)
Translation Processor
Output Transform
Schema Mapping
Outbound Association Reference Processor
Input Transform
Schema Mapping
Inbound Association Reference Processor
Out-of-Band Requests – Publisher
- Query (Request & Response)
- Direct Command from Policy
- Status Message
Translation Processor
Output Transform
Schema Mapping
Outbound Association Reference Processor
Input Transform
Schema Mapping
Inbound Association Reference Processor
Out-of-Band Requests – Publisher
- Query (Request & Response)
- Direct Command from Policy
- Status Message
Out-of-Band requests can be issued by any condition or action in a rule.
Event Transform
Event Sequencer
Pre-filter Association Processor
Sync and Ignore Publisher Filter
Post-filter Association Processor
Yes
Add?
No
Matching
Match?
Creation
Placement
Merge Processor
Merge Processor
Yes
Modify?
No
Command Transform
Notify and Reset Publisher Filter
Publisher
Notify and Reset Subscriber Filter
Command Transform
Subscriber
Placement
Creation
No
Yes
Match?
Matching
Merge Processor
No
Yes
Add?
Association Processor
Event Sequencer
Event Transform
TAO File
Driver's Event Cache
Sync and Ignore Subscriber Filter
Not part of the channel thread

The Identity Manager engine is the key module in the Identity Manager architecture. It provides the interface that allows Identity Manager drivers to synchronize information with the Identity Vault, allowing even disparate data systems to connect and share data.

The Identity Manager engine exposes the Identity Vault data and the Identity Vault events by using an XML format. The Identity Manager engine employs a rules processor and a data transformation engine to manipulate the data as it flows between two systems. Access to the rules processor and transformation engine is provided through control points called Policy Sets. Policy Sets can contain zero or more policies.

A policy implements business rules and processes primarily by transforming an event on a channel input into a set of commands on the channel output. The way each driver synchronizes data and events is configured by the administrator through a series of policies. For example, if a Creation Policy specifies that a User object must have a value for the Given Name attribute, any attempt to create a User object without a given name value is rejected.

Using Filters

Filters specify the object classes and the attributes for which the Identity Manager engine processes events and how changes to those classes and attributes are handled.

Filters only pass events occurring on objects whose base class matches one of those classes specified by the filter. 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. There is a separate filter setting for each channel, which allows the control of the synchronization direction and the authoritative data source for each class and attribute.

NOTE: In the Identity Vault, 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 or the auxiliary classes from which additional attributes might come.

For example, if the User class with the Surname and Given Name attributes is set to synchronize in the filter, 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 filter.

Filters must be configured to include the following:

- Attributes that are to be synchronized
- Attributes that are not synchronized, but are used to trigger policies to do something

See "Controlling the Flow of Objects with the Filter" in *NetIQ Identity Manager - Using Designer to Create Policies* for information on defining filters.

How Policies Function

At a high level, a policy is a set of rules that enables you to customize the way Identity Manager sends and receives updates. The driver sends changes from the connected system to the Identity Vault, where policies are used to manipulate the data to achieve the desired results.

- "Detecting Changes and Sending Them to the Identity Vault" on page 21
- "Filtering Information" on page 21
- "Using Policies to Apply Changes" on page 21

Detecting Changes and Sending Them to the Identity Vault

When a driver is written, an attempt is made to include the ability to synchronize anything a company deploying the driver might use. The developer writes the driver to detect any relevant changes in the connected system, then pass these changes to the Identity Vault.

Changes are contained in an XML document, formatted according to the Identity Manager specification. The following snippet contains one of these XML documents:

```
<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>
```

Filtering Information

Drivers are designed to report any relevant changes, then enable you to filter the information, so only the information you desire enters your environment.

For example, if a company doesn't need information about groups, it can implement a filter to block all operations regarding groups in either the Identity Vault or the connected system. If the company does need information about users and groups, it can implement a filter to allow both types of objects to synchronize between the Identity Vault and the connected system.

Defining filters to allow the synchronization of only objects that are interesting to you is the first step in driver customization.

The next step defines what Identity Manager does with the objects that are allowed by the filter. As an example, refer to the add operation in the XML document above. A user named Smith with a telephone number of 111-1111 was added to the connected system. Assuming you allow this operation, Identity Manager needs to decide what to do with this user.

Using Policies to Apply Changes

To make changes, Identity Manager applies a set of policies, in a specific order.

First, a Matching policy answers the question, "Is this object already in the data store?" To answer this, you need to define the characteristics that are unique to an object. A common attribute to check might be an e-mail address, because these are usually unique. You can define a policy that says "If two objects have the same e-mail address, they are the same object."

If a match is found, Identity Manager notes this in an attribute called an association. An association is a unique value that enables Identity Manager to associate objects in connected systems.

In circumstances where a match is not found, a Creation policy is called. The Creation policy tells Identity Manager under what conditions you want objects to be created. You can make the existence certain attributes mandatory in the creation rule. If these attributes do not exist, Identity Manager blocks the creation of the object until the required information is provided.

After the object is created, a Placement policy tells Identity Manager where to put it. You can specify that objects should be created in a hierarchical structure identical to the system they came from, or you can place them somewhere completely different, based on an attribute value.

If you want to place users in a hierarchy according to a location attribute on the object, and name them according to the Full Name, you can require these attributes in the Creation policy. This ensures that the attribute exists so your placement strategy works correctly.

There are many other things you can do with policies. Using the Policy Builder, you can easily generate unique values, add and remove attributes, generate events and commands, send e-mail, and more. Even more advanced transformations are available by using XSLT to directly transform the XML document that carries information between applications.

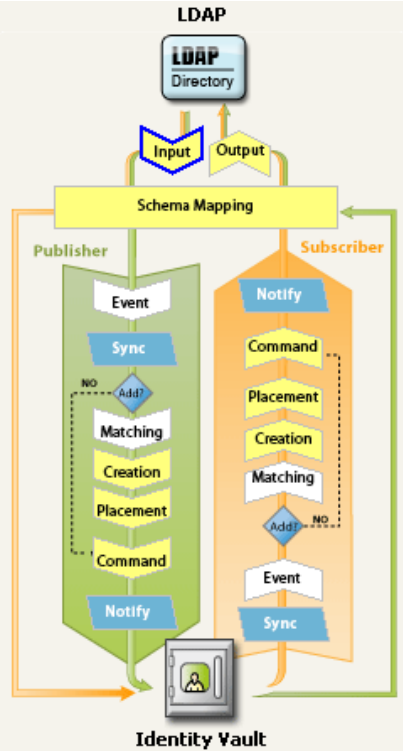
Continue to “Policy Types” on page 22 to learn more about the different types of policies, then move on to *NetIQ Identity Manager - Using Designer to Create Policies* to learn using the Policy Builder.

Policy Types

There are several different types of policies you can define on both the Subscriber and Publisher channels. Each policy is applied at a different step in the data transformation, and some policies are only applied when a certain action occurs. For example, a Creation policy is applied only when a new object is created.

The different policy types and their order of execution on the channel are:

Figure 3-2 *Order of Execution of the Policies*



Event Transformation Policy

Event Transformation policies alter the Identity Manager engine's view of the events that happen in the Identity Vault or the connected application. The most common tasks performed in an Event Transformation policy are changing an event and custom filtering, such as scope filtering and event-type filtering.

Changing an event includes performing an action to change an object, either in the source or destination directories, but not adding anything to the existing operation event.

For example, a policy could remove the association in the publisher event transform when a delete is detected and then remove the delete event, or detect events in the subscriber event transform and write back modifications to the Identity Vault. This type of policy should not be used to add destination attributes to the operation, as this policy is not used in merge operations.

Scope filtering removes unwanted events based on event location or an attribute value. For example, a policy could remove the event if the object is not in a specific container, or if the department attribute is not equal to a specific value, or if the user is not a member of a specific group.

Event-type filtering removes unwanted events based on event type. For example, a policy could remove all delete events.

Examples:

- “Scope Filtering: Example 1” on page 24
- “Scope Filtering: Example 2” on page 24
- “Type Filtering: Example 1” on page 25
- “Type Filtering: Example 2” on page 25

Scope Filtering: Example 1

This example DirXML Script policy allows events only for users who are contained within the Users subtree, are not disabled, and do not contain the word Consultant or Manager in the Title attribute. It also generates a status document indicating when an operation has been blocked. To view the policy in XML, see Event_Transformation_Scope1.xml (../samples/Event_Transformation_Scope1.xml).

```
<policy>
       <rule>
               <description>Scope Filtering</description>
               <conditions>
                           <or>
                                <if-class-name op="equal">User</if-class-name>
                           </or>
                     <or>
                      <if-src-dn op="not-in-subtree">Users</if-src-dn>
                      <if-attr name="Login Disabled" op="equal">True</if-attr>
                      <if-attr mode="regex" name="Title" op="equal">.*Consultant.*
                      </if-attr>
                      <if-attr mode="regex" name="Title" op="equal">.*Manager.*
                      </if-attr>
                       </or>
             </conditions>
             <actions>
                       <do-status level="error">
                          <arg-string>
                                <token-text>User doesn’t meet required conditions</
token-text>
                          </arg-string>
                       </do-status>
                  <do-veto/>
            </actions>
      </rule>
</policy>
```

Scope Filtering: Example 2

This DirXML Script policy vetoes all operations on User objects except for modifies of already associated objects. To view the policy in XML, see Event_Transformation_Scope2.xml. (../samples/Event_Transformation_Scope2.xml).

```
<policy>
        <rule>
                <description>Veto all operation on User except modifies of already
associated objects</description>
               <conditions>
                       <or>
                          <if-class-name op="equal">User</if-class-name>
                       </or>
                       <or>
                          <if-operation op="not-equal">modify</if-operation>
                          <if-association op="not-associated"/>
                       </or>
               </conditions>
               <actions>
                           <do-veto/>
               </actions>
        </rule>
</policy>
```

