

Novell Data Synchronizer

Developer Guide

September 10, 2010

Overview

This document is meant for Data Synchronizer developers. It is designed to provide low-level information needed for advanced work on the Sync Engine and custom connectors..

Document Conventions

In this document, file name and command information in brackets represents information that must be replaced by the actual name.

Contents

Overview	2
Document Conventions	2
Contents	3
Getting Started	4
What Is Data Synchronizer?.....	4
How Does Data Synchronizer Work?.....	4
How Do I Install Sync Engine?.....	4
What If I Need a Completely New Connector?	5
Developing a New Connector	6
Overview.....	6
Steps for Development.....	6
Connector Visuals.....	7
Configuring a New Connector.....	8
Configuring a Connector Pipeline.....	10
Supporting Reference Materials	11
XML-to-HTML Template Relationship Examples.....	11
Example: User XML Configuration.....	11
Example: User Template HTML Using Text Boxes.....	11
Example: Connector XML Configuration.....	12
Example: Connector Template HTML Using Text Boxes.....	13
Additional Examples of HTML Input Fields.....	14
1. Check Boxes.....	14
2. Text Area.....	14
3. Radios.....	14
4. Selects.....	15
5. Multi-Select Boxes (HTML and XML).....	15
6. File Selector (Folder Treeview HTML and XML).....	16
Creating Policies with XSLT Function Calls.....	17
Name Spaces.....	17
Example: Connector Setting as Part of XSLT Policy.....	17
Example: Calling a Custom Connector Function.....	18
Engine Functions.....	18
Changing Engine Behavior With <custom> Data.....	19
Example: Connector Settings XML.....	19
XML Schema Information.....	20
Metadata Schema Terms.....	20
XML Schema Examples.....	20
Result Events Codes from AI to Engine.....	23
Attachments.....	24
Creating an Attachment.....	24
Retrieving an Attachment.....	24
Application Interface and SOAP Bridge APIs.....	25
Engine to Connector Network Protocol.....	28
Using SuperXML	29
Database Information.....	30
Suppression of Mirror Delivery Events	30
Performance Monitoring and Runtime Statistics.....	31
Open Source Software Used.....	32
Glossary of Terms	33

Getting Started

This document is meant for Data Synchronizer developers. It is designed to provide low-level information needed for advanced work on the Sync Engine and custom connectors.

The document provides three things:

- Getting Started overview
- Instructions on developing a new custom connector
- Configuration reference materials

To install Data Synchronizer, see the *Data Synchronizer Installation Guide* on the Novell Data Synchronizer Documentation Web site (<http://www.novell.com/documentation/datasynchronizer1>). The Synchronizer services must be installed before you can develop a new connector. For more information, see the *Data Synchronizer Administration Guide*.

What Is Data Synchronizer?

Data Synchronizer is a bidirectional synchronization technology that connects multiple enterprise applications for simultaneous synchronization of important information.

Two such applications are GroupWise and SugarCRM. Data Synchronizer already includes connectors for these two applications.

This means that if you have Data Synchronizer, a GroupWise connector, and a SugarCRM connector, GroupWise and SugarCRM can “talk” to one another. For instance, if you make an appointment in GroupWise, it appears in the connected SugarCRM system.

How Does Data Synchronizer Work?

Data Synchronizer is a hub-and-spoke design.

- **Hub:** The hub is made up of three core services: the Sync Engine, the Configuration Engine, and the Web Administration service.
- **Spoke:** The spokes consist of connectors attached to the Sync Engine at one end and an application at the other.

The Sync Engine houses a cache for data retention, and pushes and pulls events between connected applications.

Every connector is a code translator, ensuring that one application's code is converted into a form that another connected application can understand, and vice versa.

Each connector is made up of a pipeline of filters that monitor and organize events so that events that enter a connector on one end are understandable on the other end and conform to business policy.

After your initial installation and configuration, all Synchronizer system management is done by means of a simplified Web interface.

How Do I Install Data Synchronizer?

To install Data Synchronizer, see the *Data Synchronizer Installation Guide* on the Novell Data Synchronizer Documentation Web site (<http://www.novell.com/documentation/datasynchronizer1>).

Data Synchronizer installation includes the following steps:

1. **Install Engines from Media:** Download the source media. Decide if you want a single server install or a multi-server install. Configure the Sync Engine, the Configuration Engine, and Synchronizer Web Admin.

2. **Start Services on Servers:** The Synchronizer services must be started on your server(s).
3. **Create Running Connector(s):** Add connectors and users to prepare for synchronization.
4. **Configure Created Connector(s):** Configure each connector. Different applications have different connector requirements that must be addressed here.
5. **Run and Start Application Interface(s):** Now that you have engines on servers and have configured connectors to connect to engines, you must start the Application Interface(s) to enable synchronization of the system.
6. **Administer Connectors:** You have a running Synchronizer system. To set up your pipeline and make other adjustments to your filters and settings, use the Web interface.

The Data Synchronizer add-on product installer currently installs all services and connectors on a single server. Although it is currently unsupported, a developer might want to run any service or connector on a different server. To do a multi-server installation, refer to the sample XML template files in `/opt/novell/datasync/common/packaging`. The `install.sh` script can be used to set up an individual service on a server.

What If I Need a Completely New Connector?

Although several connectors exist for common important enterprise applications, entirely new connectors can be developed on a per-connector basis by means of the customization capability built into Data Synchronizer.

You have two basic options here:

- You can develop a new custom connector before your initial install and configure it along with any out-of-the-box connectors at the initial install.
- or
- You can develop it at a later time and then add it to your running Synchronizer system.

The section “Developing a New Connector” on page 6 explains how to do this.

Developing a New Connector

Overview

The following section explains how to develop a completely new custom connector based on a preexisting connector template.

Why do you need a connector?

Different applications are designed in different ways using different coding practices. This means that a connector is needed to facilitate synchronization between applications that you want to share information.

A Synchronizer connector is basically a translation tool that converts data from one application's code "dialect" to another. When data is tagged with a certain element at the originating connector, it must be translated into a form the endpoint connector can understand and display. This cannot occur without a connector to bridge the gap.

Before you can develop a new connector, you must have installed and properly configured the Synchronizer services (the Sync Engine, the Configuration Engine, and the Web Administration service).

You have two basic options here:

- You can develop a new custom connector before your initial install and configure it along with any out-of-the-box connectors at the initial install.
- or
- You can develop it at a later time and add it to your running system then.

Steps for Development

Developing a new connector can be broken down into three basic steps:

- Developing an Application Interface
- Developing connector and user settings
- Developing filters and policies

The Application Interface is made up of compiled code that talks to your connected system(s). A sending logic acts on the entering code so that data can become meaningful as it is sent through the connector to connected applications.

