EVIDEN

Identity and Access Management

DirX Identity

Java Programming in DirX Identity

Version 8.10.14, Edition March 2026

All product names quoted are trademarks or registered trademarks of the manufacturers concerned.

© 2026 Eviden

All Rights Reserved

Distribution and reproduction not permitted without the consent of Eviden.

Table of Contents

Preface

This document describes a set of use cases that explain how to use specific features of DirX Identity. It helps users to model their use case with DirX Identity and to set up and run their DirX Identity system.

This document explains how to program Java extensions for DirX Identity. It consists of the following chapters.

- Chapter 1 provides an overview of the described use cases.
- Chapter 2 explains how to work with the programming environment.
- Chapter 3 explains how to extend real-time workflows.
- Chapter 4 describes how to create Java-based extensions for password synchronization workflows.
- Chapter 5 describes how to implement and manage user hooks for event-based processing workflows.
- Chapter 6 describes how to extend request workflows.
- Chapter 7 provides information about web service extensions.
- Chapter 8 describes how to implement consistency rules.
- Chapter 9 describes how to implement a custom connector.
- Chapter 10 describes how to implement a RESTful connector.
- Chapter 11 describes how to use the User LDAP lock feature in Java classes and Java scripts.

DirX Identity Documentation Set

*Version 8.10.14 | Build 1858 | Date 2026-03-26 *

The DirX Identity document set consists of the following manuals:

- *DirX Identity Introduction*. Use this book to obtain a description of DirX Identity architecture and components.
- *DirX Identity Release Notes*. Use this book to understand the features and limitations of the current release. This document is shipped with the DirX Identity installation as the file **release-notes.pdf**.
- *DirX Identity History of Changes*. Use this book to understand the features of previous releases. This document is shipped with the DirX Identity installation as the file **history-of-changes.pdf**.
- *DirX Identity Tutorial*. Use this book to get familiar quickly with your DirX Identity installation.
- *DirX Identity Provisioning Administration Guide*. Use this book to obtain a description of DirX Identity provisioning architecture and components and to understand the basic tasks of DirX Identity provisioning administration using DirX Identity Manager.
- *DirX Identity Connectivity Administration Guide*. Use this book to obtain a description of DirX Identity connectivity architecture and components and to understand the basic tasks of DirX Identity connectivity administration using DirX Identity Manager.
- *DirX Identity User Interfaces Guide*. Use this book to obtain a description of the user interfaces provided with DirX Identity.
- *DirX Identity Application Development Guide*. Use this book to obtain information how to extend DirX Identity and to use the default applications.
- *DirX Identity Customization Guide*. Use this book to customize your DirX Identity environment.
- *DirX Identity Integration Framework*. Use this book to understand the DirX Identity framework and to obtain a description how to extend DirX Identity.
- *DirX Identity Web Center Reference*. Use this book to obtain reference information about the DirX Identity Web Center.
- *DirX Identity Web Center Customization Guide*. Use this book to obtain information how to customize the DirX Identity Web Center.
- *DirX Identity Meta Controller Reference*. Use this book to obtain reference information about the DirX Identity meta controller and its associated command-line programs and files.
- *DirX Identity Connectivity Reference*. Use this book to obtain reference information about the DirX Identity agent programs, scripts, and files.
- *DirX Identity Troubleshooting Guide*. Use this book to track down and solve problems in your DirX Identity installation.
- *DirX Identity Installation Guide*. Use this book to install DirX Identity.

- *DirX Identity Migration Guide*. Use this book to migrate from previous versions.

Notation Conventions

Boldface type
In command syntax, bold words and characters represent commands or keywords that must be entered exactly as shown.

In examples, bold words and characters represent user input.

Italic type
In command syntax, italic words and characters represent placeholders for information that you must supply.

[ ]
In command syntax, square braces enclose optional items.

{ }
In command syntax, braces enclose a list from which you must choose one item.

In Tcl syntax, you must actually type in the braces, which will appear in boldface type.

|
In command syntax, the vertical bar separates items in a list of choices.

...
In command syntax, ellipses indicate that the previous item can be repeated.

userID_home_directory
The exact name of the home directory. The default home directory is the home directory of the specified UNIX user, who is logged in on UNIX systems. In this manual, the home pathname is represented by the notation *userID_home_directory*.

install_path
The exact name of the root of the directory where DirX Identity programs and files are installed. The default installation directory is *userID_home_directory*/**DirX Identity** on UNIX systems and **C:\Program Files\DirX\Identity** on Windows systems. During installation the installation directory can be specified. In this manual, the installation-specific portion of pathnames is represented by the notation *install_path*.

dirx_install_path
The exact name of the root of the directory where DirX programs and files are installed. The default installation directory is *userID_home_directory*/**DirX** on UNIX systems and **C:\Program Files\DirX** on Windows systems. During installation the installation directory can be specified. In this manual, the installation-specific portion of pathname is represented by the notation *dirx_install_path*.

dxi_java_home
The exact name of the root directory of the Java environment for DirX Identity. This location is specified while installing the product. For details see the sections "Installation" and "The Java for DirX Identity".

tmp_path

The exact name of the tmp directory. The default tmp directory is /tmp on UNIX systems. In this manual, the tmp pathname is represented by the notation *tmp_path*.

tomcat_install_path
The exact name of the root of the directory where Apache Tomcat programs and files are installed. This location is defined during product installation.

mount_point
The mount point for DVD device (for example, **/cdrom/cdrom0**).

1. Overview

You can customize DirX Identity extensively using various methods like schema extensions, object descriptions and setting options and parameters through wizards or object pages.

If these methods do not meet your customization requirements, DirX Identity provides mechanisms for customizing DirX Identity features through Java programming.

This document explains the general programming environment and the following specific use cases:

- Extending real-time workflows
- Extending event maintenance workflows
- Extending request workflows
- Extending Web Services
- Implementing operations and activities for consistency workflows
- Implementing a custom connector
- Implementing a Representational State Transfer (RESTful) connector

Note that we do not repeat information that is available in other parts of the DirX Identity documentation. Instead, we reference it.

Most use cases comprise these chapters:

- Prerequisites
- Documentation
- Interfaces
- Training
- Examples
- Hints, tips and tricks

2. Using the Programming Environment

This chapter describes how to work with the programming environment.

2.1. About the Development Environment

The DirX Identity development team uses Eclipse for Java development and Apache Ant for building the jar files. Therefore, delivered projects are Eclipse projects and you will often find a **build.xml** file for building the project with Ant. You can download the latest Eclipse version from http://www.eclipse.org/downloads/ and Apache Ant from http://ant.apache.org/bindownload.cgi.

See the extensive Eclipse Help for instructions on how to set up a Java project and work with Eclipse.

You can use other integrated development environments (IDEs) such as NetBeans for Java and JavaScript development, but this document does not provide any additional hints about their use.

Since most members of the DirX Identity development team work in a Windows environment, file paths are given in Windows notation.

2.2. Debugging

To debug a standalone application that you can start directly with Eclipse, simply follow the Eclipse documentation.

2.2.1. Debugging in IdS-J

To debug your classes in the Java Server IdS-J, you need to:

- Set up a batch file **debugServer.bat** from **runServer.bat**.
- Create an Eclipse Debug Configuration for a Remote Java Application.