Type Filtering: Example 1

The first rule of this example DirXML Script policy allows only objects in the Employee and Contractor containers to be synchronized. The second rule blocks all Rename and Move operations. To view the policy in XML, see Event_Transformation_Type1.xml (../samples/Event_Transformation_Type1.xml).

```
<policy>
      <rule>
         <description>Only synchronize the Employee and Contractor subtrees
         </description>
           <conditions>
             <and>
                  <if-src-dn op="not-in-container">Employees
                  </if-src-dn>
                  <if-src-dn op="not-in-container">Contractors
                  </if-src-dn></and></conditions>
               <actions>
                 <do-status level="warning">
                    <arg-string>
                    <token-text xml:space="preserve">Change ignored: Out of scope.
                    </token-text>
                    </arg-string>
                 </do-status><do-veto/>
                    </actions>
                    </rule>
                    <rule>
                    <description>Don't synchronize moves or renames
                    </description>
                  <conditions>
                      <or>
                         <if-operation op="equal">move
                         </if-operation>
                         <if-operation op="equal">rename
                         </if-operation>
                       </or>
                 </conditions>
                    <actions>
                    <do-status level="warning">
                    <arg-string>
                     <token-text xml:space="preserve">Change ignored: We don't
like you to do that.
                     </token-text></arg-string>
                     </do-status>
                     <do-veto/>
                     </actions>
                     </rule>
                     </policy>
```

Type Filtering: Example 2

This DirXML Script policy blocks all Add events. To view the policy in XML, see Event_Transformation_Type2.xml (../samples/Event_Transformation_Type2.xml).

```
<policy>
        <rule>
                <description>Type Filtering</description>
                <conditions>
                       <and>
                         <if-operation op="equal">add</if-operation>
                           </and>
                </conditions>
                <actions>
                    <do-status level="warning">
                     <arg-string>
                    <token-text>Change ignored: Adds are not allowed.</token-text>
                      </arg-string>
                        </do-status>
                            <do-veto/>
                 </actions>
          </rule>
</policy>
```

Matching Policies

Matching policies, such as Subscriber Matching and Publisher Matching, look for an object in the destination data store that corresponds to an unassociated object in the source datastore. It is important to note that Matching policies are not always needed or desired.

For example, a Matching policy might not be desired when performing an initial migration if there are no preexisting or corresponding objects.

A Matching policy must be carefully crafted to ensure that the Matching policy doesn't find false matches.

- "Match by Internet E-Mail Address: Example" on page 26
- "Match by Name: Example" on page 27

Match by Internet E-Mail Address: Example

This example DirXML Script policy matches users based on the Internet E-mail Address. To view the policy in XML, see Matching1.xml (../samples/Matching1.xml).

```
<policy>
          <rule>
                <description>Match Users based on email address</description>
                <conditions>
                           <and>
                               <if-class-name op="equal">User</if-class-name>
                           </and>
                </conditions>
                <actions>
                  <do-find-matching-object>
                    <arg-dn>
                     <token-text>ou=people,o=novell</token-text>
                      </arg-dn>
                      <arg-match-attr name="Internet EMail Address"/>
                       </do-find-matching-object>
                </actions>
         </rule>
</policy>
```

Match by Name: Example

This example DirXML Script policy matches a Group object based on its Common Name attribute. To view the policy in XML, see Matching2.xml (../samples/Matching2.xml).

```
<?xml version="1.0" encoding="UTF-8"?>
<policy>
          <rule>
                    <description>Match Group by Common Name</description>
                    <conditions>
                           <or>
                              <if-class-name op="equal">Group</if-class-name>
                           </or>
                    </conditions>
                    <actions>
                       <do-find-matching-object scope="subtree">
                         <arg-match-attr name="CN"/>
                          </do-find-matching-object>
                    </actions>
           </rule>
</policy>
```

Creation Policy

Creation policies, such as the Subscriber Creation policy and the Publisher Creation policy, define the conditions that must be met to create a new object.

For example, you create a new user in the Identity Vault, but you give the new User object only a name and ID. This creation is mirrored in the Identity Vault tree, but the addition is not immediately reflected in applications connected to the Identity Vault because you have a Creation policy specifying that only User objects with a more complete definition are allowed.

A Creation policy can be the same for both the Subscriber and the Publisher, or it can be different.

Template objects can be specified for use in the creation process when the object is to be created in the Identity Vault.

Creation policies are commonly used to:

- Veto creation of objects that don't qualify, possibly because of a missing attribute.
- Provide default attribute values.
- Provide a default password.

Examples:

- "Required Attributes: Example" on page 27
- "Default Attribute Values: Example" on page 28
- "Default Password: Example" on page 29
- "Specify Template: Example" on page 29

Required Attributes: Example

The first rule of this example DirXML Script policy requires that a User object contain a CN, Given Name, Surname, and Internet EMail Address attribute before the user can be created. The second rule requires an OU attribute for all Organizational Unit objects. The final rule vetoes all User objects with a name of Fred. To view the policy in XML, see Create1.xml (../samples/Create1.xml).

```
<policy>
      <rule>
              <description>Veto if required attributes CN, Given Name, Surname and
Internet EMail Address not available</description>
              <conditions>
                      <or>
                                  <if-class-name op="equal">User</if-class-name>
                      </or>
              </conditions>
              <actions>
                 <do-veto-if-op-attr-not-available name="CN"/>
                 <do-veto-if-op-attr-not-available name="Given Name"/>
                   <do-veto-if-op-attr-not-available name="Surname"/>
                     <do-veto-if-op-attr-not-available name="Internet EMail
Address"/>
              </actions>
      </rule>
    <rule>
              <description>Organizational Unit Required Attributes</description>
             <conditions>
                <or>
                                <if-class-name op="equal">Organizational Unit</if-
class-name>
                      </or>
             </conditions>
             <actions>
                   <do-veto-if-op-attr-not-available name="OU"/>
             </actions>
      </rule>
      <rule>
    <description>Conditionally veto guys named "Fred"</description>
    <conditions>
      <and>
        <if-global-variable name="no-fred" op="equal">true</if-global-variable>
        <if-op-attr name="Given Name" op="equal">Fred</if-op-attr>
      </and>
    </conditions>
    <actions>
      <do-status level="warning">
        <arg-string>
          <token-text xml:space="preserve" xmlns:xml="http://www.w3.org/XML/1998/
namespace">Vetoed "Fred"</token-text>
        </arg-string>
      </do-status>
        <do-veto/>
            </actions>
       </rule>
</policy>
```

Default Attribute Values: Example

This example DirXML Script policy adds a default value for a user's Description attribute. To view the policy in XML, see Create2.xml (../samples/Create2.xml).

```
<policy>
        <rule>
                <description>Default Description of New Employee</description>
            <conditions>
                        <or>
                            <if-class-name op="equal">User</if-class-name>
                        </or>
                </conditions>
                <actions>
                        <do-set-default-attr-value name="Description">
                                <arg-value type="string">
                                        <token-text>New Employee</token-text>
                                </arg-value>
                        </do-set-default-attr-value>
                </actions>
         </rule>
</policy>
```

Default Password: Example

This example DirXML Script policy provides creates a password value comprised of the first two characters of the first name and the first six characters of the last name, all in lowercase. To view the policy in XML, see Create3.xml (../samples/Create3.xml)

```
<policy>
         <rule>
                <description>Default Password of [2]FN+[6]LN</description>
            <conditions>
                        <and>
                                <if-class-name op="equal">User</if-class-name>
                                <if-password op="not-available"/>
                        </and>
                </conditions>
                <actions>
                        <do-set-dest-password>
                                <arg-string>
                                        <token-lower-case>
                                           <token-substring length="2">
                                                <token-op-attr name="Given Name"/>
                                            </token-substring>
                                        <token-substring length="6">
                                                   <token-op-attr name="Surname"/>
                                               </token-substring>
                                              </token-lower-case>
                                </arg-string>
                        </do-set-dest-password>
                 </actions>
        </rule>
</policy>
```

Specify Template: Example

This example DirXML Script policy specifies a template object if a user's Title attribute indicates that the user is a Manager (contains "Manager"). To view the policy in XML, see Create4.xml (../samples/Create4.xml).

```
<policy>
           <rule>
                        <description>Assign Manager Template if Title contains
Manager</description>
                        <conditions>
                                <and>
                                     <if-class-name op="equal">User</if-class-name>
                                      <if-op-attr name="Title" op="available"/>
                                        <if-op-attr mode="regex" name="Title"
op="equal">.*Manager.*</if-op-attr>
                                </and>
                            </conditions>
                            <actions>
                                    <do-set-op-template-dn>
                                             <arg-dn>
                                                    <token-text>Users\Manager
Template</token-text>
                                              </arg-dn>
                                     </do-set-op-template-dn>
                         </actions>
             </rule>
</policy>
```

Placement Policy

Placement policies determine where new objects are placed and what they are named in the Identity Vault and the connected application.

A Placement policy is required on the Publisher channel if you want object creation to occur in the Identity Vault. A Placement policy might not be necessary on the Subscriber channel even if you want object creations to occur in the connected application, depending on the nature of the destination data store. For example, no Placement policy is needed when synchronizing to a relational database because rows in a relational database do not have a location or a name.