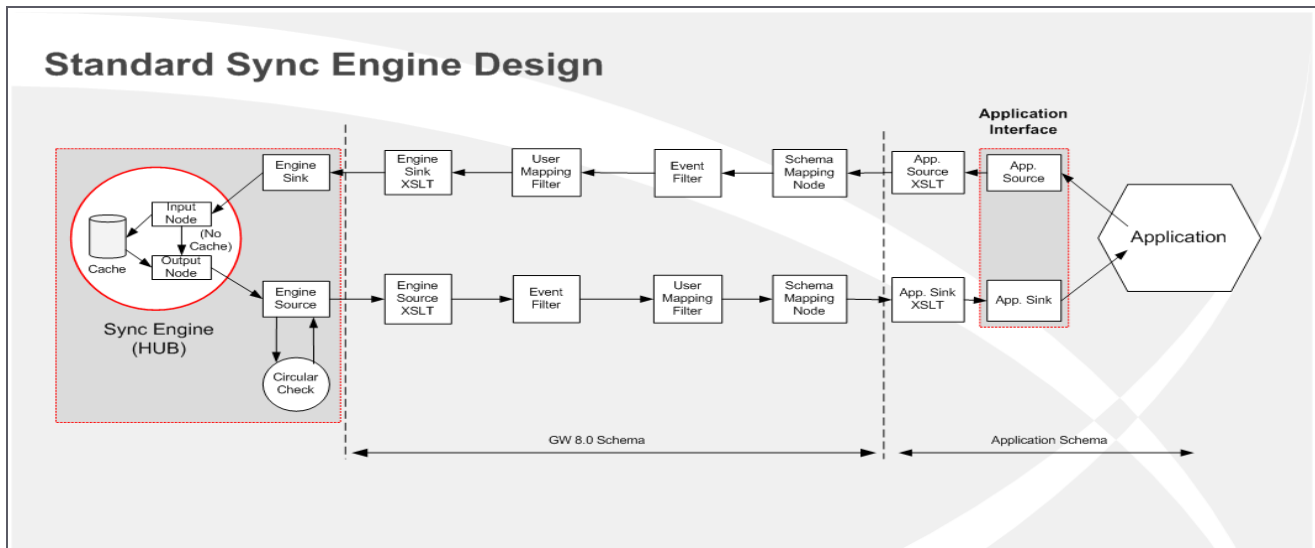
Data that is translated must have assigned destinations. Connector and user settings must be developed so that the data translated by XSLT makes it to its predetermined destination(s) according to the sending logic. Some of your new connector's settings are made in Synchronizer Web Admin, where it is named, pointed to the correct Sync Engine, and assigned a type.

When data is translated, has a set of logical destinations, and is actively running on the correct engine, you can assign filters. They create your connector pipeline's structure. The pipeline is the series of filter nodes used to manage and refine data entering the connector. Filter nodes provide a way to ensure that the connector filters data in the way your organization requires according to business protocol. You can use filter nodes to choose who gets what and at what time.

See page 7 for diagrams before you develop and configure a new connector. For instructions on configuring a new connector, see page 8.

Connector Diagrams

The following graphic is a visual representation of a Synchronizer system emphasizing the connector.



Configuring a New Connector

Use the following are steps to develop and configure a new connector.

1. Navigate to the `connectors` folder in the Data Synchronizer folder where you downloaded to your software directory. For example, `[directory installed to]/syncengine/connectors/`
2. Copy and rename the included example `connectors` folder. The folder is called `syncengine/connectors/example`. Doing this reproduces an example connector that you can use to build your new connector.
 1. To rename the folder, replace **example** in the folder name with your new connector name.
3. Navigate to the `factory` folder inside your newly created `connectors` folder.
4. Modify the default connector.xml (`syncengine/connectors/[example]/factory/connector.xml`) to define your new connector settings. The

Name	Size	Type	Date Modified
bin	3 items	folder	Thu Jun 18 15:17:04 2009
AppInterface.py	6.2 KB	Python script	Tue Jun 16 12:04:05 2009
startAI.sh	355 bytes	shell script	Tue Jun 16 12:04:05 2009
xsltab.py	1.8 KB	Python script	Tue Jun 16 12:04:04 2009
factory	5 items	folder	Thu Jun 18 16:35:02 2009
defaults	1 item	folder	Tue Jun 16 12:04:07 2009
user.xml	71 bytes	XML document	Tue Jun 16 12:04:07 2009
filters	2 items	folder	Tue Jun 16 12:03:29 2009
tpl	2 items	folder	Tue Jun 16 12:04:07 2009
connector.tpl	1.9 KB	XML document	Tue Jun 16 12:04:07 2009
user.tpl	700 bytes	plain text document	Tue Jun 16 12:04:05 2009
connector.xml	273 bytes	XML document	Tue Jun 16 12:04:04 2009
filterset.xml	2.2 KB	XML document	Tue Jun 16 12:04:04 2009
filter	0 items	folder	Tue Jun 16 12:03:28 2009
language	0 items	folder	Tue Jun 16 12:03:28 2009
lib	2 items	folder	Tue Jun 16 12:04:06 2009
demo.py	445 bytes	Python script	Tue Jun 16 12:04:05 2009
testapi.py	1.8 KB	Python script	Tue Jun 16 12:04:06 2009

Example Connector Directory and Files

connector.xml contains the name value pairs of the connector settings and specific connection information.

1. You must replace the `<type>` tag in the connector.xml with the name you gave your new connector in place of **example** above. For example, if you named your new connector `awesomeapp`, make sure the `<type>` tag is `awesomeapp`.
5. **Optional:** Your new application might require input fields not provided by default. The custom fields you require are entirely up to your organization, but they must be created now so that the new connector can use them. If you want to create custom configuration settings, do the following:
 1. Add the new tags and fields between the `<custom>` tags in the `syncengine/connectors/[example]/factory/connector.xml`.
 2. Assign your custom configuration settings templates for the new connector. This allows you to customize HTML files to visually present the custom settings XML you just created. (See “XML-to-HTML Template Relationship Examples” on page 11 for detailed examples.)
 1. Navigate to and open the custom configuration settings template file called `syncengine/connectors/[example]/factory/tpl/connector.tpl`
 2. Alter the HTML to suit your needs. You need an advanced knowledge of the Django template layer to do so. For more information, see <http://docs.djangoproject.com/en/dev/#the-template-layer>

The following brief example shows the correlation between the XML file and the TPL file:

1. XML:


```
<custom>
  <port>4500</port>
</custom>
```

2. TPL:

```
<input name="customsettings_port" class="customsetting_text" type="text" />
```

6. Modify the default user.xml (`syncengine/connectors/[example]/factory/defaults/user.xml`) to define your new user settings. The user.xml contains the name value pairs of the user settings and specific connection information.

7. Assign your user configuration settings templates for the connector. The user configuration settings template file is called `syncengine/connectors/[example]/factory/tpl/user.tpl`

The user configuration settings template allows you to customize HTML files to visually present custom settings XML.

8. **Optional:** Define and modify user settings so that users can, if required by business practice and policy, add user-specific information needed to become actively synchronized between certain connected systems.

This step can be bypassed if policies are implemented by administrators who automate a user's connectivity via a policy and/or filter. Filters are set up via the Web interface when configuring your pipeline.

9. Modify `syncengine/connectors/[example]/bin/AppInterface.py` to fit your desired connector logic. This step effectively enables your connector. The fields and settings you configured above become active only after you decide on and complete your Python script.