To run the IdS-J server in debug mode, start the server via a batch file rather than as a service. Copy the file **runServer.bat** in the folder *install_path***\ids-j\bin** to a new file, for example **debugServer.bat**. Uncomment the line that sets the debug options. The line now reads:

```
SET debug=-DIDM_DEBUG=ON -Xdebug -Xnoagent -Djava.compiler=NONE
-Xrunjdwp:transport=dt_socket,address=48174,suspend=n,server=y
```

This means you can now access the debug port **48174**. Of course, you can change the port if needed.

Next, create a Debug Configuration for the IdS-J server in Eclipse. From the **Run** menu, select **Debug Configurations** …. In the left-hand tree, select **Remote Java Application** and then click **New**. Now you can enter the properties for your Debug Configuration. Select the project with your Java sources under test. Keep **Standard** (**Socket Attach**) as the connection type. For the host, enter the IP address or DNS name where the IdS-J server is running (usually **localhost**) and the port that you entered in the file **debugServer**.**bat**, typically **48174**.

Note that you can debug an IdS-J server running on a separate system, even running in a virtual machine (VM), provided that the firewall settings permit remote access to the debug port.

To debug, first start the server by running the batch file **debugServer.bat** and then click **Debug** in the Eclipse Debug Configuration. The Eclipse Debug perspective should open and you should see the IdS-J threads running.

2.2.2. Debugging in Tomcat

To debug your classes in Apache Tomcat (especially Web Center and SPML Provisioning Services user hooks), there are the following scenarios:

1. Tomcat is installed as a service.
2. You have installed the zip version of Tomcat and start it via **catalina.bat**.

If you have installed Tomcat as a service, open the Tomcat configuration via **Start** > **Programs** > **Apache** > **Configure Tomcat**. Select the **java** tab and then enter the following JVM options analogous to debugging the IdS-J:

- -Xdebug
- -Xrunjdwp:transport=dt_socket,address=48175,server=y,suspend=n

Make sure you are using a port that is not used by another application.

Set up an Eclipse Debug Configuration the same way as described for IdS-J, but use another name for your configuration and enter the correct debug port for Tomcat. For more details, see http://tomcat.apache.org/faq/development.html#rd.

If you start your Tomcat via a batch file, set the following environment variables:

- JPDA_ADDRESS=8000 (or another value such as 48175)
- JPDA_TRANSPORT=dt_socket

Start Tomcat using **catalina jpda start**.

An open source Eclipse plug-in is also available (http://www.eclipsetotale.com/tomcatPlugin.html) that allows you to start, stop and debug Tomcat from within Eclipse in addition to creating war files and configuring Tomcat easily.

3. Extending Provisioning Workflows

This chapter describes how to create Java-based extensions for real-time workflows. This comprises only provisioning workflows. For information on extending password synchronization or event-based processing workflows, see the next chapters.

You can extend provisioning workflows via user hooks, connector filters or custom mapping definitions.

3.1. Prerequisites

We assume that you are familiar with Java and with building Java projects and that Ant and a Java compiler are installed and in your path.

3.2. Documentation

To understand this issue, we recommend that you read the following chapters:

3.2.1. DirX Identity Connectivity Administration Guide

Managing Provisioning Workflows > Managing Java-based Provisioning Workflows

Read the entire chapter to understand the basics of real-time provisioning workflows.

Managing Passwords

If you intend to use password management, you should read this chapter as an introduction.

3.2.2. DirX Identity Application Development Guide

Understanding the Default Application Workflow Technology > Understanding Java-based Workflows

Read the entire chapter, especially these sections:

Java-based Workflow Architecture - provides all information for understanding real-time provisioning workflow features and architecture.

Customizing Java-based Workflows - explains in detail how to customize real-time provisioning workflows. Here you can find three sections:

Using User Hooks - explains how to use user hooks for customization.

Mapping with Java Classes - describes how to build custom mapping definitions. The section **Testing the Real-Time Workflow Mapping Classes** shows how to perform offline testing.

Using Connector Filters - explains how to use connector filters to intercept all operations to and from a connector.

3.2.3. DirX Identity Integration Framework Guide

For more details on the interfaces for real-time workflows, see the chapter **Java Connector Integration Framework**.

3.2.4. JavaDoc

For a detailed description of the relevant Java interface, see the Java documentation in the folders **Documentation/DirXIdentity/ConnFrameWork** and **Documentation/DirXIdentity/RealtimeWorkflows** on the product DVD.

You can also find information about reading the configuration and working with requests and responses in this guide.

Most of the classes reflecting SPML/DSML are generated from the SPML schemata. Therefore, the inline java documentation of these classes does not help. For understanding them, please read the XML schemata and check request and response samples.

3.3. Training

View two webinars that explain this topic:

Real-time Workflows - Part I

Real-time Workflows - Part II

you can download webinars from our support portal. If you do not have access to the support portal, contact your responsible support organization.

3.4. Writing a Custom Mapping

You can find sample sources in the following folder of your installation DVD:

Additions\RealtimeWorkflows\samples

The Java Eclipse Project **dxmTestMapping** can help you develop and test your Java mapping classes needed for a specific real-time workflow. The project is delivered on the DirX Identity DVD as a zip file in **Additions/MappingTest** and can be unpacked to any location in the file system. It has all the necessary libraries in its own subfolder and is independent of any installed DirX Identity files.

3.5. Writing a User Hook

For samples of user hooks, review the contents of the **Additions/RealtimeWorkflows/samples** folder on the product DVD. They demonstrate basic user hook handling and provide several valuable scenarios, such as working with blacklists in LDAP. The sub-folder **userhooks** provides all user hook implementations that are used in the default provisioning workflows.

One important sample demonstrates how to send an email notification from a user hook. The Java class notifyMailAccountCreationUserHook.java is contained in the sub-folder **userhooks/common**. It is used in the sample workflow for Extranet Portal for mail notification of account creations.

See the chapter "Loading the Connectivity Scenario" in the *DirX Identity Tutorial* for more information about the My-Company scenario.

Read the chapter "Implementing a User Hook for Email Notifications" in the *DirX Identity Application Development Guide*.

3.6. Writing a Connector Filter

A sample is provided on the product DVD in the folder:

Additions/RealtimeWorkflows/samples

3.7. Hints, Tips and Tricks

For writing mapping classes, user hooks and connector filters, we recommend starting with the Eclipse project dxmTestMapping. Check the chapter "Testing the Real-Time Workflow Mapping Classes" in the *DirX Identity Application Development Guide*.

The project is delivered on the product DVD in the **Additions/Mapping Test** folder together with a readme. It allows you to test and debug your mapping class in a local environment outside the IdS-J server.

You can develop and test in several stages:

1. Test with special EntryMappingController reading one source entry from a file and writing the mapping result also to a file.
2. Test with the same controller, but read the source entry from the real connected system.
3. Test with the same controller and write the modifications for the mapped entry to the real connected system.
4. Test with the real synchronization controller (Synchronize To Target System or Validation) as a unit test running completely in Eclipse.

Copy the appropriate configuration snippets for the connector and the mapping from DirX Identity Manager by selecting the appropriate port entry beneath the join activity of the workflow definition from the **Content (resolved)** tab. Check the system design mode in the menu bar to see the resolved content.