- “Placement By Attribute Value: Example 1” on page 30
- “Placement By Attribute Value: Example 2” on page 31
- “Placement By Name: Example” on page 32

Placement By Attribute Value: Example 1

This example DirXML Script policy creates the user in a specific container based on the value of the Department attribute. To view the policy in XML, see Placement1.xml (../samples/Placement1.xml).

```
<policy>
          <rule>
                 <description>Department Engineering</description>
                 <conditions>
                      <and>
                              <if-class-name op="equal">User</if-class-name>
                          <if-op-attr mode="regex" name="Department"
op="equal">.*Engineering.*</if-op-attr>
                       </and>
                </conditions>
                <actions>
                      <do-set-op-dest-dn>
                            <arg-dn>
                                     <token-text>Eng</token-text>
```

```
                        <token-text>\</token-text>
                                  <token-op-attr name="CN"/>
                        </arg-dn>
                  </do-set-op-dest-dn>
             </actions>
      </rule>
      <rule>
            <description>Department HR</description>
            <conditions>
                  <and>
                        <if-class-name op="equal">User</if-class-name>
                        <if-op-attr mode="regex" name="Department"
op="equal">.*HR.*</if-op-attr>
                  </and>
            </conditions>
            <actions>
                  <do-set-op-dest-dn>
                        <arg-dn>
                              <token-text>HR</token-text>
                        <token-text>\</token-text>
                         <token-op-attr name="CN"/>
                          </arg-dn>
                  </do-set-op-dest-dn>
             </actions>
       </rule>
</policy>
```

Placement By Attribute Value: Example 2

This DirXML Script policy determines placement of a User or Organization Unit by the src-dn in the input document. To view the policy in XML, see Placement2.xml (../samples/Placement2.xml).

```
<policy>
         <rule>
                <description>PublisherPlacementRule</description>
                <conditions>
                           <or>
                                  <if-class-name op="equal">User</if-class-name>
                         <if-class-name op="equal">Organizational Unit</if-class-
name>
                           </or>
                           <or>
                                 <if-src-dn op="in-subtree">o=people, o=novell</
if-src-dn>
                           </or>
                  </conditions>
                  <actions>
                   <do-set-op-dest-dn>
                             <arg-dn>
                                 <token-text>People</token-text>
                                 <token-text>\</token-text>
                                 <token-unmatched-src-dn convert="true"/>
                              </arg-dn>
                         </do-set-op-dest-dn>
                    </actions>
          </rule>
</policy>
```

Placement By Name: Example

This example DirXML Script policy creates the user in a specific container based on the first letter of the user's last name. Users with a last name beginning with A-I are placed in the container Users1, while J-R are placed in Users2, and S-Z in Users3. To view the policy in XML, see Placement3.xml (../samples/Placement3.xml).

```
<policy>
         <rule>
               <description>Surname - A to I in Users1</description>
               <conditions>
                              <and>
                                     <if-class-name op="equal">User</if-class-name>
                             <if-op-attr mode="regex" name="Surname" op="equal">[A-
I].*</if-op-attr>
                               </and>
               </conditions>
               <actions>
                  <do-set-op-dest-dn>
                            <arg-dn>
                                   <token-text>Users1</token-text>
                              <token-text>\</token-text>
                                 <token-op-attr name="CN"/>
                             </arg-dn>
                      </do-set-op-dest-dn>
                </actions>
         </rule>
         <rule>
               <description>Surname - J to R in Users2</description>
               <conditions>
                           <and>
                                    <if-class-name op="equal">User</if-class-name>
                                     <if-op-attr mode="regex" name="Surname"
op="equal">[J-R].*</if-op-attr>
                           </and>
               </conditions>
               <actions>
               <do-set-op-dest-dn>
                    <arg-dn>
                                  <token-text>Users2</token-text>
                        <token-text>\</token-text>
                             <token-op-attr name="CN"/>
                             </arg-dn>
                     </do-set-op-dest-dn>
                </actions>
          </rule>
          <rule>
                 <description>Surname - S to Z in Users3</description>
                 <conditions>
                           <and>
                                    <if-class-name op="equal">User</if-class-name>
                                     <if-op-attr mode="regex" name="Surname"
```

```
op="equal">[S-Z].*</if-op-attr>
                              </and>
                  </conditions>
                  <actions>
                         <do-set-op-dest-dn>
                      <arg-dn>
                                      <token-text>Users3</token-text>
                                  <token-text>\</token-text>
                                   <token-op-attr name="CN"/>
                                  </arg-dn>
                           </do-set-op-dest-dn>
                 </actions>
        </rule>
</policy>
```

Command Transformation Policy

Command Transformation policies alter the commands that Identity Manager is sending to the destination data store by either substituting or adding commands. Intercepting a Delete command and replacing it with Modify, Move, or Disable command is an example of substituting commands in a Command Transformation policy. Creating a Modify command based on the attribute value of an Add command is a common example of adding commands in a Command Transformation policy.

In the most general terms, Command Transformation policies are used to alter the commands that Identity Manager executes as a result of the default processing of events that were submitted to the Identity Manager engine.

It is also common practice to include policies here that do not fit neatly into the descriptions of any other policy.

- “Convert Delete to Modify: Example” on page 33
- “Create Additional Operation: Example” on page 34
- “Setting Password Expiration Time: Example” on page 35

Convert Delete to Modify: Example

This DirXML Script policy converts a Delete operation to a Modify operation of the Login Disabled attribute. To view the policy in XML, see Comannd1.xml (../samples/Command1.xml).

```
<policy>
       <rule>
               <description>Convert User Delete to Modify</description>
               <conditions>
                        <and>
                                <if-operation op="equal">delete</if-operation>
                                <if-class-name op="equal">User</if-class-name>
                        </and>
               </conditions>
               <actions>
                       <do-set-dest-attr-value name="Login Disabled">
                               <arg-value type="state">
                                        <token-text>true</token-text>
                               </arg-value>
                      </do-set-dest-attr-value>
                      <do-veto/>
               </actions>
     </rule>
</policy>
```

Create Additional Operation: Example

This DirXML Script policy determines if the destination container for the user already exists. If the container doesn't exist, the policy creates an Add operation to create the Container object. To view the policy in XML, see Command2.xml (../samples/Command2.xml).

```
<policy>
    <rule>
          <description>Check if destination container already exists</description>
         <conditions>
                  <and>
                         <if-operation op="equal">add</if-operation>
                  </and>
         </conditions>
    <actions>
                  <do-set-local-variable name="target-container">
                          <arg-string>
                                    <token-dest-dn length="-2"/>
                          </arg-string>
                  </do-set-local-variable>
                  <do-set-local-variable name="does-target-exist">
                          <arg-string>
                                  <token-dest-attr class-name="OrganizationalUnit"
name="objectclass">
                                                 <arg-dn>
                                                                <token-local-variable
name="target-container"/>
                                                  </arg-dn>
                                  </token-dest-attr>
                          </arg-string>
                 </do-set-local-variable>
       </actions>
   </rule>
   <rule>
          <description>Create the target container if necessary</description>
          <conditions>
                    <and>
                           <if-local-variable name="does-target-exist"
op="available"/>
```

```
                        <if-local-variable name="does-target-exist" op="equal"/>
                    </and>
            </conditions>
            <actions>
                    <do-add-dest-object class-name="organizationalUnit"
direct="true">
                        <arg-dn>
                                <token-local-variable name="target-container"/>
                        </arg-dn>
                </do-add-dest-object>
                <do-add-dest-attr-value direct="true" name="ou">
                        <arg-dn>
                                <token-local-variable name="target-container"/>
                         </arg-dn>
                         <arg-value type="string">
                                 <token-parse-dn dest-dn-format="dot" length="1"
src-dn-format="dest-dn" start="-1">
                                                 <token-local-variable name="target-
container"/>
                                  </token-parse-dn>
                         </arg-value>
                </do-add-dest-attr-value>
            </actions>
    </rule>
</policy>
```

Setting Password Expiration Time: Example

This DirXML Script policy modifies the Identity Vault user's Password Expiration Time attribute. To view the policy in XML, see Command3.xml (../samples/Command3.xml).

```
<?xml version="1.0" encoding="UTF-8"?>
<policy xmlns:jsystem="http://www.novell.com/nxsl/java/java.lang.System">
    <rule>
             <description>Set password expiration time for a given interval from
current day</description>
             <conditions>
                     <and>
                          <if-operation op="equal">modify-password</if-operation>
                     </and>
            </conditions>
            <actions>
                      <do-set-local-variable name="interval">
                            <arg-string>
                                        <token-text>30</token-text>
                            </arg-string>
```

```
                        </do-set-local-variable>
                         <do-set-dest-attr-value class-name="User" name="Password
Expiration Time" when="after">
                                <arg-association>
                                        <token-association/>
                                </arg-association>
                                <arg-value type="string">
                                             <token-xpath
expression="round(jsystem:currentTimeMillis() div 1000 + (86400*$interval))"/>
                                </arg-value>
                         </do-set-dest-attr-value>
             </actions>
      </rule>
</policy>
```

Schema Mapping Policy

Schema Mapping policies hold the definition of the schema mappings between the Identity Vault and the connected system.

The Identity Vault schema is read from the Identity Vault. The Identity Manager driver for the connected system supplies the connected application's schema. After the two schemas have been identified, a simple mapping is created between the Identity Vault and the target application.

After a Schema Mapping policy is defined in the Identity Manager driver configuration, the corresponding data can be mapped.

It is important to note the following:

- The same policies are applied in both directions.
- All documents that are passed in either direction on either channel between the Identity Manager engine and the application shim are passed through the Schema Mapping policies.

See "Defining Schema Map Policies" in *NetIQ Identity Manager - Using Designer to Create Policies* for administrative information.