For examples of how this can look, look at other connectors' `AppInterface.py` files found in your connector files.

10. Restart all components (the Sync Engine, the Configuration engine, the Web Administration service, and your connectors). Restarting allows the connectors to engage with the engines.

When you have completed this step, your connector is alive and listening. It only needs to be named and directed to listen to the correct Sync Engine.

11. Create a connector instance at the Web Administration interface:

1. Log in to Synchronizer Web Admin.
2. Select *Add Connector* in the *Manage Connectors* field.
3. Fill in the connector settings:

1. **Connector Name:** Create a unique name for your connector.

This name is not necessarily the same as your connector type. For instance, you might have multiple connectors of the same type, thus requiring different names to differentiate them.

2. **Engine:** Select which engine the connector will operate with.
3. **Connector Type:** Select the proper connector type for your new connector.

4. Click *Add Connector*.

12. Finally, add users to each connector using the *Manage Users* tab in Synchronizer Web Admin.

Important: You must add users in order for events to synchronize.

At this stage, your new connector should be fully operational.

Configuring a Connector Pipeline

Now that the new connector is configured, the pipeline must be developed by configuring various nodes.

Synchronizer nodes allow a developer to implement specific business policies throughout a pipeline in a logic of their choosing. These nodes are written in XSLT to monitor and (re)direct data entering the pipeline. Pipeline configuration occurs at the application programming interface (API).

To configure the pipeline:

1. Carefully define the custom node plug-ins you've chosen to use in your pipeline.

The example custom node plug-ins file is called **syncengine/connectors/example/filter**.

2. Carefully define the order of the filters you've chosen to use in your default pipeline.

The example filter order file is called **syncengine/connectors/example/factory/filterset.xml**.

3. Assign the XML configurations for your default filters.

They are located in two folders (`sink` and `source`) in the **syncengine/connectors/example/factory/filters/** file.

Supporting Reference Materials

The following sections are less sequential than those in “Getting Started.” Refer to the material below according to topic and need.

XML-to-HTML Template Relationship Examples

The following instructions and examples show how to write custom settings templates for connectors. For more examples, see the GroupWise TPLs at [syncengine/connectors/groupwise/factory/tpl/](#).

In the following two example boxes, the name attribute in the HTML example corresponds with a matching tag in the XML configuration. To create an instance of an input field, you must specify its type by including a class attribute. For example, in the HTML below, the class attribute is set to `customsetting_text`, which indicates that its class type is a text box.

Important: When you add a new input field, make sure you create a new matching node in the XML source.

Example: User XML Configuration

```
<customsettings>
  <disableAlerts>true</disableAlerts>
  <defaultTitle>Hello, my name is Inigo Montoya. You killed my father. Prepare to
  die.</defaultTitle>
  <saying>It's a moo point. A cow's opinion. It just doesn't matter. It's  moo.</saying>
</customsettings>
```

Example: User Template HTML Using Text Boxes

```
<tr>
  <td class="label">disableAlerts:</td>
  <td class="element">
    <input name="customsetting_disableAlerts" class="customsetting_text"
    type="text"/>
  </td>
</tr>
<tr>
  <td class="label">defaultTitle:</td>
  <td class="element">
    <input name="customsetting_defaultTitle" class="customsetting_text"
    type="text"/>
  </td>
</tr>
<tr>
  <td class="label">saying:</td>
  <td class="element">
    <input name="customsetting_saying" class="customsetting_text"
    type="text"/>
  </td>
</tr>
```

Example: Connector XML Configuration

```
<connector>
  <settings>
    <common>
      < caching>enabled</ caching>
      < type>flatfile</ type>
      < startup>manual</ startup>
      < log>
        < file>groupSang.log</ file>
        < level>Debug</ level>
      </ log>
      < enabled>>true</ enabled>
    </ common>
    < custom>
      < inboundDirectory>/tmp/inbound/</ inboundDirectory>
      < outboundDirectory>/tmp/outbound/</ outboundDirectory>
      < pollRate>20</ pollRate>
    </ custom>
  </ settings>
</ connector>
```

During developing or debugging, it is often useful to see diagnostic logging, which shows everything that happens to an event and gives more detailed insight into the Sync Engine and connectors. To enable diagnostic verbose debug mode, set the log level to debug, enable verbose mode, and set the diagnostic XML tag by hand:

```
<log>
  <file>default.pipeline1.mcLocal.log</file>
  <failures>on</failures>
  <level>debug</level>
  <verbose>diagnostic</verbose>
</log>
```

Example: Connector Template HTML Using Text Boxes

```
<tr>
  <td class="label">Inbound Directory:</td>
  <td class="element">
    <input name="customsetting_inboundDirectory"
id="customsetting_inboundDirectory" class="customsetting_text"
    type="text"/>
  </td>
</tr>
<tr>
  <td class="label">Outbound Directory:</td>
  <td class="element">
    <input name="customsetting_outboundDirectory"
class="customsetting_text" type="text"/>
  </td>
</tr>
<tr>
  <td class="label">Poll Rate:</td>
  <td class="element">
    <input name="customsetting_pollRate" class="customsetting_text"
type="text"/>
  </td>
</tr>
```

Additional Examples of HTML Input Fields

1. Check Boxes

```
<tr>
  <td class="label">Hax:</td>
  <td class="element">
    <input name="customsetting_soapServer" checked="false"
class="customsetting_checkbox" type="checkbox"/>
  </td>
</tr>
```

2. Text Area

```
<tr>
  <td class="label">Trusted Application Name</td>
  <td class="element">
    <textarea name="customsetting_trustedAppName" style="width: 200px;"
class="customsetting_textarea" type="text"/></textarea>
  </td>
</tr>
```

3. Radio Buttons

```
<tr>
  <td class="label">Trusted Application Key:</td>
  <td class="element">
    <input type="radio" class="customsetting_radio"
name="customsetting_trustedAppKey" value="A1" />ASDF2384AZ
    <input type="radio" class="customsetting_radio"
name="customsetting_trustedAppKey" value="A2" checked="checked"
/>23SZEE4A2
    <input type="radio" class="customsetting_radio"
name="customsetting_trustedAppKey" value="A3" />2356AASS3ZA3
  </td>
</tr>
```

4. Select Boxes

```
<tr id="configureUsers">
  <td class="label">Configure Users:</td>
  <td class="element">
    <select id="customsetting_configureUsersOn"
name="customsetting_configureUsersOn" class="customsetting_select">
      <option value="startup">On connector start up</option>
      <option value="event">When an event is received from the engine
with their name</option>
    </select>
  </td>
</tr>
```

5. Multi-Select Boxes (HTML and XML)

HTML

```
<tr id="addEvents" class="advanced_options" style="display:none;">
  <td class="label">Add Events To Sync:</td>
  <td class="element">
    <select multiple="multiple" id="customsetting_addEvents"      name="customsetting_addEvents"
class="customsetting_msl"      style="width:300px;">
      <option value="AddressBookAdd">AddressBookAdd</option>
      <option value="AddressBookItemAdd">AddressBookItemAdd</option>
      <option value="FolderAdd">FolderAdd</option>
      <option value="FolderItemAdd">FolderItemAdd</option>
      <option value="ProxyAccessAdd">ProxyAccessAdd</option>
      <option value="PersonalGroupItemAdd">PersonalGroupItemAdd</option>
    </select>
  </td>
</tr>
```

XML

```
<addEvents>
  <PersonalGroupItemAdd>0</PersonalGroupItemAdd>
  <AddressBookAdd>0</AddressBookAdd>
  <AddressBookItemAdd>0</AddressBookItemAdd>
  <FolderAccept>0</FolderAccept>
  <FolderItemAdd>1</FolderItemAdd>
  <FolderAdd>0</FolderAdd>
  <ProxyAccessAdd>0</ProxyAccessAdd>
</addEvents>
```

6. File Selector (Folder Treeview HTML and XML)

To create a file/folder selector, create a div object with the class attribute `customsetting_fileselector`. You must also include an ID attribute that matches the name of the corresponding XML node.