After you have successfully tested your Java classes, deploy the workflow to the IdS-J server and test it there. If you encounter problems that cannot be solved in your local test environment, start IdS-J for debugging as described above.

Please follow also the guidelines on checking and handling timeouts in the chapter 9 “Implementing a Custom Connector”.

4. Extending Password Synchronization Workflows

This chapter describes how to create Java-based extensions for password synchronization workflows. This comprises currently only the User Password Event Manager workflow. For information on extending event-based processing workflows, see the next chapter.

You can extend the User Password Event Manager workflow via user hooks.

4.1. Prerequisites

We assume that you are familiar with Java and with building Java projects and that Ant and a Java compiler are installed and in your path.

4.2. Documentation

To understand this issue, we recommend that you read the following chapters:

4.2.1. DirX Identity Connectivity Administration Guide

Managing Passwords

You should read this chapter as an introduction.

4.2.2. DirX Identity Application Development Guide

Understanding the Default Application Workflow Technology -> Understanding Java-based Workflows

Read the entire chapter, especially these sections:

Java-based Workflow Architecture - provides overall information for understanding Java-based workflow features and architecture.

Customizing Password Synchronization Workflows - explains in detail how to customize password synchronization workflow, namely usage of user hooks:

Using User Hooks - explains how to use user hooks for customization.

4.2.3. JavaDoc

For a detailed description of the relevant Java interface, see the Java documentation in the folders **Documentation/DirXIdentity/ConnFrameWork** and **Documentation/DirXIdentity/RealtimeWorkflows** on the product DVD.

You can also find information about reading the configuration and working with requests and responses in this guide.

Most of the classes reflecting SPML/DSML are generated from the SPML schemata. Therefore, the inline java documentation of these classes does not help. For understanding them, please read the XML schemata and check request and response samples.

4.3. Writing a User Hook

For samples of user hooks, review the contents of the **Additions/RealtimeWorkflows/samples** folder on the product DVD.

A simple implementation of the password user hook IPasswordUserHook is available as an abstract Java class AbstractPasswordUserHook.java and is contained in the sub-folder **userhooks/common**. It should be used as a common super class for custom implementation of password user hook in order to minimize implementation effort.

5. Extending Event-based Processing Workflows

This chapter describes how to implement and manage user hooks for event-based processing workflows.

5.1. Prerequisites

We assume that you are familiar with Java and with building Java projects and that Ant and a Java compiler are installed and in your path.

5.2. Documentation

To understand this issue, we recommend reading the following chapters.

5.2.1. DirX Identity Application Development Guide

Understanding the Default Application Workflow Technology > Understanding Java-based Workflows

Read the entire chapter, especially these sections:

Java-based Workflow Architecture - provides all information for understanding real-time provisioning workflow features and architecture.

Customizing Event-based Maintenance Workflows - explains in detail how to customize event-based maintenance workflows. Here you can find three sections:

Implementing a User Hook for an Event Maintenance Workflow - explains how to create a user hook. The section **Configuring a User Hook for an Event-based Maintenance Workflow** describes how to configure an implemented user hook.

Using Event Contexts - explains some useful helper methods when working with event-based workflow user hooks.

Deploying a User Hook - describes how to deploy an implemented and configured user hook.

5.2.2. Java Documentation

For the Java documentation of the API, see the folder

Documentation/DirXIdentity/ /EventMaintenanceWorkflows

on the product DVD.

5.3. Training

View the webinar that explains this topic:

Event-based Workflows

Note: you can download webinars from our support portal. If you do not have access to the support portal, contact your responsible support organization.

5.4. Examples

For a sample implementation, see the folder "**Additions/EventMaintenanceWorkflows**" on the installation media.

5.5. Hints, Tips and Tricks

To write a user hook, we recommend that you follow the same approach as for writing mapping classes and mapping user hooks: first test and debug in your local Eclipse environment, and then run the workflow in IdS-J.

You can run the activity for an event-based maintenance workflow as a framework job. Then the event maintenance controller together with your user hook runs in the Eclipse environment and you can easily debug it.

For running such a job as a unit test in Eclipse, see the hints in the chapter on mapping classes.

5.5.1. Job Configuration

The configuration file should have the following structure:

```
<job>
    <controller name="EventController"
        className="com.siemens.idm.jobs.ebr.AccountEventController">
        <logging level="5" filename="src.test/confs/ebr/trace.txt">
        </logging>
        <property name="server" value="localhost"/>
        …
     </controller>
    <connector role="reader"  name="EventFileReader"
     …
     </connector>
</job>
```

The root element <job> contains two child elements:

<controller> configures the event maintenance controller for users, accounts, and so on.

<connector> configures the component, which reads an input event for a file.

You can copy most of the <controller> element from the LDAP activity entry of the appropriate workflow. In Identity Manager, navigate to the event maintenance workflow of the appropriate object type; for example, for accounts. Beneath the workflow, select the *join* activity and then open the tab with the resolved content. Here you find the XML document that represents the activity configuration. Copy the part with the <job> element to your configuration file. Enter the <logging> element with the log level and the name of the log file.

Use the following XML snippet for the reader configuration in your file:

```
    <connector role="reader"  name="EventFileReader"
className="siemens.dxm.connector.framework.event.SpmlEventFileReader"
>
        <connection type="SPML"
            filename="src.test/confs/ebr/request.xml">
            <property name="runInParseMode" value="true" />
            <property name="validate" value="false" />
        </connection>
    </connector>
```

Adapt the location of the file that contains the input events.

5.5.2. Input Events

The file with the input events has the following structure:

```
<events>
<addEvent
name="dxm.event.SvcTSAccount.cluster='localhost'.resource='cn=My-
Company'"
…
</addEvent>

<modifyEvent
name="dxm.event.SvcTSAccount.cluster='localhost'.resource='cn=My-
Company'"
…
</modifyEvent>
…
```

```
</events>
```

The root element <events> contains one or more event children. They are either <addEvent>, <modifyEvent> or <deleteEvent>.

Each event is the extension of the respective SPMLv1 request and contains the same child elements as the SPML request.

Each event contains the SPML <identifier> and a <source> element and optionally SPML <operationalAttributes>. In addition, the <addEvent> contains SPML <attributes>, while the <modifyEvent> contains SPML <modifications>.

The name attribute of the event reflects the topic of the message. It is not important for this test configuration. The IdS-J server evaluates it to find the appropriate workflow.

Typically the workflow will not evaluate the <source> element, but the user hook may do so if it is of interest which component has produced the event. Here is a sample snippet:

```
     <source application="joinEngine"
         resource="cn=My-Company"
         cluster="localhost"
     />
```

The *application* attribute contains the name of the sending component. The term joinEngine is used for the provisioning workflows and indicates here that the event was produced because the validation or synchronization workflow changed the LDAP entry.

The most important part will be the <attributes> of an <addEvent> and the <modifications> of a <modifyEvent>. They contain the list of attributes that were changed. The actions of the event controller and the user hook will most likely depend on them.