- "Basic Schema Mapping Policy: Example" on page 36
- "Custom Schema Mapping Policy: Example" on page 37

Basic Schema Mapping Policy: Example

This example DirXML Script policy shows the raw XML source of a basic Schema Mapping policy. However, when you edit the policy through Designer for Identity Manager, the default Schema Mapping editor allows the policy to be displayed and edited graphically. To view the policy in XML, see SchemaMap1.xml (../samples/SchemaMap1.xml).

```
<?xml version="1.0" encoding="UTF-8"?><attr-name-map>
       <class-name>
                  <app-name>WorkOrder</app-name>
                  <nds-name>DirXML-nwoWorkOrder</nds-name>
        </class-name>
        <class-name>
                   <app-name>PbxSite</app-name>
                   <nds-name>DirXML-pbxSite</nds-name>
        </class-name>
      <attr-name class-name="DirXML-pbxSite">
                   <app-name>PBXName</app-name>
                   <nds-name>DirXML-pbxName</nds-name>
        </attr-name>
        <attr-name class-name="DirXML-pbxSite">
                   <app-name>TelephoneNumber</app-name>
                   <nds-name>Telephone Number</nds-name>
        </attr-name>
        <attr-name class-name="DirXML-pbxSite">
                   <app-name>LoginName</app-name>
                   <nds-name>DirXML-pbxLoginName</nds-name>
         </attr-name>
         <attr-name class-name="DirXML-pbxSite">
                   <app-name>Password</app-name>
                   <nds-name>DirXML-pbxPassword</nds-name>
         </attr-name>
       <attr-name class-name="DirXML-pbxSite">
                   <app-name>Nodes</app-name>
                   <nds-name>DirXML-pbxNodesNew</nds-name>
       </attr-name>
   </attr-name-map>
```

Custom Schema Mapping Policy: Example

This example DirXML Script policy uses DirXML Script to perform custom Schema Mapping. To view this policy in XML, see SchemaMap2.xml (../samples/SchemaMap2.xml).

```
<?xml version="1.0" encoding="UTF-8"?><policy>
       <rule>
    <!--
    The Schema Mapping Policy can only handle one-to-one mappings.
    That Mapping Policy maps StudentPersonal addresses.
    This rule maps StaffPersonal addresses.
  -->
          <description>Publisher Staff Address Mappings</description>
          <conditions>
                 <and>
                              <if-local-variable name="fromNds" op="equal">false</
if-local-variable>
                              <if-xpath op="true">@original-class-name =
'StaffPersonal'</if-xpath>
                  </and>
          </conditions>
          <actions>
                <do-rename-op-attr dest-name="SA" src-name="Address/Street/Line1"/>
               <do-rename-op-attr dest-name="Postal Office Box" src-name="Address/
Street/Line2"/>
              <do-rename-op-attr dest-name="Physical Delivery Office Name" src-
name="Address/City"/>
              <do-rename-op-attr dest-name="S" src-name="Address/StatePr"/>
```

```
              <do-rename-op-attr dest-name="Postal Code" src-name="Address/
PostalCode"/>
          </actions>
  </rule>
  <rule>
          <description>Subscriber Staff Address Mappings</description>
    <!--
    The Schema Mapping Policy has already mapped addresses to StudentPersonal.
    This rule maps StudentPersonal to StaffPersonal.
  -->
          <conditions>
                <and>
                     <if-local-variable name="fromNds" op="equal">true</if-local-
variable>
                       <if-op-attr name="DirXML-sifIsStaff" op="equal">true</if-
op-attr>
                </and>
           </conditions>
           <actions>
                     <do-rename-op-attr dest-name="Address/Street/Line1" src-
name="StudentAddress/Address/Street/Line1"/>
                     <do-rename-op-attr dest-name="Address/Street/Line2" src-
name="StudentAddress/Address/Street/Line2"/>
                     <do-rename-op-attr dest-name="Address/City" src-
name="StudentAddress/Address/City"/>
                     <do-rename-op-attr dest-name="Address/StatePr" src-
name="StudentAddress/Address/StatePr"/>
                    <do-rename-op-attr dest-name="Address/PostalCode" src-
name="StudentAddress/Address/PostalCode"/>
            </actions>
      </rule>
</policy>
```

Output Transformation Policy

Output Transformation policies primarily handle the conversion of data formats from data that the Identity Manager engine provides to data that the application shim expects. Examples of these conversions include:

- Attribute value format conversion
- XML vocabulary conversion
- Custom handling of status messages returned from the Identity Manager engine to the application shim

All documents that the Identity Manager engine supplies to the application shim on either channel pass through the Output Transformation policies. Since the Output Transformation happens after schema mapping, all schema names are in the application namespace.

- “Attribute Value Conversion: Example” on page 39
- “Customer Handling of Status Messages:” on page 39

Attribute Value Conversion: Example

This example DirXML Script policy reformats the telephone number from the (nnn) nnn-nnnn format to the nnn.nnn.nnnn format. The reverse transformation can be found in the Input Transformation policy examples. To view the policy in XML, see Output_Transformation1.xml (../samples/Output_Transformation1.xml).

```
<policy>
       <rule>
                <description>Reformat all telephone numbers from (nnn) nnn-nnnn to
nnn.nnn.nnnn</description>
               <conditions/>
               <actions>
                            <do-reformat-op-attr name="telephoneNumber">
                                  <arg-value type="string">
                                                <token-replace-first
regex="^\((\d\d\d)\) *(\d\d\d)-(\d\d\d\d)$" replace-with="$1.$2.$3">
                                                            <token-local-variable
name="current-value"/>
                                                </token-replace-first>
                                  </arg-value>
                             </do-reformat-op-attr>
                </actions>
     </rule>
</policy>
```

Customer Handling of Status Messages:

This example DirXML Script policy detects status documents with a level not equal to success that also contain a child password-publish-status element within the operation data and then generates an e-mail message using the DoSendEmailFromTemplate action. To view the policy in XML, see Output_Transformation2.xml (../samples/Output_Transformation2.xml).

```
<?xml version="1.0" encoding="UTF-8"?>
      <policy>
                <description>Email notifications for failed password publications</
description>
                 <rule>
                             <description>Send e-mail for a failed publish password
operation</description>
                             <conditions>
                                       <and>
                                                  <if-global-variable mode="nocase"
name="notify-user-on-password-dist-failure" op="equal">true</if-global-variable>
                                                   <if-operation op="equal">status</
if-operation>
                                                    <if-xpath
op="true">self::status[@level != 'success']/operation-data/password-publish-
status</if-xpath>
                                        </and>
                           </conditions>
                           <actions>
                           <!-- generate email notification -->
                            <do-send-email-from-template notification-
dn="\cn=security\cn=Default Notification Collection" template-
dn="\cn=security\cn=Default Notification Collection\cn=Password Sync Fail">
                                  <arg-string name="UserFullName">
                                          <token-src-attr name="Full Name">
                                                <arg-association>
```

```
                                                                    <token-xpath
expression="self::status/operation-data/password-publish-status/association"/>
                                                 </arg-association>
                                            </token-src-attr>
                                       </arg-string>
                                  <arg-string name="UserGivenName">
                                             <token-src-attr name="Given Name">
                                                <arg-association>
                                                                     <token-xpath
expression="self::status/operation-data/password-publish-status/association"/>
                                                   </arg-association>
                                          </token-src-attr>
                                   </arg-string>
                                   <arg-string name="UserLastName">
                                        <token-src-attr name="Surname">
                                            <arg-association>
                                                    <token-xpath
expression="self::status/operation-data/password-publish-status/association"/>
                                                 </arg-association>
                                          </token-src-attr>
                                   </arg-string>
                                   <arg-string name="ConnectedSystemName">
                                              <token-global-variable
name="ConnectedSystemName"/>
                                   </arg-string>
                                   <arg-string name="to">
                                      <token-src-attr name="Internet Email Address">
                                        <arg-association>
                                                          <token-xpath
expression="self::status/operation-data/password-publish-status/association"/>
                                                   </arg-association>
                                          </token-src-attr>
                                    </arg-string>
                                    <arg-string name="FailureReason">
                                              <token-text/>
                                             <token-xpath expression="self::status/
child::text()"/>
                                    </arg-string>
                        </do-send-email-from-template>
               </actions>
      </rule>
</policy>
```

Input Transformation Policy

Input Transformation policies primarily handle the conversion of data formats from data that the application shim provides to data that the Identity Manager engine expects. Examples of these conversions include:

- Attribute value format conversion
- XML vocabulary conversion
- Driver heartbeat
- Custom handling of status messages returned from the application shim to the Identity Manager engine

All documents supplied to the Identity Manager engine by the application shim on either channel pass through the Input Transformation policies.

Attribute Value Format Conversion: Example

This example DirXML Script policy reformats the telephone number from the nnn.nnn.nnnn format to the (nnn) nnn-nnnn format. The reverse transformation can be found in “Output Transformation Policy” on page 38 examples. To view the policy in XML, see Input_Transformation1.xml (../samples/Input_Transformation1.xml).

```
<policy>
       <rule>
                <description>Reformat all telephone numbers from nnn.nnn.nnnn to
(nnn) nnn-nnnn</description>
               <conditions/>
               <actions>
                         <do-reformat-op-attr name="telephoneNumber">
                                   <arg-value type="string">
                                             <token-replace-first
regex="^(\d\d\d)\.(\d\d\d)\.(\d\d\d\d)$" replace-with="($1) $2-$3">
                                                       <token-local-variable
name="current-value"/>
                                             </token-replace-first>
                                   </arg-value>
                         </do-reformat-op-attr>
               </actions>
      </rule>
</policy>
```

Driver Heartbeat: Example

This DirXML Script policy creates a status heartbeat event. The driver’s heartbeat functionality is used to send a success message (HEARTBEAT: $driver) at each heartbeat interval. The message can be monitored by NetIQ Audit.The Identity Manager driver must support heartbeat, and heartbeat must be enabled at the driver configuration page. To view the policy in XML, see Input_Transformation2.xml (../samples/Input_Transformation2.xml).

```
<?xml version="1.0" encoding="UTF-8" ?>
 <policy>
     <rule>
        <description>Heartbeat Rule, v1.01, 040126, by Holger Dopp</description>
         <conditions>
                   <and>
                         <if-operation op="equal">status</if-operation>
                         <if-xpath op="true">@type="heartbeat"</if-xpath>
                   </and>
        </conditions>
        <actions>
                <do-set-xml-attr expression="." name="text1">
                     <arg-string>
                           <token-global-variable name="dirxml.auto.driverdn" />
             </arg-string>
               </do-set-xml-attr>
               <do-set-xml-attr expression="." name="text2">
                     <arg-string>
                                 <token-text>HEARTBEAT</token-text>
                     </arg-string>
              </do-set-xml-attr>
        </actions>
  </rule>
</policy>
```

Start Up Policies

When a driver is started, the Identity Manager engine sends a start-up event to the policy set. A Start-up event looks like this:

```
<nds dtdversion="4.0" ndsversion="8.x">
<source>
<product edition="Advanced" version="4.5">DirXML</product>
<contact>Novell, Inc.</contact>
</source>
<input>
<status level="success" type="startup"/>
</input>
</nds>
```

You can create policies on top of a start-up event to perform custom actions at the driver start up. For example, initialization of persistent driver-scope variables, sending a notification from the DirXML Script to an external auditing or monitoring system, and so on.