The file/folder selector displays a collapsible treeview of the directory structure on the the Web server. To specify the root directory, you must include (`%testSelector,/opt/%`).

The first argument is the name of the treeview object (which matches the ID attribute in the div) and the second argument is the root of the treeview object.

Here's an example of the final syntax:

HTML

```
<!-- USER FileTree TPL Example Syntax --->
<tr>
  <td class="label">
    Please Select Root Directories:
  </td>
  <td class="element buttonrow">
    <div class="customsetting_fileselector" id="testSelector">(%
testSelector, /opt/ %)</div>
  </td>
</tr>
<tr>
  <td class="label">
    Please Select Home Directories:
  </td>
  <td class="element buttonrow">
    <div class="customsetting_fileselector" id="fileSelector">(%
fileSelector, / root/devel %)</div>
  </td>
</tr>
```

XML

```
<testSelector>
  <value>/opt/gnome</value>
  <value>/opt/kde3/share/applications</value>
</testSelector>
```


Creating Policies with XSLT Function Calls

XSLT function calls allow a connector developer greater power and flexibility in designing policies. Developers can write functions in Python and expose the functions to their XSLT policies.

Name Spaces

Import the following name space for functions exposed by the engine:

<http://www.novell.com/nsxml/gwsync/com.novell.gwsync.syncengine.xslt.api>.

To import functions you define for your own connector, add the Python file to the

`/opt/gwsync/syncengine/connectors/[your connector]/lib` folder and import it to

[http://www.novell.com/nsxml/gwsync/com.novell.gwsync.connectors.\[your connector\].\[lib filename\]](http://www.novell.com/nsxml/gwsync/com.novell.gwsync.connectors.[your connector].[lib filename])

Example: Connector Setting as Part of an XSLT Policy

This example shows the use of a connector setting as part of an XSLT policy. The policy calls an engine-exposed function named `getConnectorSettings()` and saves the resulting nodeset as a variable. The policy then uses XPath to pull out the connector's caching setting and inserts it into the resulting XML document.

```
<!-- Settings examples -->
<?xml version='1.0' encoding='utf-8'?>
<xsl:stylesheet version='1.0'
xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
xmlns:gwsync='http://www.novell.com/nsxml/gwsync/com.novell.gwsync.syncengine.xslt
.api'
exclude-result-prefixes = "gwsync demo">
  <xsl:template match="node()|@*">
    <!-- Get the connector settings -->
    <xsl:variable name="connectorSettings"      select="gwsync:getConnectorSettings()"/>
    <!-- Grab a single value -->
    < caching><xsl:value-of
      select="$connectorSettings/settings/common/caching"/></ caching>
    </xsl:template>
</xsl:stylesheet>
```

Example: Calling a Custom Connector Function

This example shows how to call a custom connector function from XSLT. Python code is in `/opt/gwsync/syncengine/connectors/example/lib/demo.py`.

Note: The class must be named the same as the filename (with the first letter uppercase). Each function inside the class must take `self` and `context` as the first arguments.

```
class Demo:
    def myConnectorFunc(self, context):
        return "You called a connector function!"
```

The following XSLT will call the above Python function:

```
<?xml version='1.0' encoding='utf-8'?>
<xsl:stylesheet version='1.0'
xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
xmlns:demo='http://www.novell.com/nsxml/gwsync/com.novell.gwsync.connecto
rs.example.demo'
exclude-result-prefixes = "demo">
  <xsl:template match="node()|@*">
    <!-- Get NASA's twitter -->
    <result><xsl:value-of select="demo:myConnectorFunc()"/></result>
  </xsl:template>
</xsl:stylesheet>
```

Engine Functions

The following list of engine functions is exposed via <http://www.novell.com/nsxml/gwsync/com.novell.gwsync.syncengine.xslt.api>:

- **logMessage(message, level='warning')**: Sends a message to the connector's log.
- **getSettingsForTarget(targetDn)**: Returns an XML nodeset of the settings for the requested target. To select a single value, use XPath on the resulting nodeset.
- **getConnectorSettings()**: Returns an XML nodeset of the settings for the connector. To select a single value, use XPath on the resulting nodeset.
- **getEngineSettings()**: Returns an XML nodeset of the settings for the requested target. To select a single value, use XPath on the resulting nodeset.
- **getConfigEngineSettings()**: Returns an XML nodeset of the settings for the Configuration Engine. To select a single value, use XPath on the resulting nodeset.
- **dnExists(dn)**: Tests for existence of the given DN in the LDAP directory (if one is configured and accessible) and returns the result as a boolean.
- **isTargetEnabled(dn)**: Tests for existence of the given DN in the connector's target table and returns the result as a Boolean.
- **getMatchingUserDN(sourceName)**: Given a cn, search all configured user containers in LDAP and return the DN of the matching user if there is exactly one match.
- **base64Encode(message)**: Returns the message Base64-encoded.
- **base64Decode(encodedMessage, returnAsNodeset=False)**: Returns the Base64-encoded message as unencoded text.

Changing Engine Behavior With <custom> Data

stripCustomTags is a connector setting that changes the type of <custom> tags that are included in an event coming from connected applications into a given connector. It does not have a field in Synchronizer Web Admin interface and must be entered by hand in the Edit XML Source page from the connector settings.

The valid settings for <stripCustomTags> follow:

- **always:** Strips the entire <custom> tag before sending the event through to filters.
- **never:** Keeps the entire <custom> tag intact and might contain sub-tags for multiple connectors other than the connector receiving the event.
- **others:** Keeps the <custom> tag in the event, but removes sub-tags for custom data that doesn't belong to the connector receiving the event.

Example: Connector Settings XML

```
<connector>
  <settings>
    <common>
      < caching>enabled</ caching>
      < type>flatfile</ type>
      < enabled>true</ enabled>
      < log>
        < file>ff2.log</ file>
        < level>Debug</ level>
      </ log>
      < stripCustomTags>others</ stripCustomTags>
    </ common>
    < custom>
      < inboundDirectory>/tmp/inbound2/</ inboundDirectory>
      < outboundDirectory>/tmp/outbound2/</ outboundDirectory>
      < pollRate>20</ pollRate>
    </ custom>
  </ settings>
</ connector>
```

XML Schema Information

Metadata Schema Terms

The following are key metadata schema terms. Terms with an * are fields the Sync Engine adds. Terms in () are items that should only be visible to the Sync Engine.