Here is a snippet of <attributes>:

```
<spml:attributes>
   <spml:attr name="objectclass">
        <dsml:value>dxrTargetSystemAccount</dsml:value>
        <dsml:value>top</dsml:value>
   </spml:attr>
   <spml:attr name="description">
         <dsml:value>test account created new</dsml:value>
   </spml:attr>
</spml:attributes>
```

And here is one for <modifications>:

```xml
<spml:modifications>
   <spml:modification name="dxrTSState" operation="replace">
           <dsml:value>ENABLED</dsml:value>
   </spml:modification>
</spml:modifications>
```

A <deleteEvent> does not contain any specific child elements; that is, no <attributes> or <modifications>. It contains the <identifier> with the DN of the deleted entry. Here is a sample snippet:

```xml
<spml:identifier type="urn:oasis:names:tc:SPML:1:0#DN">
   <spml:id>cn=Jane Webinar3,cn=accounts,cn=New-
LDAP2,cn=Cluster1,cn=DemoCluster,cn=TargetSystems,cn=My-
Company</spml:id>
</spml:identifier>
```

6. Extending Request Workflows

This chapter describes how to extend request workflows with custom routines.

6.1. Prerequisites

We assume that you are familiar with Java and with building Java projects and that Ant and a Java compiler are installed and in your path.

6.2. Documentation

To understand this issue, we recommend reading these chapters:

6.2.1. DirX Identity Application Development Guide

Understanding the Default Application Workflow Technology > Understanding Request Workflows

Read the entire chapter, especially these sections:

Request Workflow Architecture - provides all information for understanding request workflow features and architecture.

Customizing Request Workflows - explains in detail how to customize request workflows. Here you can find these sections:

Using Variable Substitution - explains how to set variable text sections in mail texts that are replaced during runtime.

Implementing an Activity - describes how to implement and deploy a job for a custom activity. It especially expands on how to read configuration data and workflow objects.

Implementing a Java Class for Finding Participants - describes how to implement your custom algorithm for finding participants.

Implementing Participant Filters and Constraints - shows how you can filter participants and how to implement rules and constraints on participants.

6.3. Interfaces

This chapter provides information about the interfaces you can use.

For the API documentation, consult the following folder on your DVD:

Documentation\DirXIdentity\RequestWorkflows\index.html

6.4. Training

View the webinar that explain this topic:

Request Workflows

Note: you can download webinars from our support portal. If you do not have access to the support portal, contact your responsible support organization.

6.5. Examples

For sample sources, see the following folder on your DVD:

Additions\RequestWorkflows\samples.

6.6. Hints, Tips and Tricks

If you need to read or write in the DirX Identity domain, you can obtain an LDAP connection from the Java server.

For participants finder, constraint and filter, see the *Extended interfaces. They provide a context, which provides a getter for the connection.

If you implement an activity, your job class should extend a provided Abstract Job. This also provides the LDAP connection through a getter method. See the corresponding chapter in the *DirX Identity Application Development Guide* and the sample.

There is no environment outside of IdS-J for running and testing jobs of a request workflow, participant finders and constraints. For information on how to debug with IdS-J, see the chapter "Using the Programming Environment" in this guide.

7. Extending Web Services

This chapter describes how to use and extend web services.

7.1. Prerequisites

We assume that you are familiar with Java and with building Java projects and that Ant and a Java compiler are installed and in your path.

7.2. Documentation

To understand this issue, we recommend reading these chapters:

7.2.1. DirX Identity Integration Framework Guide

The Web Services chapter in this guide describes various aspects of this issue:

Authentication, **Authorization**, **Web Services Runtime Operation**, **Common SPML Aspects** - provide basic information about the subject.

User Management to **Groups Management** - explains specific issues for specific object types and the corresponding web services.

User Hooks - describes how to use the web services user hooks.

Using the Sample Client – provides a sample implementation that can be used as a starting point for your custom implementation.

7.3. Interfaces

This chapter provides information about the interfaces you can use.

For a Java documentation of the user hook interfaces, see the folder

Documentation/DirXIdentity/ProvisioningWebServices

on your installation media.

7.4. Training

View the webinar that explains this topic:

Web Services

Note: you can download webinars from our support portal. If you do not have access to the support portal, contact your responsible support organization.

7.5. Examples

You can find a sample Java-based SOAP client that can be used as the basis for your custom implementation in the library **com.siemens.dxm.provisioning.jar** (you can find the jar file in the installation area). It contains the classes that represent all the requests and responses.

For a sample implementation of a user hook, see the folder

Additions/ProvisioningWebServices/samples

on your installation media.

8. Implementing Consistency Rules

This chapter describes how to implement consistency rules.

8.1. Prerequisites

We assume that you are familiar with Java and with building Java projects and that Ant and a Java compiler are installed and in your path.

8.2. Documentation

To understand this issue, we recommend reading these chapters:

8.2.1. DirX Identity Provisioning Administration Guide

Managing Automatic Provisioning and Consistency Checking

Read the entire chapter, especially these sections:

- **Understanding Rule Types** > **Consistency Rules** - explains the concept of consistency rules and describes the default consistency rules delivered with the product.
- **Managing Consistency Rule Operations** > **Using Operations Based on Java Methods** - explains the basics of Java-based operations and rules.
- **Managing Consistency Rule Operations** > **Adding a Java Operation** - explains how to integrate your custom Java class.

8.2.2. DirX Identity Customization Guide

Writing Java Extensions

Read the section "Writing an Extension to Implement a Consistency Rule" to understand how to build your own custom class.

8.3. Training

We do not provide any special training or webinars for this issue.

8.4. Examples

A configuration sample for the My-Company sample domain is provided in the file **rules.zip** on your DVD in the folder:

Documentation\DirXIdentity

You can find the corresponding description in the section "Writing an Extension to Implement a Consistency Rule" in the chapter "Writing Java Extensions" in the *DirX Identity Customization Guide*.

8.5. Hints, Tips and Tricks

This section gives additional hints and guidelines.

8.5.1. Using Consistency Rules in Event-based Operation

Writing consistency rules that save objects through the service layer means that each save operation issues a corresponding event if you work with event-based workflows. Try to minimize save operations to avoid triggering too many event-based workflows.

If you run consistency rules from event-based processing workflows, this means that consistency rules perform save operations as well as the event-based job itself. To avoid this problem, integrate your code into user hooks and pass the result to the event-based job. The result is that only one save operation is performed.

8.5.2. Testing a Consistency Rule

You can write your own unit test to test your Java-based consistency rule outside the workflow and without the rule processor. In the unit test, provide a SvcSession object to the Identity domain and a RuleContext.