There are no plug-ins currently available to create policies in the Start-up policy set. To create these policies, run the following steps:

1. Create policies in any of the driver policy sets. For example, Subscriber Command Transformation Policy set.
2. Link the policies to the Start-up policy set.
   - **2a** Go to **Driver Properties**, click the **General** tab, then select the DirXML-Policies attribute from the **Valued Attributes** list and click **Edit**.
   - **2b** In the Edit Attribute dialog, select the desired policy, then click the **Edit** button. The TypeNamedUiHandlerLevel attribute specifies the order in which you want to execute the policy. Policy at level 0 is executed first followed by level 1, 2, and so on. The TypeNamedUiHandlerInterval attribute specifies the policy set in which you want to execute the policy. For the Start-up policy, set the value to 15, then click **OK**.

Shut-Down Policies

When a driver is stopped, the Identity Manager engine sends a stop event to the policy set. A Shut-Down event looks like this:

```
<nds dtdversion="4.0" ndsversion="8.x"><source>
<product edition="Advanced" version="4.5">DirXML</product>
<contact>Novell, Inc.</contact>
</source>
<input>
<status level="success" type="shutdown"/>
</input>
</nds>
```

You can create policies on top of a start-up event to perform custom actions at the driver shutdown. For example, saving the state of driver-scope variables by writing them to the eDirectory objects, making use of Java to perform some customized tasks, and so on.

There are no plug-ins currently available to create policies in the Shut-Down policy set. To create these policies, run the following steps:

1. Create policies in any of the driver policy sets. For example, Subscriber Command Transformation Policy set.
2. Link the policies to the Shut-down policy set.
   - **2a** Go to **Driver Properties**, click the **General** tab, then select the DirXML-Policies attribute from the **Valued Attributes** list and click **Edit**.
   - **2b** In the Edit Attribute dialog, select the desired policy, then click the **Edit** button. The TypeNamedUiHandlerLevel attribute specifies the order in which you want to execute the policy. Policy at level 0 is executed first followed by level 1, 2, and so on. The TypeNamedUiHandlerInterval attribute specifies the policy set in which you want to execute the policy. For the Shut-down policy, set the value to 16, then click **OK**.

Defining Policies

All policies are defined in one of two ways:

- Using the Policy Builder interface to generate DirXML Script. Existing, non-XSLT rules are converted to DirXML Script automatically upon import.
- Using XSLT style sheets.

Schema Mapping policies can also be defined (and usually are) using a schema mapping table.

Policy Builder and DirXML Script

The Policy Builder interface is used to define the majority of policies you might implement. The Policy Builder interface uses a graphical environment to enable you to easily define and manage policies.

The underlying functionality of rule creation within Policy Builder is provided by DirXML Script, however, you do not need to work directly with DirXML Script.

Instead, you have access to a wide variety of conditions you can test, actions to perform, and dynamic values to add to your policies. Each option is presented using intelligent drop-down lists, providing only valid selections at each point, and quick links to common values.

See *NetIQ Identity Manager - Using Designer to Create Policies*, for more information on Policy Builder. See “DirXML Script” on page 45 for more information on DirXML Script.

TIP: Although it is not necessary for using Policy Builder, the DirXML Script DTD is available in the Identity Manager DTD Reference Documentation.

---

4 Understanding Policy Components

DirXML Script

DirXML Script is the primary method of implementing Identity Manager policies. It describes a policy that is implemented by an ordered set of rules. A rule consists of a set of conditions to be tested and an ordered set of actions to be performed when the conditions are met.

DirXML Script is created using the Policy Builder, which provides a GUI interface for easy use.

Identity Manager is an XML-based application, and DirXML Script uses XML documents to modify and manipulate the data being sent between the Identity Vault and the external data store. To understand DirXML Script, you need to understand XML. For more information on XML, see the W3C Extensible Markup Language (XML) (http://www.w3.org/XML/) Web site.

DirXML Script has a document type definition (DTD) that defines how DirXML Script works. To read the DTDs that Identity Manager uses, see the Identity Manager DTD Reference (https://www.netiq.com/documentation/identity-manager-developer/dtd-documentation.html).

Naming Conventions for Policies

Identity Manager contains naming conventions for policies that are stored in a driver or library.

Naming Convention for Driver Policy Objects

Driver policy objects are policies that exist underneath a driver or channel object. These policies are usually consumed only by this driver. A driver can contain many policies; without the naming conventions, it is easy to be confused.

```
<channel>-<policyset>[-<feature name>][WhatIsThisPolicyDoing]
```

Table 4-1 *Driver Policy Object Naming Convention*

Policy Set	DirXML Script Policy	Style Sheet Policy	Samples
Subscriber Event Transformation	sub-etp	sub-ets	`sub-etp-VetoAdds`, `sub-ets-ChangeRenameToMove`
Subscriber Matching	sub-mp	sub-ms	
Subscriber Creation	sub-cp	sub-cs	
Subscriber Placement	sub-pp	sub-ps	
Subscriber Command Transformation	sub-ctp	sub-cts	
Publisher Event Transformation	pub-etp	pub-ets	`pub-etp-VetoAdds`, `pub-ets-ChangeRenameToMove`
Publisher Matching	pub-mp	pub-ms	
Publisher Creation	pub-cp	pub-cs	
Publisher Placement	pub-pp	pub-ps	
Publisher Command Transformation	pub-ctp	pub-cts	`pub-ctp-HandleFromMerge`, `pub-cts-PasswordSync`
Input Transformation	itp	its	
Output Transformation	otp	ots	
Schema Mapping	smp	sms	

Naming Convention for Policy Objects in Libraries

Policy objects in policy libraries might be consumed by more than one driver in different policy sets and channels. The naming conventions for library policies are adapted from the driver policies.

```
lib-<feature name>-<WhatIsThisPolicyDoing>[-<channel>][-<policyset>]
```

- **Lib:** Static prefix to mark the policy as a library policy. This is important so that you can tell which policies belong to that driver and which policies do not.
- **Feature Name:** Short name that describes the feature this policy is implementing. Examples might be CredProv for Credential Provisioning or PwdSync for Password Synchronization. The feature name groups multiple policies together.
- **WhatIsThisPolicyDoing:** A compound word or phrase where the words are joined without spaces and are capitalized within the compound word. This word or phrase is a brief descriptive name for the policy.

For example:

```
lib-CredProv-ConvertPayload-opt
lib-CredProv-ProcessPayload-itp
lib-CredProv-RequiredAttributes-sub-cp
lib-CredProv-Triggers-cub-ctp
```

Variables

DirXML Script supports two kinds of variables: global and local. A global variable is a variable that is defined in a global configuration value for the driver or the driver set. Global variables are by definition read-only. A local variable is a variable that is set by a policy. A local variable can exist in one of two different scopes: policy or driver. A policy-scoped variable is only visible during the processing of the current operation by the policy that sets the variable. A driver-scoped variable is visible from all DirXML Script policies running within the same driver until the driver is stopped.

A variable name must be a legal XML name. For information on what is a legal XML name, see W3C Extensible Markup Language (XML) (http://www.w3.org/TR/2006/REC-xml11-20060816/#sec-suggested-names).

There are a number of global and local variables that are automatically defined.

Table 4-2 *Defined Global and Local Variables*

Name	Type	Description
dirxml.auto.driverdn	global/string	Slash format DN of the current driver.
dirxml.auto.localserverdn	global/string	DN of the local server where the current driver is running in LDAP format.
dirxml.auto.driverguid	global/string	GUID of the current driver.
dirxml.auto.treename	global/string	Tree name of the local eDirectory instance.
fromNds	policy local/boolean	True if the source data store is the Identity Vault. False if the source data store is the connected application.
destQueryProcessor	policy local/java object	Instance of XdsQueryProcessor used to query the destination data store.
srcQueryProcessor	policy local/java object	Instance of XdsQueryProcessor used to query the source data store.
destCommandProcessor	policy local/java object	Instance of XdsCommandProcessor used to execute the command using the destination data store.
srcCommandProcessor	policy local/java object	Instance of XdsCommandProcessor used to execute the command using the source data store.
dnConverter	policy local/java object	Instance of DNConverter.
current-node	policy local/node set	The loop variable for each iteration of the for each element.
current-value	policy local/node set	The loop variable for each iteration of the reformat operation attribute.

Name	Type	Description
current-op	policy local/node set	The current operation. Setting this variable using the <do-set-local-variable> element causes the first operation specified by <arg-node-set> to become the current operation for the remainder of the current policy execution or until it is set to another value. The new current operation must be an element sibling of the original current operation and must have been added by the current policy.

Variable Expansion

Many conditions, actions, and tokens support dynamic variable expansion in their attributes or content. Where supported, an embedded reference of the form $<variable-name>$ is replaced with the value of the local or global variable with the given name. $<variable-name>$ must be a legal variable name. For information on what is a legal XML name, see W3C Extensible Markup Language (XML) (http://www.w3.org/TR/2004/REC-xml-20040204/#NT-Name).