- **destination**: Fully qualified connector name that the event is going to. It is valid only when type = query.
- **SourceName**: The name of the object (user, group, etc.) that the event occurs for.
- **sourceType**: Examples include user and group.
- **type**: Examples are add, modify, query, delete, sync, and status.
- **OriginType**:
- **redeliveryAttempts**: If an event isn't fully processed, a background thread periodically tries to redeliver it in case the problem was temporary. Each time it is redelivered, the tag is incremented. A connector developer could potentially set an XSLT policy that drops the event after a given number of failed attempts.
- ***origin**: Fully qualified connector name.
- ***timestamp**: Time that the Sync Engine first sees an event.
- ***id**: Globally unique event ID.
- ***sourceDN**: LDAP distinguished name (DN) matched to sourceName.
- ***objectID**: Source application ID for the item.
- ***origSourceName**:
- ***(cacheID)**:
- ***(creationEventID)**:

XML Schema Examples

The following examples of XML schema metadata represent what is currently live and enforced by the Sync Engine.

1. Sample doc:

```
<?xml version="1.0"?>
<event>
  <metadata>
    <key/value pairs>
  </metadata>
  <item>
    <arbitrary xml or string data/>
  </item>
</event>
```

2. The following metadata fields are automatically created by the engine:

```
<origin>default.pipeline1.activesync1</origin>
<timestamp>Mon, 18 May 2009 15:13:06 PDT</timestamp>
<id>default.pipeline1.activesync1.sourcesoapbridge.bc2038507a868b5b4563f998584 a706b</id>
```

3. The following metadata fields must be supplied by the Application Interface when it sends out normal events. All other metadata tags supplied by the Application Interface are discarded.

```
<type>add</type>
<sourceName>user1</sourceName>
  <sourceType>user</sourceType>
```

4. For direct query calls, a non-standard "query" event is created. Origin, timestamp, and ID are created as usual. Source name and source type are not. The **<type>** tag is set to "query" and the engine also automatically creates a "destination" tag. When sending a direct query to another connector, the **<item>** data contains arbitrary XML that makes sense to the receiving connector. It is up to the sender to construct queries that the other end understands. For example:

```
<type>query</type>
  <destination>default.pipeline1.groupwise2</destination>
```

5. Inside a query's item tag, there should be a **<query>** sub-tag that has the actions and parameters necessary to complete that action or request. The receiving application performs that request or action and fills out a **<result>** tag inside the **<item>** tag, as follows:

Request:

```
<event>
  <metadata>
    <type>query</type>
    <destination>default.pipeline1.grpws</destination>
  </metadata>
  <item>
    <query>
      <!-- Name of function to call -->
      <action>getAddressBookListRequest</action>
      <!-- Name of user to login as -->
      <directQueryUsername>admin</directQueryUsername>
      <!-- Parameters necessitated by function -->
      <name>Frequent Contacts</name>
    </query>
  </item>
</event>
```

Result:

```
<event version="1">
  <metadata>
    <type>query</type>
    <destination>default.pipeline1.grpws</destination>
    <origin>default.pipeline1.ff</origin>
    <timestamp>Mon, 08 Jun 2009 23:53:49 PDT</timestamp>
    <id>default.pipeline1.ff-source-
      soapbridge.f7fada06a45168a3c1490c2d47ad8e62</id>
  </metadata>
  <item>
    <query>
      <name>Frequent Contacts</name>
    </query>
    <result>
      <gwm:getAddressBookListResponse
        xmlns:gwm="http://schemas.novell.com/2005/01/GroupWise/methods"
        xmlns:gwt="http://schemas.novell.com/2005/01/GroupWise/types">
        <gwm:books>
          <gwt:book>
            <gwt:id>49FAD119.gwdom.gwpo.104.1306B66.1.3.1@53</gwt:id>
            <gwt:name>Frequent Contacts</gwt:name>
            <gwt:version>3</gwt:version>
            <gwt:modified>2009-05-01T17:38:17Z</gwt:modified>
            <gwt:isPersonal>1</gwt:isPersonal>
            <gwt:isFrequentContacts>1</gwt:isFrequentContacts>
          </gwt:book>
          <gwt:book>
            <gwt:id>GroupWiseSystemAddressBook@52</gwt:id>
            <gwt:name>Novell GroupWise Address Book</gwt:name>
          </gwt:book>
          <gwt:book>
            <gwt:id>49FAD119.gwdom.gwpo.104.1306B66.1.1.1@53</gwt:id>
            <gwt:name>admin</gwt:name>
            <gwt:version>3</gwt:version>
            <gwt:modified>2009-05-01T17:38:17Z</gwt:modified>
            <gwt:isPersonal>1</gwt:isPersonal>
          </gwt:book>
        </gwm:books>
        <gwm:status>
          <gwt:code>0</gwt:code>
        </gwm:status>
      </gwm:getAddressBookListResponse>
    </result>
  </item>
</event>
```

Result Events Codes from Application Interface to Engine

Note: There are currently only two options: Success or Failed.

```
<event>
  <metadata>
    <!-- This is a copy of the original event's metadata, with the exception of
which is now Status-->
    <type>Status</type>
    <origin>default.pipeline1.activesync1</origin>
    <timestamp>Mon, 18 May 2009 15:13:06 PDT</timestamp>
    <id>default.pipeline1.activesync1.sourcesoapbridge.bc2038507a868b5b4563f9985
84a706b</id>
    <sourceName>user1</sourceName>
    <sourceType>user</sourceType>
    <objectID>49F0C5B3.gwdom.gwpo.100.1743575.1.15E.1@3:A.gwdom.gwpo.100.0.1.0.1
@19</objectID>
  </metadata>
  <item>
    <status>Success</status>
    <statusCode>1</statusCode> <!-- 0 or 1 -->
    <statusDesc>The item was added successfully.</statusDesc>
    <!-- The event's original type is copied here -->
    <type>add</type>
    <!--objectID is copied from the event, if available. -->
    <objectID>49F0C5B3.gwdom.gwpo.100.1743575.1.15E.1@3:A.gwdom.gwpo.100.0.1.0.1
@19</objectID>
  </item>
</event>
```

Attachments

Attachment data is not included directly in the item XML for an event. Because attachments can be large, they aren't sent over a SOAP connection. Instead, the Application Interface opens a new connection to the Sync Engine (still on the same port) and downloads/uploads the file directly without SOAP marshaling overhead. The Sync Engine does not enforce the attachment size. If the connected application has a size limitation, it needs to be enforced by the connector.

Creating an Attachment

When a connector encounters an attachment in an item, it should call **createNewAttachment(filename)**, which returns an attachment ID. In the item XML, it should substitute the attachment XML tags with the **<attachment id="123">** placeholder where 123 is the number from **createNewAttachment(filename)**

Note: When you call **createNewAttachment(filename)**, the filename argument is optional. It is used in case the application wants to present it to the user. You can keep whatever **<attachment>** XML you already have and just add the **id** attribute and remove any inline file data if it's currently being included.