The following sample snippet shows how you can easily create a session:

```
    /**
     * Creates a session to sample domain with domain admin
     * and binds.
     * @return session to sample domain.
     * @throws StorageException in case no bound session can be
created.
     */
    private static SvcSession getDefaultSession() throws
StorageException {
        SvcSession session = null;
        try {
            session = new SvcSession();
        } catch (Exception e) {
            e.printStackTrace();
            throw (e instanceof StorageException) ? (
StorageException)e : new StorageException(e);
        }
        StorageBindProfile bp = getDefaultBindProfile();
        if (session.Bind(bp) == 0) {
            try {
                String[] ignoreTags=new String[] {"editor"
,"propertypage","action" };
                String rootDN = bp.getRootDN();
                session.loadConfiguration
("storage://DirXmetaRole/cn=Config.xml,cn=Object
Descriptions,cn=Configuration,"+rootDN+"?content=dxrObjDesc",
ignoreTags);
            } catch (Exception e) {
                e.printStackTrace();
                throw (e instanceof StorageException) ?
(StorageException)e : new StorageException(e);
            }
        }
        else {
            throw new StorageException("Bind failed");
        }

        return session;
    }

    /**
     * Produces a default bind profile for access to sample domain.
     * @return bind profile for sample domain.
     */
    private static StorageBindProfile getDefaultBindProfile() {
        StorageBindProfile profile = new StorageBindProfile();
        profile.setHost("localhost");
        profile.setPort(389);
        profile.setRootDN("cn=My-Company");
        profile.setUser("cn=DomainAdmin,cn=My-Company");
        profile.setPassword("dirx");
        profile.setAuthenticationMethod("simple");
        profile.setSSL(false);

        return profile;
    }
```

The following snippet shows how to create a RuleContext and pass it to the Java Action Class:

```
StorageObject subject = session.getObject(subjectDN);
            RuleContext ruleCtx = new RuleContext();
            ruleCtx.setSubject(subject);
            ruleCtx.setSimulateOnly(false);
            ruleCtx.setUserStorage(session);

            ProvisioningServices ps = new ProvisioningServices();
            ps.setContext(ruleCtx);
```

9. Implementing a Custom Connector (General)

This chapter describes how to implement a custom connector.

9.1. Prerequisites

We assume that you are familiar with Java and with building Java projects and that Ant and a Java compiler are installed and in your path.

9.2. Documentation

To understand this issue, we recommend that you read these chapters:

9.2.1. DirX Identity Integration Framework Guide

The chapter **Java Connector Integration Framework** provides information about the interfaces a connector must implement and about its configuration and deployment.

9.2.2. JavaDoc

For the Java documentation of the interfaces, see the folder **Documentation/DirXIdentity/ConnFrameWork** on the product DVD.

The package **siemens.dxm.connector.framework.util** provides some helper utilities like serialization of SPML classes and response creation.

Note that most of the classes reflecting SPML/DSML are generated from the SPML schemata. Therefore, the inline java documentation of these classes does not help. For understanding them, please read the XML schemata and check request and response samples.

9.3. Training

We do not provide any special training or webinars for this issue.

9.4. Examples

For a sample connector, see the classes in the folder SampleConnector/java on the product DVD.

9.5. Hints, Tips and Tricks

This section provides additional hints.

9.5.1. Test

We recommend testing the connector outside of IdS-J with a JUnit Test from within a Java IDE such as Eclipse. For start-up, you can use the same project dxmTestMapping that is mentioned in the section in this guide on hints for provisioning workflow extensions. You don’t need all the included jar files, but that does not matter for development.

Test using a standalone job configuration reading requests from a file. Change the following template to your requirements regarding file locations and connector configuration:

```
<job>
    <controller
className="siemens.dxm.connector.framework.DefaultControllerStandalon
e">
        <logging level="9" filename="yourFolder/trace.txt">
        </logging>
    </controller>

    <connector
        role="reader"
        name="SPML file reader"
        className="siemens.dxm.connector.framework.SpmlFileReader">
        <connection type="SPML" filename="yourFolder/request.xml">
        </connection>
        <property name="validate" value="false"/>
    </connector>

    <connector
        role="connector"
        className="siemens.dxm.connector.sample.SampleConnector"
        name="Sample connector">
        <connection type="file"
filename="yourFolder/response.xml"
            >
            <property name="validate" value="false"/>
        </connection>
    </connector>

    <connector
        role="responseWriter"
        name="SPML File writer"
```

```
className="siemens.dxm.connector.framework.test.SpmlTestWriter">
        <connection type="SPML"
            filename="yourFolder/receivedRsp.xml">
            <property name="referencefile"
                value="yourFolder/referenceRsp.xml"
            />
        </connection>
    </connector>
</job>
```

For a sample on how to run this configuration as a JUnit test, see one of the test classes of the dxmTestMapping project; for example, TestSample.

Use several SPML request files for testing. In order to have more than one request in a file, XML requires a root element. You can use the SPML <batchRequest> for this purpose, but SPML allows only a restricted set of requests within the batch. Especially, the batch must not contain a <searchRequest>. Therefore, the connector framework supports a proprietary extension: the root element <test> allows all types of SPML requests and responses.

9.5.2. Timeout

Workflows might be cancelled due to timeout or a server shutdown. In this case, the server performs a graceful abort and first informs all running worker threads., the server stops the threads after the grace period is over.

The job controller (also called the join engine) handles the cancel indications: it checks before it processes the next entry and stops when cancel has been requested.

Normally, connectors are not concerned with these issues unless they perform operations where they need to wait for another entity; for example, network operations or a batch or system script. In these cases, they should limit waiting time and check whether the activity has been cancelled. They can do this by calling the isCancelled method on the task context.

A connector should implement the interface **siemens.dxm.connector.framework.DxmContext**. The job controller passes the job context via its setContext() method. The connector can get the task context by

```
TaskContext taskContext = (TaskContext)context.get(TaskContext.class)
```

The task context holds the flag indicating an abort. You can check the flag as shown in the following code snippet:

```
if (((taskContext != null) && taskContext.isCancelled())
        || Thread.currentThread().isInterrupted()) {
    // stop and return to job controller
}
```

A hard stop of the thread is indicated by the interrupted flag of the thread. Never ignore InterruptedExceptions! Return immediately to the controller. Make sure you never reset either flag!

10. Implementing a RESTful Connector

This chapter describes guidelines for implementing a connector for RESTful Web Services. Please consider these guidelines, especially the ones on workflow configuration and mapping: most often they can be applied with only slight adaptations.

10.1. Documentation

To understand this topic, we recommend that you read Chapter 9, "Implementing a Custom Connector (General)" in this guide and the additional documentation listed in its "Documentation" section.

The inline Java documentation of the REST-related interfaces and implementing classes is part of the whole connector framework in the folder **Documentation/DirXIdentity/ConnFrameWork** on the product media. See the section REST framework for the REST-related interfaces and classes.

10.2. Workflow Configuration and Connector Attribute Handling

This section provides guidelines for configuring the workflows, especially the mapping. Please follow these guidelines unless there is an important reason not to do so.

10.2.1. Configuring All Entry Types

This section provides guidelines for configuration that apply to all entry types.

10.2.1.1. Handling the Entry Identifier

Most REST services generate a unique identifier for each entry (user, group or role) that they create. They use it to identify the entry in service requests or in (user) attributes as a reference to a linked entry.

The REST service returns this identifier in the response to a create request. The connector should return it in the identifier of the Add Response with an identifier type "GenericString". The workflow should map it to the **dxrPrimaryKey** Identity domain attribute. In the mapping to the connected system, the **dxrPrimaryKey** is used as the identifier for all requests except the Add. As the identifier is not initially known in Identity, the identifier in an Add Request is left empty.

10.2.1.2. Distinguishing Users and Groups

Use operational attributes to allow the connector to distinguish operations between users and groups. For the channel to the connected system, set the operational attribute "objType" to the right value; use either "user" or "group". For additional types, take a value of your own; for example, "role". Here is the snippet for the accounts channel:

```
<operationalAttrMapping mappingType="constant" name="objType">
    <value>user</value>
</operationalAttrMapping>
```

10.2.1.3. Searching All Entries of a Type

For searching all entries of a type (for example, all users), leave the search base empty. This means no string "all" or similar.