If the given variable does not exist, the reference is replaced with the empty string. Where it is desirable to use a single $ and not have it interpreted as a variable reference, it should be escaped with an additional $ (for example, You owe me $$100.00). See the Identity Manager DTD Reference Documentation (https://www.netiq.com/documentation/identity-manager-developer/dtd-documentation.html) for the content attributes that support variable expansion. For more information on the local variables scope, see Variables. If the same local variable exists in the policy scope and the driver scope, the variable in the policy scope takes precedence. For information on GCV and their precedence, see Global configuration Values in the Identity Manager DTD Reference Documentation (https://www.netiq.com/documentation/identity-manager-developer/dtd-documentation.html).

Date/Time Parameters

Tokens that deal with dates and times have arguments that deal with the format, language, and time zone of the date and time representation. Date format arguments can be specified with a '!' character or without a '!' character. If the format begins with a '!' character, then the format is a named format. Legal names are defined in Table 4-3 on page 48.

Table 4-3 *Legal Date/Time Parameters*

Name	Description
!CTIME	Number of seconds since midnight, January 1, 1970. (Compatible with eDirectory time syntaxes).
!JTIME	Number of milliseconds since midnight, January 1, 1970. (Compatible with Java time).
!FILETIME	Number of 100-nanosecond intervals since January 1, 1601 (Compatible with Win32 FILETIME).
!FULL.TIME	Language-specific FULL time format.
!LONG.TIME	Language-specific LONG time format.
!MEDIUM.TIME	Language-specific MEDIUM time format.

Name	Description
!SHORT.TIME	Language-specific SHORT time format.
!FULL.DATE	Language-specific FULL date format.
!LONG.DATE	Language-specific LONG date format.
!MEDIUM.DATE	Language-specific MEDIUM date format.
!SHORT.DATE	Language-specific SHORT date/time format.
!FULL.DATETIME	Language-specific FULL date/time format.
!LONG.DATETIME	Language-specific LONG date/time format.
!MEDIUM.DATETIME	Language-specific MEDIUM date/time format.
!SHORT.DATETIME	Language-specific SHORT date/time format.

If the format does not begin with '!', then it is interpreted as a custom date/time format conforming to the patterns recognized by the Java class java.text.SimpleDateFormat.

Language arguments can be specified by an identifier that conforms to IETF RFC 3066. The list of identifiers understood by the system can be obtained by calling the Java class java.util.Locale.getAvailableLocales() and replacing all underscores in the result with hyphens. If a language argument is omitted or blank, then the default system language is used.

Time zone arguments can be specified in any identifier recognizable by the Java class java.util.TimeZone.getTimeZone(). A list of identifiers understood by the system can be obtained by the Java class calling java.util.TimeZone.getAvailableIDs(). If a time zone argument is omitted or blank, then the default system time zone is used.

Regular Expressions

A regular expression is a formula for matching text strings that follow some pattern. Regular expressions are made up of normal characters and metacharacters. Normal characters include uppercase and lowercase letters and digits. Metacharacters have special meanings. The following table contains some of the most common metacharacters and their meanings.

Table 4-4 *Common Regular Expressions*

Metacharacter	Description
.	Matches any single character.
$	Matches the end of the line.
^	Matches the beginning of a line.
*	Matches zero or more occurrences of the character immediately preceding.
\	Literal escape character. It allows you to search for any of the metacharacters. For example \$ finds $1000 instead of matching at the end of the line.
[ ]	Matches any one of the characters between the brackets.

Metacharacter	Description
[0-9]	Matches a range of characters with the hyphen. The example matches any digit.
[A-Za-z]	Matches multiple ranges. The example matches all uppercase and lowercase letters.
(?u)	Enables Unicode-aware case folding. This flag can impact performance.
(?i)	Enables case-insensitive matching.

The Argument Builder is designed to use regular expressions as defined in Java. For information, see the Oracle Java documentation for your version of Java.

XPath 1.0 Expressions

The arguments to some conditions, actions, and tokens use XPath 1.0 expressions. XPath is a language created to provide a common syntax and semantics for the functionality shared between XSLT and XPointer. XPath is used primarily for addressing parts of an XML document, but also provides basic facilities for manipulation of strings, numbers, and Booleans.

The XPath specification requires that the embedding application provide a context with several application-defined pieces of information. In the DirXML Script (see “DirXML Script” on page 45), XPath is evaluated with the following context:

- The context node is the current operation executed by the policy, unless otherwise specified in the description of the expression.

  A Modify event in Identity Manager looks like this:

```
<nds dtdversion="3.5" ndsversion="8.x">
  <source>
    <product version="4.0.1">DirXML</product>
    <contact>Novell, Inc.</contact>
  </source>
  <input>
    <modify class?name="User" event?id="656B0450E1A3BC5C6525780D003E7F4D ?
1294053788689" from?merge="true"
qualified?src?dn="O=data\OU=users\CN=username" src?dn="data\users\username"
src?entry?id="33045">
      <association>656B0450E1A3BC5C6525780D003E7F4D</association>
      <modify?attr attr?name="DirXML?EntitlementRef">
        <remove?all?values/>
      </modify?attr>
      <modify?attr attr?name="CN">
        <add?value>
          <value naming="true" timestamp="1294053459#23"
type="string">username</value>
        </add?value>
      </modify?attr>
    </modify>
  </input>
</nds>
```

When a policy rule is executed on the above event, the context node is set to the `Modify` node instead of `/`. To obtain the values of some of the nodes, start the XPath expression from the `Modify` node instead of `/nds/input/modify`. For example, the XPath expression for obtaining the `class-name` attribute should be `@class-name` instead of `nds/input/modify/@class-name`. The XPath expression for extracting the value of `attr-name=CN` can be `modify-attr[@attr-name="CN"]/add-value/value/` or `modify-attr/add-value/value/`.

- The context position and size are 1.
- There are several available variables:
  - Variables available as parameters to style sheets within Identity Manager (currently fromNds, srcQueryProcessor, destQueryProcessor, srcCommandProcessor, destCommandProcessor, and dnConverter).
  - Global configuration variables.
  - Local policy variables.
  - If there is a name conflict between different variable sources, the order of precedence is local (policy scope), local (driver scope), and global.
  - Because of the XPath syntax, any variable that has a colon character in its name is not accessible from XPath.
- There are several namespaces definitions available.
  - Any namespaces explicitly declared on the `<policy>` element by using the `XMLNS:prefix`.
  - The following implicitly defined namespaces (unless the same prefix has been explicitly defined):
    - `xmlns:js="http://www.novell.com/nxsl/ecmascript"`
    - `xmlns:es="http://www.novell.com/nxsl/ecmascript"`
    - `xmlns:query="http://www.novell.com/nxsl/java/com.novell.nds.dirxml.driver.XdsQueryProcessor”`
    - `xmlns:cmd="http://www.novell.com/nxsl/java/com.novell.nds.dirxml.driver.XdsCommandProcessor”`
    - `xmlns:jdbc="urn:dirxml:jdbc"`
  - Any namespace prefix that is not otherwise mapped is automatically mapped to `http://www.novell.com/nxsl/java/<prefix>` , if the prefix is the fully qualified class name of the Java class that can be resolved to an available Java class through introspection.
  - Namespace declarations to associate a prefix with a Java class must be declared with the policy element.
- There are several available functions:
  - All built-in XPath 1.0 functions.
  - Java extension functions, as provided by NXSL.
    - Java extension functions are accessed through a namespace prefix mapped to a URI of the form: `http://www.novell.com/nxsl/java/<fully-qualified-class-name>`.
    - For convenience, any prefix that is not otherwise mapped is mapped to `http://www.novell.com/nxsl/java/<prefix>,` if the prefix is the fully qualified class name of a Java class that can be discovered through introspection.
  - ECMAScript extension functions, as provided by NXSL:
    - The ECMAScript extension function definitions come from the set of ECMAScript resources associated with the driver.

- The ECMAScript extension functions are accessed through a namespace prefix mapped to the URI `http://www.novell.com/nxsl/ecmascript`.
- The prefixes `js` and `es` are both implicitly mapped to `http://www.novell.com/nxsl/ecmascript` unless otherwise explicitly defined.

The W3 Web site (http://www.w3.org/TR/1999/REC-xpath-19991116) contains more information about XPath.

XPath Examples

Here are some simple XPath examples commonly used in Identity Manager:

Add Event

```
<input>
<add cached-time="20130423053016.248Z" class-name="User" event-id="rj-idmdt-
122&#35;20130423053016#1#1:71e2b5fd-cf71-4ebc-06a1-fdb5e27171cf" qualified-src-
dn="O=data\OU=users\CN=sfpuserz" src-dn="\Rose-Mayflower-2\data\users\sfpuserz"
src-entry-id="39025" timestamp="1366695016#45">
      <add-attr attr-name="Postal Code">
        <value timestamp="1366695016#24" type="string">Sacramento</value>
      </add-attr>
      <add-attr attr-name="OU">
        <value timestamp="1366695016#22" type="string">DCM</value>
      </add-attr>
      <add-attr attr-name="Title">
        <value timestamp="1366695016#16" type="string">senior manager</value>
      </add-attr>
      <add-attr attr-name="co">
        <value timestamp="1366695016#18" type="string">USA</value>
      </add-attr>
      <add-attr attr-name="Telephone Number">
       <value timestamp="1366695016#12" type="teleNumber">+1 818 936-6205</value>
      </add-attr>
      <add-attr attr-name="S">
        <value timestamp="1366695016#19" type="string">California</value>
      </add-attr>
      <add-attr attr-name="Given Name">
        <value timestamp="1366695016#11" type="string">sfpuserz</value>
      </add-attr>
      <add-attr attr-name="company">
        <value timestamp="1366695016#23" type="string">Francos</value>
      </add-attr>
      <add-attr attr-name="Surname">
        <value timestamp="1366695016#26" type="string">joe</value>
      </add-attr>
      <add-attr attr-name="workforceID">
        <value timestamp="1366695016#15" type="string">800001</value>
      </add-attr>
      <add-attr attr-name="CN">
        <value timestamp="1366695016#45" type="string">sfpuserz</value>
      </add-attr>
    </add>
</input>
```

Element	Description
add-attr[@attr-name="Surname"]/value	Returns the value of the add-value node for the Surname attribute.
add-attr[@attr-name=" Facsimile Telephone Number" /value/component[@name="faxNumber"]	Returns the value of the fax number from the Facsimile Telephone Number structured attribute.