The actual file data is sent to the Sync Engine separately. Because this can occur asynchronously, you should send the event XML before starting the attachment file upload.

To upload, you have three options:

- Use **uploadAttachmentFilename(pathToFile, attachmentID)** to upload a file already stored on disk.
- Upload an already-open (open for reading) Python file object. Use **uploadAttachmentFileObj(fileObject, attachmentID, filesize)** to tell it the file size to be copied.
- Use **uploadAttachmentData(rawDataString, attachmentID)** to upload data already stored in memory as a string or bytes.

Retrieving an Attachment

When an Application Interface sees something like **<attachment id="123">** in the item XML coming from another connector, it retrieves the attachment contents in one of three ways:

- Use **downloadAttachmentFilename(pathToFile, attachmentID)** to download a file already stored on disk. The return value is a success flag (True/False).
- Use **downloadAttachmentFileObj(fileObject, attachmentID, filesize)** to download an already-open python file object (open for writing). You must tell it the filesize to be copied. The return value is a success flag (True/False).
- Use **downloadAttachmentData(rawDataString, attachmentID)** to download data and store it in memory as a string of bytes. The return value is the data (or the None object, if there was an error).

Note: This is generally a bad option, because the whole attachment is stored in memory and there is no way to impose a size limit.

Application Interface and SOAP Bridge APIs

The following section lists all API functions available to the GenericApplicationInterface Python class that can be used from the AppInterface.py logic. The GenericApplicationInterface is an easy-to-use wrapper around the underlying network protocol (SOAP Bridge) between the Application Interface and the Sync Engine. The protocol is based on SSL, SOAP, and supersocket .

Generic Application Interface

(as of 2010-07-13 unstable rev 1278)

```
def sigTERM(signalNum, frame)
def sigUSR1(self, signalNum, frame)
def __del__(self)
def createLogger(self)
def kill(self)
def internalStartup(self, configXML)
def internalShutdown(self)
def testStartup(self, config)
def testShutdown(self)
def startEngineListener(self)
def startEngineClient(self)
def uploadAttachmentFileObj(self, fileObj, attachmentID, eventResult, filesize, md5hash = None, sha1hash = None, applicationHash = None)
def uploadAttachmentFilename(self, filename, *args, **kwargs)
def uploadAttachmentData(self, rawData, *args, **kwargs)
def downloadAttachmentFileObj(self, fileObj, attachmentID)
def downloadAttachmentFilename(self, filename, *args, **kwargs)
def downloadAttachmentData(self, *args, **kwargs)
def sendToEngine(self, event, processDepth = 1)
def sendToEngineStartNode(self, event)
def sendToEngineCache(self, event)
def sendToEngineAndConnectors(self, event)
def sendXMLToEngine(self, xmlEvent, processDepth = 1)
def sendXMLToEngineStartNode(self, xmlEvent)
def sendXMLToEngineCache(self, xmlEvent)
def sendXMLToEngineAndConnectors(self, xmlEvent)
def sendDirectQuery(self, destinationConnectorID, event)
def createNewAttachment(self, filename)
def getAttachmentInfo(self, attachmentID)
def eventSuccess(self, eventResult)
def eventDelivered(self, eventResult)
def eventDropped(self, eventResult)
def eventDeferred(self, eventResult)
def eventError(self, eventResult)
def eventUnknownError(self, eventResult)
def eventWantsAdd(self, eventResult)
def eventStatusConfirmation(self, eventResult)
def eventStatusSuccess(self, eventResult)
def eventStatusFailure(self, eventResult)
def makeStatus(self, event, objectID = None, status = 'Success', statusCode = None, statusDesc = None)
def sendStatus(self, *args, **kwargs)
def submitFolderList(self, targetDN, folderXML)
def submitFolderListByTargetName(self, targetName, folderXML)
def requestFolderList(self, targetDN = None, connectorID = None)
def authenticate(self, username, password)
```

```

def getTargetByDN(self, targetDN)
def getTargetByMatchingName(self, targetName)
def stringMapToDict(self, stringMap)
def setTargetName(self, targetDN, targetName, targetType)
def getConnectorObjectID(self, creationEventID, connectorID)
def translateConnectorObjectID(self, srcObjectID, destConnectorID)
def translateConnectorFolderID(self, srcTargetName, srcFolderID, destConnectorID)
def translateConnectorTargetName(self, srcTargetName, srcTargetType, destConnectorID)
def remapObjectID(self, creationEventID, newObjectID)
def getObjectSourceName(self, objectID)
def getConfigEngineConfig(self)
def getConnectorConfig(self)
def startup(self, config)
def shutdown(self)
def engineEventReceived(self, event, processDepth)
def directQueryReceived(self, destinationConnectorID, event)
def logException(self)
def setLogLevel(self, logLevelString)
def setLogVerbosity(self, verbosityLevelString)
def setLogFailures(self, failureModeString)
def incomingEventConsumer(self)
def getEngineConnection(self)

```

SOAP Bridge Functions

```

@soapmethod(soap_types.Any, _returns=soap_types.String)
def startConnector(self, connectorConfig)

@soapmethod(_returns=soap_types.String)
def stopConnector(self)

@soapmethod(soap_types.String, soap_types.Integer, _returns = soap_types.String)
def processEvent(self, eventXML, processDepth)

@soapmethod(soap_types.String, _returns=soap_types.Map(soap_types.String))
def getTargetByMatchingName(self, targetName)

@soapmethod(soap_types.String, _returns=soap_types.Map(soap_types.String))
def getTargetByDN(self, targetDn)

@soapmethod(soap_types.String, soap_types.String, soap_types.String, _returns = soap_types.String)
def setTargetName(self, targetDn, targetName, targetType)

@soapmethod(soap_types.String, soap_types.Array(soap_types.String), soap_types.String,
_returns=soap_types.Array(LDAPSearchResult))
def ldapSearch(self, filterString, attrs, searchBaseType)

@soapmethod(soap_types.String, soap_types.String, _returns=soap_types.String)
def authenticate(self, username, password)

@soapmethod(_returns=soap_types.Array(syncengine.config.incoming.Target))
def getTargets(self)

@soapmethod(soap_types.String, soap_types.String, _returns=soap_types.String)
def getConnectorObjectID(self, creationEventID, connectorID)