10.2.1.4. Searching an Entry with ID

For finding the joined entry in the connected system, use its Identifier (dxrPrimaryKey) and pass it as the search base in the search request.

10.2.1.5. Handling Deleted Entries

If the REST service supports a delete operation, the mapping is straightforward as for other types of systems:

If the dxrState of the Identity entry (account or group) is DELETED, the mapping needs to create a delete request. It performs this task by setting the request type to DELETE as in this snippet:

```
targetMapResult.setRequestType(Request.Type.DELETE)
```

In the opposite direction from the connected system to Identity, set dxrTsState to DELETED if the REST entry doesn't exist or some state attribute indicates the deleted state.

Some systems do not support deleting entries. If they provide an attribute for indicating the deleted state, the mapping is straightforward: configure a Java mapping that provides the appropriate values for both directions. If the service does not provide this kind of an attribute out of the box, consider either appropriating an existing attribute that is not needed in your environment or extending the schema of the connected system to introduce this attribute.

10.2.2. Configuring Accounts

This section provides guidelines for configuring accounts.

10.2.2.1. Handling the Logon Name

The **dxrName** attribute of an Identity account should contain the logon name of the user in the connected system. The object description for the accounts must generate a unique value. A good choice is to take the user's email address. Check the existing template descriptions for samples.

10.2.2.2. Handling Enable / Disable

In principle, we distinguish between the following cases:

- The REST user contains a state attribute; for example, isSuspended.

Use a direct mapping between the Identity attributes **dxrState** / **dxrTsState** and the REST attribute. You will typically need to write a Java mapping because the REST service will almost never support the same values as Identity (ENABLED, DISABLED, TOBEDELETED, etc.).

- The REST service supports suspend / enable operations.

The connector should support a virtual attribute for the state, let’s say **dxrState**. The mapping can be as in the case above and map the Identity states to ENABLED / DISABLED.

The connector must evaluate this attribute in SPML requests passed by the join engine and issue the appropriate suspend / enable operations to the REST service. The connector must also populate this state attribute with ENABLED / DISABLED when returning a searchResultEntry. Often the REST responses contain appropriate attributes that are read-only in these cases.

10.2.3. Configuring Account – Group Memberships

In the Identity domain, always store group memberships at the account. The referenced attribute in the group is dxrPrimaryKey.

At the target system entry, you configure this in the Advanced tab by enabling the flag "Reference Group from Account" and setting "Referenced property" to "dxrPrimaryKey. This setting assumes that the identifier of the REST entry is stored in dxrPrimaryKey (as described in previous sections) and is used in the REST service for linking users and groups.

In the workflow configuration in the Connectivity database, you need to link the members channel from the accounts channel (General tab). In the mapping to the REST service, use the virtual attribute "members" for the member values (that is, the values taken from **dxrPrimaryKey**). The REST connector must evaluate them appropriately, as described in the following sections.

10.2.4. Configuring the Connector

Use the existing standard names for the connector’s configuration parameters as much as possible. Use one of the existing connectors as a template; for example, Salesforce or Office365.

The <connection> attributes user, password, ssl, server and path contain the user’s logon name, its password, whether to use “http” or “https”, the address of the REST service and the relative path to the application respectively.

Authentication is often done via OAuth. In these cases, the connector must send authentication requests to an OAuth service. The attribute **authEndpoint** should contain the relative or absolute path to the OAuth service depending on whether or not it is located on the same server as the "productive" REST service. The connector must identify itself as an application and therefore provide its identifier (in OAuth, it is called the client or application ID) and its secret (kind of password). The client ID and client secret should be configured in the attributes **clientID** and **clientSecret** respectively. Use the LDAP attributes **dxmClientID** and **dxmClientSecret** to store them in the connected directory. Usually, the connector must also provide the name of a user and a password. They should be configured in the attributes **user** and **password** and taken from the bind profile.

If the REST service can only be accessed through a proxy, store its server and port in an additional connected directory, reference it from the connected directory via the attribute **dxmProxyServer-DN** and pass the values as **proxyHost** and **proxyPort** properties to the connector.

Here is a sample snippet of a <connection> template with proxy and OAuth properties:

```
<connection
password="${DN4ID(THIS)@dxmBindProfile-DN@dxmPassword}"
server="${DN4ID(THIS)@dxrConnectionLink@dxmService-DN@dxmAddress}"
ssl="${DN4ID(THIS)@dxmSSL}”
user="${DN4ID(THIS)@dxmBindProfile-DN@dxmUser}">
    <property name="clientId"
value="${DN4ID(THIS)@dxrConnectionLink@dxmClientId}"/>
    <property name="clientSecret"
value="${DN4ID(THIS)@dxrConnectionLink@dxmClientSecret}"/>
    <property name="securityToken"
value="${DN4ID(THIS)@dxrConnectionLink@dxmBindProfile-
DN@dxmSpecificAttributes(securitytoken)}"/>
    <property name="authEndpoint"
value${DN4ID(THIS)@dxrConnectionLink@dxmSpecificAttributes(oauth_path
)@dxmAddress}"/>
    <property name="path"
value="${DN4ID(THIS)@dxrConnectionLink@dxmSpecificAttributes(servicep
ath)}"/>
    <property name="proxyHost"
value="${DN4ID(THIS)@dxrConnectionLink@dxmProxyServer-DN@dxmService-
DN@dxmAddress}"/>
    <property name="proxyPort"
value="${DN4ID(THIS)@dxrConnectionLink@dxmProxyServer-DN@dxmService-
DN@dxmDataPort}"/>
</connection>
```

Only the parameters user, password, ssl, server and port are XML attributes. The others must be configured as <property> elements within the

<connection> element.

10.3. Connector Structure

A connector typically performs the following steps when processing a provisioning operation (add, modify, delete, search):

- Before the first operation: authenticate, mostly via OAuth or basic authentication.
- Determine the type of an entry: user, group. Sometimes also other types need to be supported.
- Transform to the specific REST operation(s): the HTTP operation POST, PATCH (practically never: PUT), DELETE or GET, sometimes with a payload that is most often JSON.
- Often not only one, but a series of operations must be sent. As a result, entry attributes need to be split. A typical example: user – group memberships are managed and queried in additional operations.
- Send the HTTP operation and evaluate the response code.
- For create, extract the identifier of the new entry. For search, evaluate the JSON payload and transform to SPML search result entries.

The connector framework supports this process by providing a connector template that follows this sequence and uses a couple of interfaces and sample implementations together with other common utilities.

The REST template connector is implemented in the **AbstractRestSampleConnector** class. Its source is provided in the folder **Additions/RESTfulConnector** of the product media. The following sections describe how it processes selected requests and the interfaces it requires.

A specific connector can extend this template and can override some of its methods when necessary. Normally, it must provide its own implementations for the important interfaces: the request transformer, the command producer and the response evaluator.