Modify Event

```
<input>
    <modify class name="User" event id="656B0450E1A3BC5C6525780D003E7F4D
1294053788689" from merge="true" qualified src dn="O=data\OU=users\CN=username"
src dn="data\users\username" src entry id="33045">
      <association>656B0450E1A3BC5C6525780D003E7F4D</association>
      <modify attr attr name="DirXML EntitlementRef">
        <remove all values/>
      </modify attr>
      <modify attr attr name="CN">
        <add value>
          <value naming="true" timestamp="1294053459#23" type="string">username</
value>
        </add value>
      </modify attr>
    </modify>
</input>
```

Element	Description
@src-dn	Returns the value of the src-dn attribute inside the event node.
@event-id	Returns the value of the event-id attribute.
modify-attr[@attr-name="CN"]/add-value/value	Returns the value of the add-value node for the Surname attribute.

Nested Groups

By default, the Identity Manager engine, when reading or searching the Member and Group Member attributes of Identity Vault objects, returns only those values that are “static” values. Static values are objects that received group membership by direct assignment to the group rather than inherited assignment through a nested group.

If you want the Identity Manager engine’s searches to return values inherited through nested groups, you can create policies (and stylesheets) that search for and read the “calculated” values for the Member and Group Membership attributes. Calculated values include objects that are either 1) statically assigned membership or 2) dynamically assigned membership by virtue of the nested group and the dynamic group hierarchy calculations used by the Identity Vault. You implement this behavior in policies and stylesheets by using the following pseudo attributes: `[pseudo].Member` and `[pseudo].Group Membership`. A single query operation can contain only the pseudo attributes or the real attributes; mixing both attributes in the same query will result in an error.

If you want to change the Identity Manager engine default so that it always searches for and reads the “calculated” values for the Member and Group Membership attributes, use the **Revert to Calculated Membership Value Behavior** engine control value. Changing this value causes the Identity Manager engine to revert to the method used prior to Identity Manager 3.6.1. In pre-3.6.1 versions, the Identity Manager engine's search of the Member and Group Member attributes retrieved all “calculated” values. For information about changing the value, see “Driver Properties” in the *NetIQ Identity Manager Driver Administration Guide*.

5 Downloading Identity Manager Policies

NetIQ has provided sample policies you can download and use in your environment. The policies are available at the NetIQ Support Web site (http://support.novell.com/patches.html). To download the policies:

1. Go to the NetIQ Support Web site (http://support.novell.com/patches.html).
2. On the left, click **Download** > **Patches** > **Patch Finder**.
3. Select **Identity Manager** in the product field, then click **Search**.
4. Click **Identity Manager 2.0** and **Identity Manager 2.0.1**.

   These policies can be used with any version of Identity Manager.
5. Browse to and select the desired policy.

   Table 5-1 contains a list of the policies available for download.
6. Select **proceed to download**, to download the policy.
7. Click **download** by the file name.
8. Click **Save**, then browse to and select a location to save the file.
9. Click **Save**, then click **Close**.
10. Extract the file, then read the `How_To_Install.rtf` file for installation instructions.

Table 5-1 *Downloadable Policies*

Name	File Name
Policy to Place by Surname	`placebyname.tgz`
Policy: Reset value of the email attribute	`pushback.tgz`
Policy to enforce the presence of attributes	`requiredattrs.tgz`
Policy: Create email from GivenName & Surname	`setemailname.tgz`
Policy: Create FullName from GivenName, Surname	`synthfullname.tgz`
Policy: Convert First/Last name to uppercase	`uppercasenames.tgz`
Policy to add user to group based on Title	`addcreategroups.tgz`
Policy: Assign template to user based on title	`assigntemplate.tgz`
Disable user account and move when terminated	`dismvonterm.tgz`
Policy to filter events	`filterby.tgz`
Govern Groups for user based on the title attribute	`groupchange.tgz`

To use Designer to import the files, see "Importing a Policy From an XML File" in *NetIQ Identity Manager - Using Designer to Create Policies*. To use iManager to import the files, see "Importing a Policy from an XML File" in *NetIQ Identity Manager Policies in iManager Guide*.

6 Defining Policies by Using XSLT Style Sheets

XSLT, which is a standard language for transforming XML documents, can be used for implementing policies as XSLT style sheets. The XSLT processor in the Identity Manager engine is compliant with the 16 November 1999 W3C recommendation. For the relevant specifications, see the following:

- XSL Transformations (XSLT) (http://www.w3.org/TR/1999/REC-xslt-19991116)
- XML Path Language (XPath) (http://www.w3.org/TR/1999/REC-xpath-19991116)

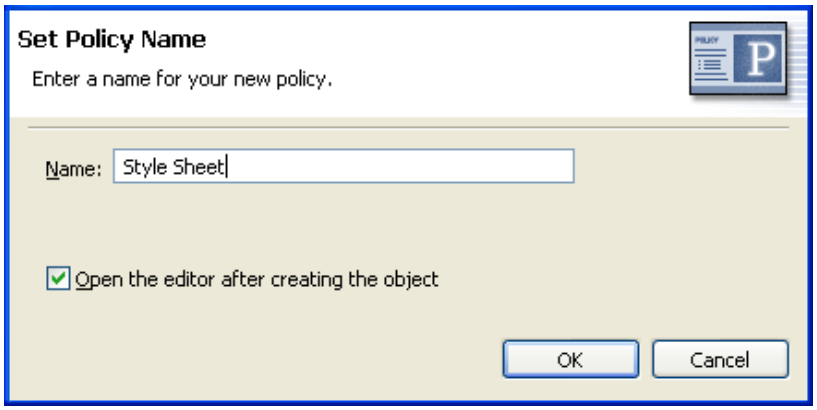
The following sections describe the specifics of using XSLT style sheets with Identity Manager.

Managing XSLT Style Sheets in Designer

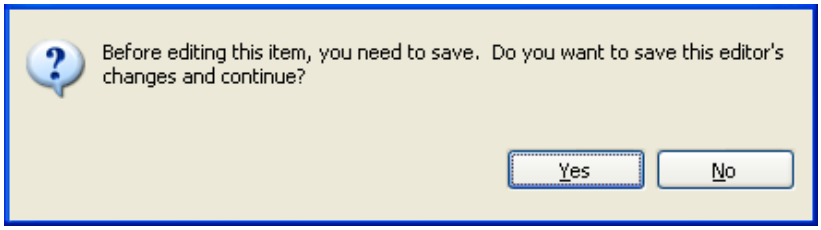
XSLT policy style sheets can be added, modified, and deleted using Designer’s XML Editor. The following sections provide details:

Adding an XSLT Style Sheet in Designer

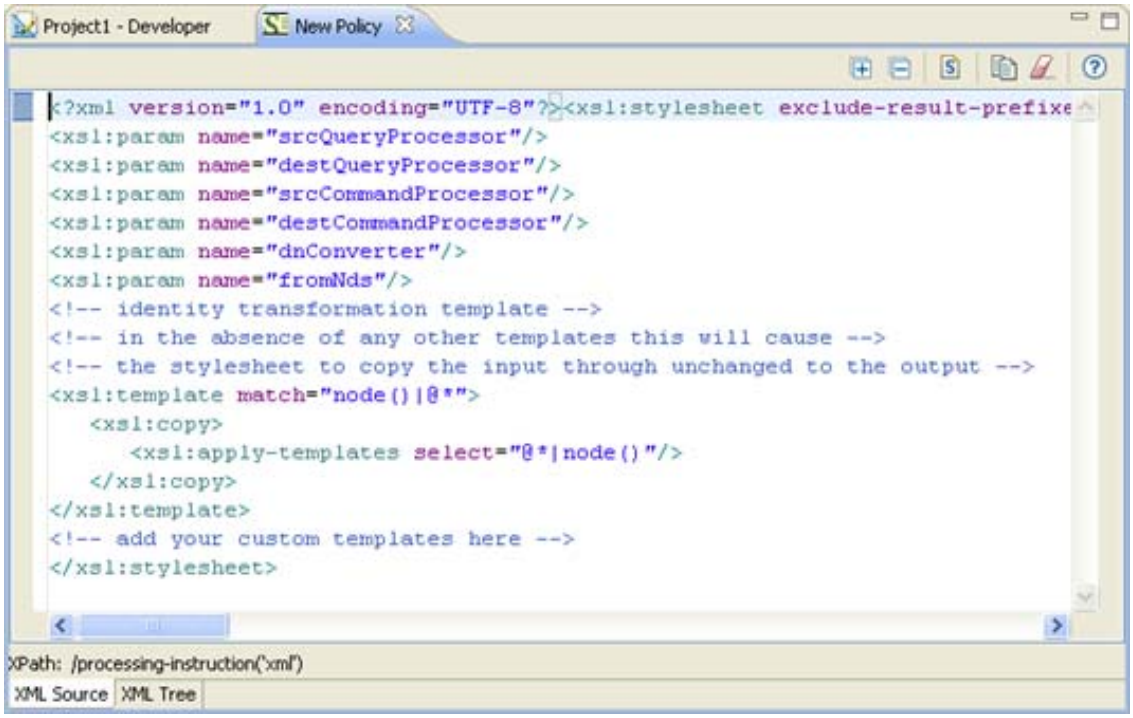
1. Open a project in Designer and select the **Outline** tab.
2. Select the driver and location where you want the style sheet.
3. Right-click and select **New > XSLT**.
4. Specify the name of the style sheet.
5. Select **Open Editor after creating policy**, then click **OK**.