```

```

@soapmethod(soap_types.String, soap_types.String, _returns=soap_types.String)
def translateConnectorObjectID(self, srcObjectID, destConnectorID)

@soapmethod(soap_types.String, soap_types.String, _returns=soap_types.String)
def remapObjectID(self, creationEventID, newObjectID)

@soapmethod(soap_types.String, soap_types.String, soap_types.String, _returns=soap_types.String)
def translateConnectorFolderID(self, srcTargetName, srcFolderID, destConnectorID)

@soapmethod(soap_types.String, soap_types.String, soap_types.String, _returns=soap_types.String)
def translateConnectorTargetName(self, srcTargetName, srcTargetType, destConnectorID)

@soapmethod(soap_types.String, _returns=soap_types.String)
def getObjectSourceName(self, objectID)

@soapmethod(soap_types.String, soap_types.String, _returns=soap_types.String)
def processQuery(self, destConnectorID, queryXML)

@soapmethod(_returns=soap_types.Any)
def getInfo(self)

@soapmethod(soap_types.String, _returns=soap_types.String)
def createNewAttachment(self, filename)

@soapmethod(soap_types.String, _returns=soap_types.String)
def getAttachmentInfo(self, attachmentID)

@soapmethod(soap_types.String, soap_types.String, _returns = soap_types.String)
def submitFolderList(self, targetDN, folderXML)

@soapmethod(soap_types.String, soap_types.String, _returns = soap_types.String)
def submitFolderListByTargetName(self, targetName, folderXML)

@soapmethod(soap_types.String, soap_types.String, _returns =
soap_types.Map(soap_types.Array(soap_types.Map(soap_types.String))))
def requestFolderListByConnector(self, targetDN, connectorID)

@soapmethod(_returns = soap_types.String)
def getConfigEngineConfig(self)

@soapmethod(_returns = soap_types.String)
def getConnectorConfig(self)

```

Engine to Connector Network Protocol

The way sockets are handled does not follow the normal client/server model. There is one SOAP connection for the connector sending events to the Sync Engine, and one SOAP connection for the Sync Engine pushing events to the connector. Normally the connector opens a connection to the Sync Engine for one direction, and the Sync Engine opens a connection to the connector for the other direction. However, one of the design requirements was that the connector uses a single port and can work behind a firewall, meaning that the connector can always initiate a connection to the Sync Engine, but the Sync Engine can't necessarily open a connection to the connector (the connector might be behind NAT and isn't directly addressable).

To avoid this problem, the connector opens two socket connections to the Sync Engine on startup. The Sync Engine treats the first connection as a standard SOAP connection with the Sync Engine as the SOAP server and the connector as the SOAP client. The second connection reverses the roles: even though the connector initiated the socket, the Sync Engine is the SOAP client and the connector is the SOAP server. This model is compliant with TCP and is technically possible to replicate in any language; however it can be tricky because it is not very common. After the sockets are set up, everything else follows standard practices. At that point, there are two persistent HTTP connections each using SOAP over SSL version 3 (TLS).

Immediately after the socket is connected, the following data is sent to the Sync Engine:

integer: version number of protocol (currently 1)
string length 10, null padded: purpose of socket ("normal" or "reversed")
string length 256, null padded: connector ID in the form "default.pipeline1.connectorName"

If the header is valid, the Sync Engine returns an integer: 1 for success, 0 for failure. After that point the socket follows standard HTTP SOAP communication.

For more information on how that binary data is packed and unpacked, see <http://docs.python.org/library/struct.html>

Authentication:

During installation, a DataSync SSL certificate authority is created and issues certificates for each service. Any SOAP connection to the engines requires client-side SSL certificates in addition to server-side SSL to provide authentication and prevent unauthorized requests. When a SOAP client connects to one of the SOAP sockets, it needs to supply an SSL certificate with a valid trust chain or else the connection is dropped.

The certificate authority scripts are located at `/opt/novell/datasync/common/bin/ssl`

The CA issued certificates are maintained at `/var/lib/datasync/common/CA`

The trust chain used for authentication is stored in `/var/lib/datasync/common/CA/trustedroot.pem`

Any certificate trusted by an issuer in this file is trusted for communication into the engine.

PEM files for services are stored in the following files:

```
/var/lib/datasync/configengine/soapserver.pem  
/var/lib/datasync/syncengine/connectors.pem  
/var/lib/datasync/syncengine/remoteManagement.pem  
/var/lib/datasync/webadmin/server.pem
```

Using SuperXML

The following examples are for SuperXML (Python XML library). The SuperXML library is designed to help developers more easily write XML.

```
xml = superxml.fromString(someString)

xml = superxml.fromDict(someDict)

xml.writeFile('/tmp/blah.xml')

print xml.getString() # pretty is the default
print xml.getString(pretty = True)
print xml.getString(pretty = False)

xml = superxml.fromString('<foo><a>asdf</a></foo>')
xml.root.a

someDict = xml.getDict()
```

For more API documentation, go to <http://codespeak.net/lxml/objectify.html>.

Database Information

The PostgreSQL database is named `datasync`. In the current Development VM, it requires both a username and a password.

- To connect, run `psql -U datasync_user datasync`
- The following query shows you what is in the `gwsync` database: `\dt`
- The following command describes the schema for a given table: `\d cache`
- The following command shows how many items are in the cache: `select count(*) from cache;`
- The following command lists all the events in the cache, excluding the raw event XML, which will usually fill the screen: `select id,"eventTimestamp",origin,"eventID","sourceName","sourceType" from cache;`
- The following command lists only the raw event XML from the cache: `select data from cache;`
- The following command shows which cache items still need to be delivered and which connectors have already processed them: `select * from retention;`


Suppressing Mirror Delivery Events

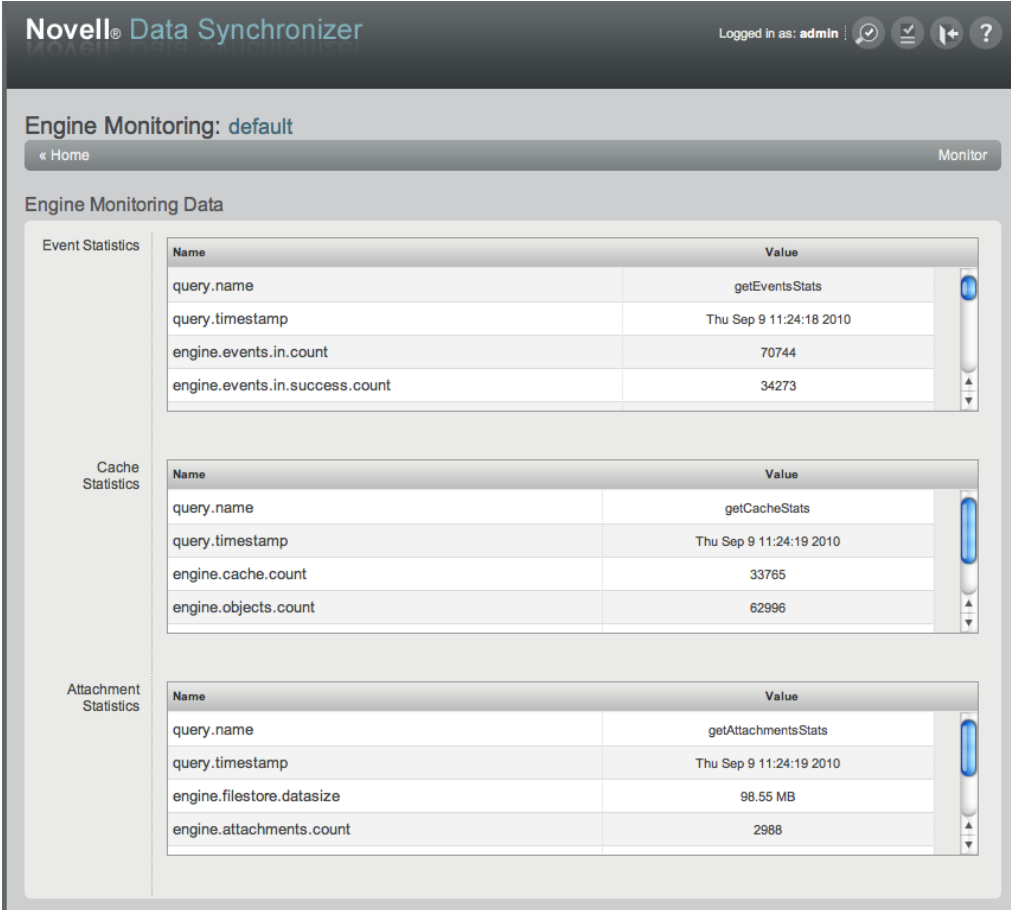
Connectors are responsible for suppressing the delivery of events that are mirror images of an event coming from the Sync Engine to the Application Interface.