10.3.1. REST-Specific Interfaces

The following figure illustrates the REST-specific interfaces that the **AbstractRestSampleConnector** uses:

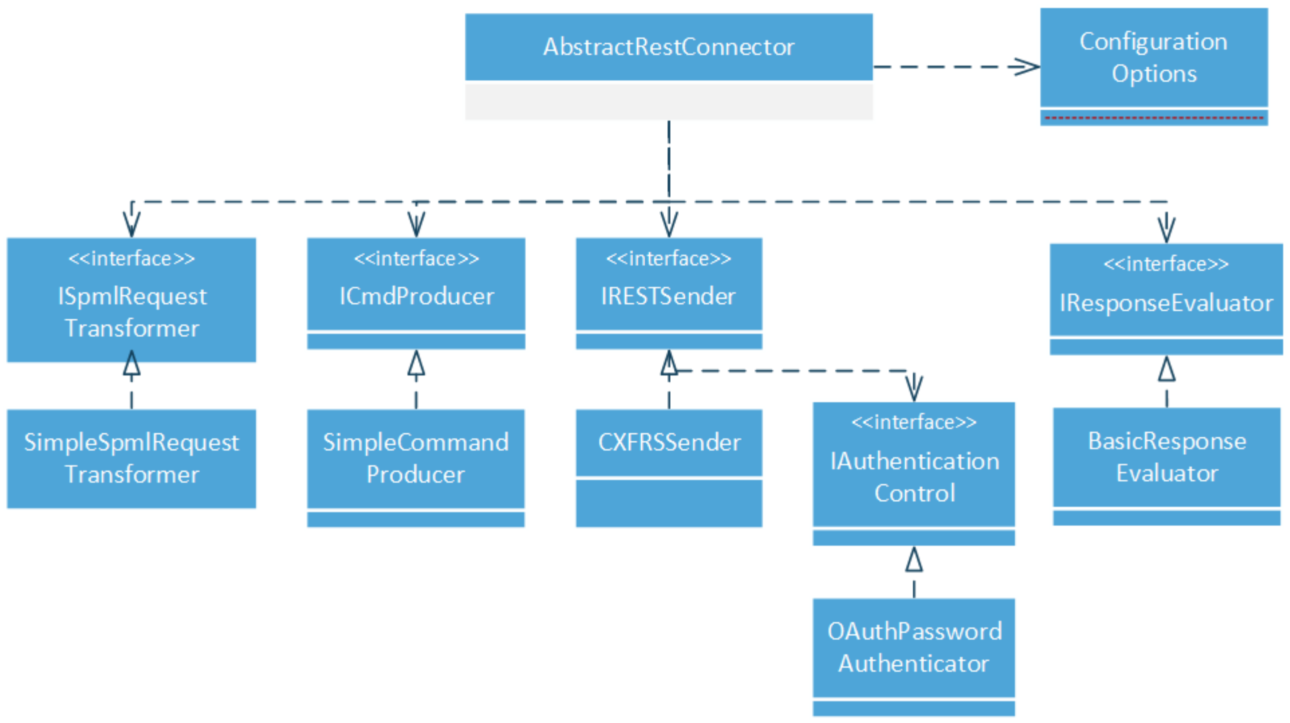


Figure 1. REST Interfaces used by the Abstract REST Connector

The **AbstractRestConnector** class provides the configuration parameters in the **ConfigurationOptions** convenience class.

The interfaces and basic implementations used include:

ISpmlRequestTransformer – transforms the given SPML requests to attribute sets or – for a search request – to filters and requested attributes. A sample implementation is **SimpleSpmlRequestTransformer**.

ICmdProducer – produces the REST operations (GET, POST, etc.) that correspond to the SPML requests (search, add, etc.). A sample implementation is **SimpleCommandProducer**.

IRESTSender – sends the REST operations to the remote REST service. If available, it uses an authentication control for authentication. The recommended implementation is **CXFRSSender**, based on Apache CXF.

IAuthenticationControl – performs authentication and adds the corresponding HTTP header to the REST operation. An implementation for OAuth2 is **OauthPasswordAuthenticator**.

IResponseEvaluator - evaluates the REST responses and supports transforming them to SPML responses. A sample implementation is **BasicRspEvaluator**.

All these interfaces support an **open** method which is used for passing the configuration parameters. For details, see the next sections.

10.3.2. Add User

This section shows how these interfaces work together to create a user.

Each connector implements an **add** method that receives a SPML AddRequest and returns a SPML AddResponse. The template connector extracts the entry type (here we assume User) from the operational attribute (see the section "Configuring All Entry Types") and then calls the specific method (here: addUser).

The following sequence diagram illustrates the first set of actions in addUser:

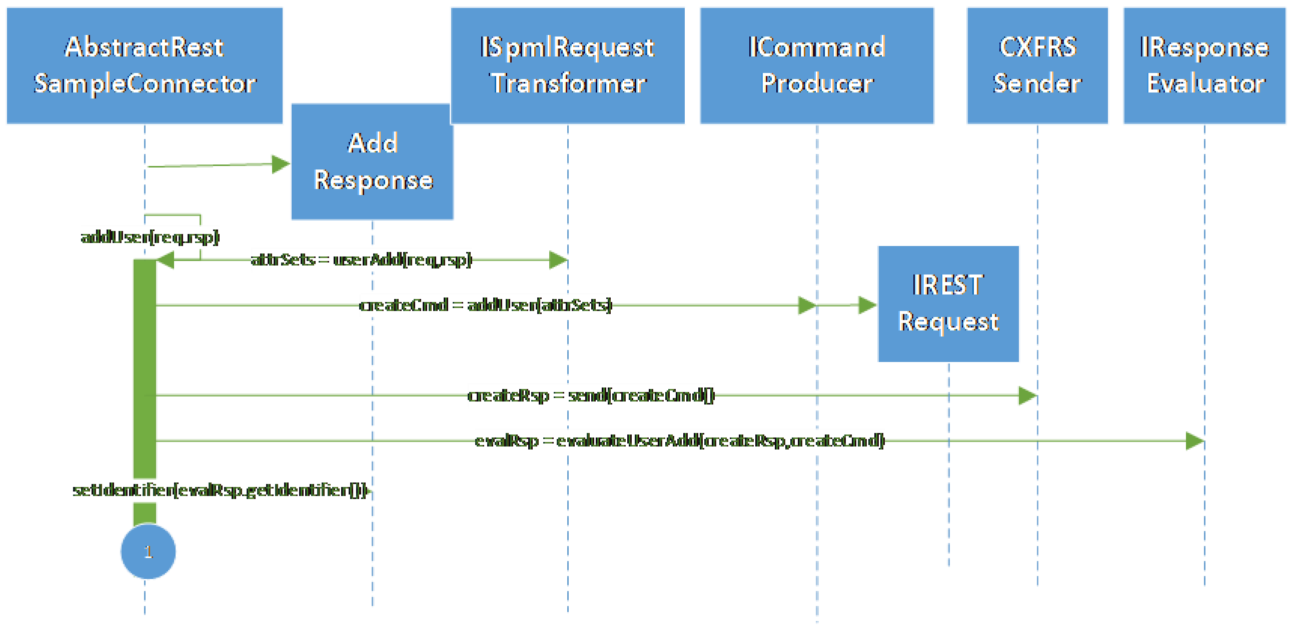

AbstractRest SampleConnector
Add Response
ISpmlRequest Transformer
ICommand Producer
IREST Request
CXFRS Sender
IResponse Evaluator
addUser(req,rsp)
attrSets = userAdd(req,rsp)
createCmd = addUser(attrSets)
createRsp = send(createCmd)
evalRsp = evaluateUserAdd(createRsp,createCmd)
setIdentifier(evalRsp.getIdentifier())
1

Figure 2. addUser Actions - Part 1

The REST connector prepares an AddResponse and then calls the request transformer's userAdd method to separate the attributes from the AddRequest into sets as a preparation for the subsequent commands. For adding a user, we expect one command for creating the user and a list of commands to add the new user to groups. Consequently, the transformer will produce one set with all the attributes required for the user creation and another set with the group memberships. The attribute set for the create operation is a map with the attribute names as keys and the values as either String or list of Strings.