6 Select **Yes** to save the project before editing the new style sheet.



7 Add the style sheet information below the line **Add your custom templates here**.



8 Click **File > Save to**, to save the style sheet.

Modifying an XSLT Style Sheet in Designer

1. Open a project in Designer and select the **Outline** tab.
2. Select the XSLT style sheet you want to modify.
3. Right-click, then select **Edit**.

   Modify the style sheet as desired. To clear the existing style sheet content, click **Clear**  in the XML editor toolbar.

Deleting an XSLT Style Sheet in Designer

1. Open a project in Designer and select the **Outline** tab.
2. Select the XSLT style sheet that you want to delete, right-click, then select **Delete**.

   Alternatively, you can clear the XSLT policy without deleting the object. To do this, right-click the XSLT policy, then select **Clear**.

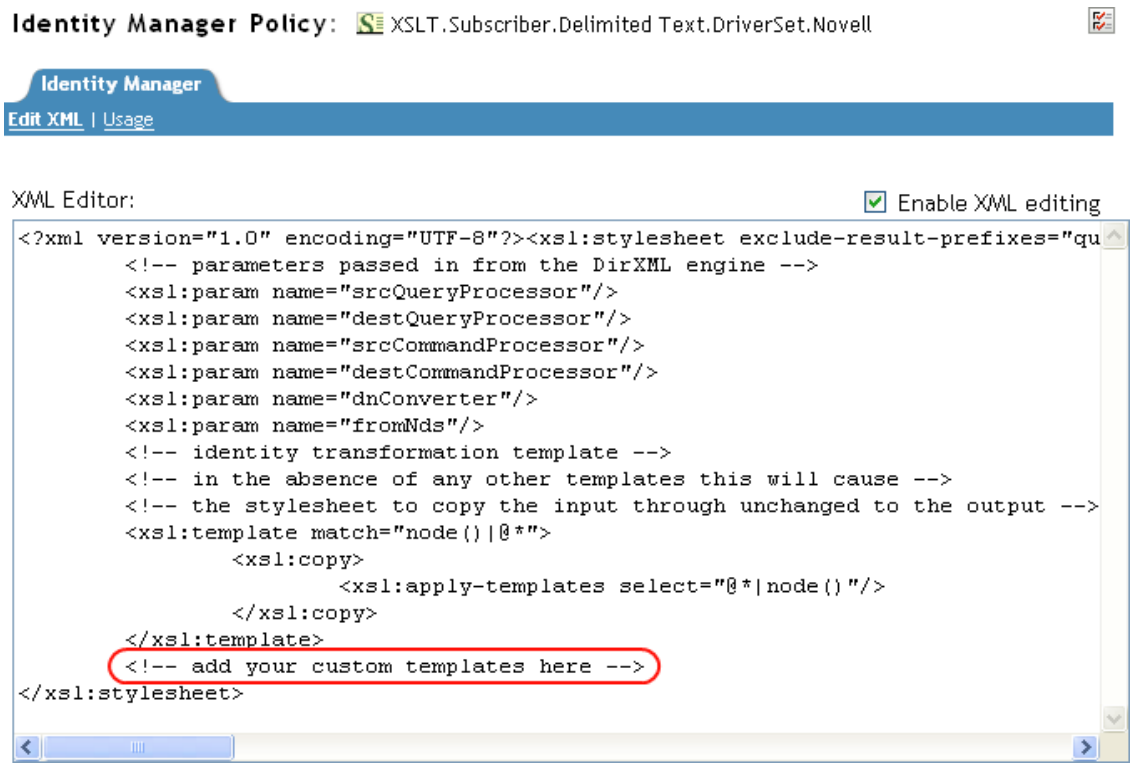
Managing XSLT Style Sheets in iManager

XSLT policy style sheets are added, modified, and deleted using iManager. The following sections provide details:

- "Adding an XSLT Policy in iManager" on page 59
- "Modifying an XSLT Style Sheet in iManager" on page 60
- "Deleting an XSLT Style Sheet in iManager" on page 60

Adding an XSLT Policy in iManager

1. Open the Identity Manager Driver Overview for the driver you want to manage.
2. Click the icon representing the policy were you want to add the XSLT style sheet.
3. Click **Insert**.
4. Provide a name for the new XSLT style sheet, select **XSLT**, then click **OK**.
5. Select **Enable XML Editing** to edit the XSLT style sheet.
6. Add the style sheet information below the line **add your custom templates here**.



7 Click **OK** to save the XSLT style sheet.

Modifying an XSLT Style Sheet in iManager

1 Open the Identity Manager Driver Overview for the driver you want to manage.
2 Click the icon representing the policy where the XSLT style sheet you want to modify is stored.
3 Select the XSLT style sheet you want to modify from the list of policies, then click Edit.
4 Edit the XSLT style sheet, then click **OK**.

Deleting an XSLT Style Sheet in iManager

1 Open the Identity Manager Driver Overview for the driver you want to manage.
2 Click the icon representing the policy where the XSLT style sheet you want to delete is stored.
3 Select the XSLT style sheet you want to delete from the list of policies, then click Delete.
4 Click **OK**, to verify that you want to delete the XSLT style sheet.

Prepopulated Information in the XSLT Style Sheet

When you create a new style sheet in iManager or Designer, it is prepopulated with a style sheet that implements the identity transformation. In the absence of additional templates, the identity transformation allows the input XML document to pass through the style sheet unchanged. You usually implement policy by adding additional templates to act on only the XML that you want to be changed. If your style sheet is being used to translate a document to or from an XML vocabulary that is different than XDS (such as the Input and Output Transformations for the SOAP and Delimited Text drivers) you might need to remove the identity template.

Using the Parameters that Identity Manager Passes

The Identity Manager engine passes the policy style sheets the following parameters to the style sheet:

Table 6-1 *Style Sheet Parameters*

Parameter	Description
srcQueryProcessor	A Java object that implements the XdsQueryProcessor interface. This allows the style sheet to query the source data store for more information.
destQueryProcessor	A Java object that implements the XdsQueryProcessor interface. This allows the style sheet to query the destination data store for more information.
srcCommandProcessor	A Java object that implements the XdsCommandProcessor interface. This allows the style sheet to write back a command to the event source.
destCommandProcessor	A Java object that implements the XdsCommandProcessor interface. This allows the style sheet to issue a command directly to send a command to the destination data store.
dnConverter	A Java object that implements the XdsCommandProcessor interface.This allows the style sheet to convert Identity Vault object DNs from one format to another. For more information, see Interface DNCoverter (http://developer.novell.com/ndk/doc/dirxml/dirxmlbk/api/com/novell/nds/dirxml/driver/DNConverter.html).
fromNds	A Boolean value that is True if the source data store is the Identity Vault and False if it is the connected application.

When you create a new style sheet in iManager or Designer, it is prepopulated with a style sheet that contains the declarations for these parameters.

When using the query and command parameters with the schema mapping policies, input transformation policies, and output transformation policies, the following limitations apply:

- 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 references are added to the query.
- Responses from the application shim are in the form returned by the shim with no modification or schema mapping performed and no resolution of association references.
- Queries issued to the Identity Vault must be in the form expected by the Identity Vault. In other words, schema names must be in the Identity Vault namespace and the query must be XDS. Association references are not resolved.
- Responses from the application shim are in the form returned by the shim with no modification or schema mapping performed.

Query Processors

Use of the query processors depends on the XSLT implementation of extension functions. To make a query, you need to declare a namespace 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.XdsQueryProcessor"
```

When you create a new style sheet in iManager or Designer, it is prepopulated with the namespace declaration. For more information about query processors see Class XdsQueryProcessor (http://developer.novell.com/ndk/doc/dirxml/dirxmlbk/api/com/novell/nds/dirxml/driver/XdsQueryProcessor.html).

The following example uses one of the query processors (the long lines are wrapped and do not begin with a <): To view the style sheet, see Query_Processors.xsl (../samples/Query_Processors.xsl).

```
<!-- Query object name queries NDS for the passed object name -->

<xsl:template name="query-object-name">
   <xsl:param name="object-name"/>

<!-- build an xds query as a result tree fragment -->
   <xsl:variable name="query">
            <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>
   </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>
```

Here is another example.

```
<?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>
```

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. However, the XSLT processor implements extension functions that 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 “Query Processors” on page 61 using the query processor, and the following example that illustrates using Java for string manipulation. The long lines are wrapped and do not begin with a <. To view the style sheet, see Extension_Functions.xsl (../samples/Extension_Functions.xsl).

```
<!-- 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>
```

Creating a Password: Example Creation Policy

The following style sheet can be used for a Creation policy. It creates a user, generates a password for the user from the user's Surname and CN attributes, and performs an identity transformation that passes through everything in the document except the events you are trying to intercept and transform. To view the style sheet, see Create_Password.xsl (../samples/Create_Password.xsl).

```
<?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>
```

While constructing DirXML-PasswordSyncStatus, you may can encounter values like:

```
39DB7DED8436EE4DF38039DB7DED84362014032514142272100000000000 1Code(-8032) Operation
vetoed by policy
```

The value of DirXML-PasswordSyncStatus is composed of the following:

- The first 32 bytes represent the GUID of the driver
- The next 17 bytes represent the Date/Time in yyyyMMddHHmmssSSS format
- The next 8 bytes are 00000000

- The next 4 bytes indicate any one of the following status codes:
  - 0000: ERROR
  - 0001: WARNING
  - 0002: RETRY
  - 0003: FATAL
  - 0004: SUCCESS
- The next string is the status message, if any.

Creating an Identity Vault User: Example Creation Policy

This style sheet can be used for a Creation policy. 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. To view the style sheet, see Create_User.xsl (../samples/Create_User.xsl)

```
<?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 Identity Manager so things
     are pretty to look at -->
<xsl:strip-space elements="*"/>
<xsl:preserve-space elements="value,component"/>
<xsl:output method="xml" indent="yes"/>

<!-- Identity Manager 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>
```