For instance, GroupWise keeps track of the recently seen Sync Engine events and their object IDs. When a GW slap notification occurs, you can tell if the notification is for a truly new event or if it's the add/modify/delete just performed.

Other connectors can handle this differently. For instance, SugarCRM uses a service account to do its work. If it sees that an event was triggered by that service account, it knows to ignore it.

Performance Monitoring and Runtime Statistics

You can view the current state of the Sync Engine and certain connectors by using the Monitoring button: 



The screenshot shows the 'Novell Data Synchronizer' interface. At the top, it says 'Logged in as: admin'. The main heading is 'Engine Monitoring: default'. Below this, there are three sections of statistics, each with a table of 'Name' and 'Value'.

Event Statistics

Name	Value
query.name	getEventsStats
query.timestamp	Thu Sep 9 11:24:18 2010
engine.events.in.count	70744
engine.events.in.success.count	34273

Cache Statistics

Name	Value
query.name	getCacheStats
query.timestamp	Thu Sep 9 11:24:19 2010
engine.cache.count	33765
engine.objects.count	62996

Attachment Statistics

Name	Value
query.name	getAttachmentsStats
query.timestamp	Thu Sep 9 11:24:19 2010
engine.filestore.datasize	98.55 MB
engine.attachments.count	2988

Information on specific statistics and their meaning is available at:
http://www.novell.com/documentation/datasynchronizer1/datasync1_admin/data/bprzfyh.html

Open Source Software Used

The following open source software is used in various parts of the Synchronizer system:

Project Name	Web site	Open Source License
soaplib	http://trac.optio.webfactional.com/	LGPL
jQuery	http://jquery.com	MIT or GPL
tablesorter (jQuery plug-in)	http://tablesorter.com/docs/	MIT or GPL
Impromptu	http://trentrichardson.com	MIT or GPL
uiTableFilter (jQuery plug-in)	http://gregweber.info/projects/uitablefilter	MIT or GPL
django	http://www.djangoproject.com/	BSD
suds	https://fedorahosted.org/suds/	LGPL
NetworkX	http://network.lanl.gov/	LGPL
CodePress	http://codepress.sourceforge.net/index.php	LGPL
pysqlpool	http://code.google.com/p/pysqlpool/	GNU or LGPL
CherryPy	http://www.cherrypy.org	BSD
pytz	http://pytz.sourceforge.net/	MIT

Glossary of Terms

The following is a glossary of commonly used Synchronizer terms and their definitions.

Application: Usually one of these enterprise software packages: GroupWise, SugarCRM, Salesforce, and SharePoint. Can also apply to future connected applications as well.

Application Interface: The point at which the Sync Engine spoke interfaces with the linked application. (*Not to be confused with the API or Synchronizer Web Admin.)

Circular Check Node: Stemming from the Engine Source Node, this node makes sure that event data does not return to its source application. If it perceives event data that is en route to the source application, it dumps the event from that spoke so that an infinite loop isn't created.

Connector: The series of nodes that filter, sort, push, and pull event data between the Sync Engine engine and the connected application. Every connector has two interfaces: an engine interface and an Application Interface. Connectors are provided for GroupWise, SugarCRM, Salesforce, and SharePoint.

Engine/Application Sink XSLT: This filter is the last chance to alter the XML event data code before it moves into either the Sync Engine engine or the connected application, depending on its source.

Engine/Application Source XSLT: This filter is the first chance to alter the XML event data code before it moves away from its source.

Filter: A node that applies certain rules to event data, helping to cull and focus the event data on its way to or from the Sync Engine.

Logging Node: Logs the progress of event data as it travels between the Sync Engine and the connected application. There is one hidden Logging Node created between every non-Logging Node.

Low-Level/Hidden Nodes: *See Logging Node above.*

Mapping Filter: Changes the XML schema coming from the Match Filter to a schema that makes sense to the connected application.

Match Filter: Matches an identity on one side of the system to an identity on another side, based on a rule so that events are routed to the correct identity. The filter could match user names, e-mail addresses, or other identifiers chosen by the customer and/or the customer's business logic and needs.

Node: A discrete step with a defined task in a pipeline. It is one instance of a plug-in. You can think of a plug-in as a class and a node as an object of that class.

Node Container: Looks like a node, but it contains other nodes. It is a mini-pipeline that acts as a black box in the main pipeline. Its purpose is to hide complexity and isolate internal system functions.

Pipeline: From a back-end perspective, the pipeline is a process of movement through nodes, becoming more defined by means of filters, and then routing to various appropriate connected applications. The terms pipeline and task are synonymous in the context of the Synchronizer system. However, the term pipeline is more appropriate for a Sync Engine developer.

Plug-in: Software that implements a specific source, sink, or filter. For this project, plug-ins are defined in the following form: connectorName.pluginType.pluginName. The plug-in name is optional; it applies only if a connector has multiple plug-ins of the given type.

Rate Limiting/Throttling Node: Slows the event sending and receiving process in a case where either the Sync Engine or the connected application pushes event data too fast for the other to handle. It does so by providing a temporary cache to hold data long enough to slow the system.

Remote Transport Nodes: Sister nodes that reside on each side of the cloud in a remote spoke design. One pair of nodes resides on the Sync Engine side of the network/Internet and one pair resides on the connected application side. Each pair is made up of a Receive Node and a Send Node.

Routing Connector: An engine that ties together multiple other Sync Engines.

Schema: The pattern by which XML code is written to communicate in an application. The Sync Engine project is based on the GroupWise 8.0 schema.

Sharding: An action that breaks up groups of data into smaller logical groups to minimize the amount of data traffic traveling between the Sync Engine and the connected application.

Source Node: The first node in a pipeline. In the context of the pipeline, it is the source of the event.

Sink Node: The last node in a pipeline; it is the end of the pipeline.

Sync Engine: The hub in a hub-and-spoke system. It sends and receives event data to and from a connected application to other connected applications. It is made up of an Input Node, an Output Node, and a Cache Database. The Input Node can bypass the Cache, if told to do so, and send event data directly to the Output Node.

Task: In this context, a repeatable activation of a given pipeline. The task waits for incoming events to act on.

The terms pipeline and task are synonymous in the context of the Synchronizer system. However, the term task is more appropriate for a user at the Application Interface level.