The connector then forwards these attribute sets to the command producer in the method addUser. The command producer produces a REST request, which consists of the HTTP method (in this case: POST), an HTTP path, the attribute set as payload and query parameters.

The connector forwards the command to the REST sender. The sender uses Apache CXF to issue REST operations. The sender handles authentication. If an authenticationControl was set, the sender asks it to extend the HTTP headers. The authenticationControl also might change the endpoint of the REST service. If no authenticationControl is available, the sender assumes basic authentication and produces the appropriate header itself. For more details on authentication, see the "Authentication" section.

The sender transforms the payload to JSON and then sends the REST operation using CXF. It accepts only JSON as the response format and populates the result (payload, HTTP result code along with the REST request) into a REST response and then returns it to the connector.

The connector passes this response to the response evaluator. The evaluator creates and returns an evaluated response. For an addRequest, the response should contain the identifier of the new entry.

The connector puts this identifier into the prepared SPML AddResponse and into the attribute sets so that it can be used in the subsequent commands.

The next actions for handling group memberships and additional attributes are shown in the next diagram:

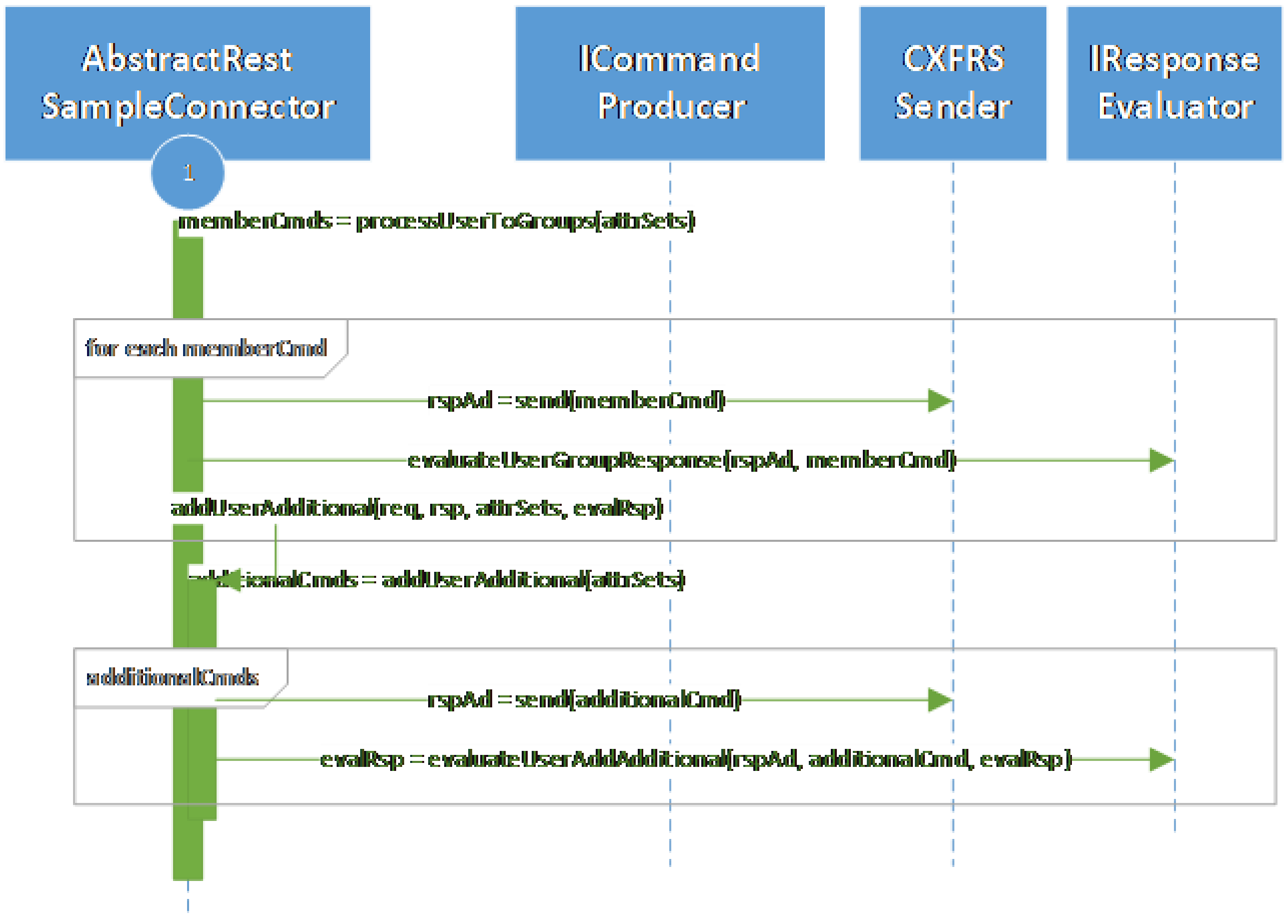

AbstractRest SampleConnector
ICommand Producer
CXFRS Sender
IResponse Evaluator
1
memberCmds = processUserToGroups(attrSets)
for each memberCmd
rspAd = send(memberCmd)
evaluateUserGroupResponse(rspAd, memberCmd)
addUserAdditional(req, rsp, attrSets, evalRsp)
additionalCmds = addUserAdditional(attrSets)
additionalCmds
rspAd = send(additionalCmd)
evalRsp = evaluateUserAddAdditional(rspAd, additionalCmd, evalRsp)

Figure 3. addUser Actions – Part 2

The connector asks the command producer for the commands to create user-group memberships, passes each of them to the REST sender and then asks the evaluator to check the responses. On failures, the evaluator throws an exception and thus stops processing, causing the connector to return an error response.

For some applications, additional commands should be sent to fully provision a new user. This function is handled in the connector method addUserAdditional. The connector asks the command producer for the list of specific REST commands, passes them to the sender and then lets the evaluator check them.

The handling of modify and delete requests is very similar.

10.3.3. Search

This section shows how the interfaces work together to search users. It includes searching in iterations, where the REST service provides the result in chunks, each of which must be explicitly retrieved.

10.3.3.1. First Search

In the search method, the connector receives an SPML SearchRequest and returns a SearchResponse. As in the other methods, the template connector extracts the entry type (here we assume User) from the operational attribute (see the section "Configuring All Entry Types") and then calls the specific method (here: searchUser).

The following sequence diagram gives an overview of the actions in searchUser. Searches for other entry types are very much the same.

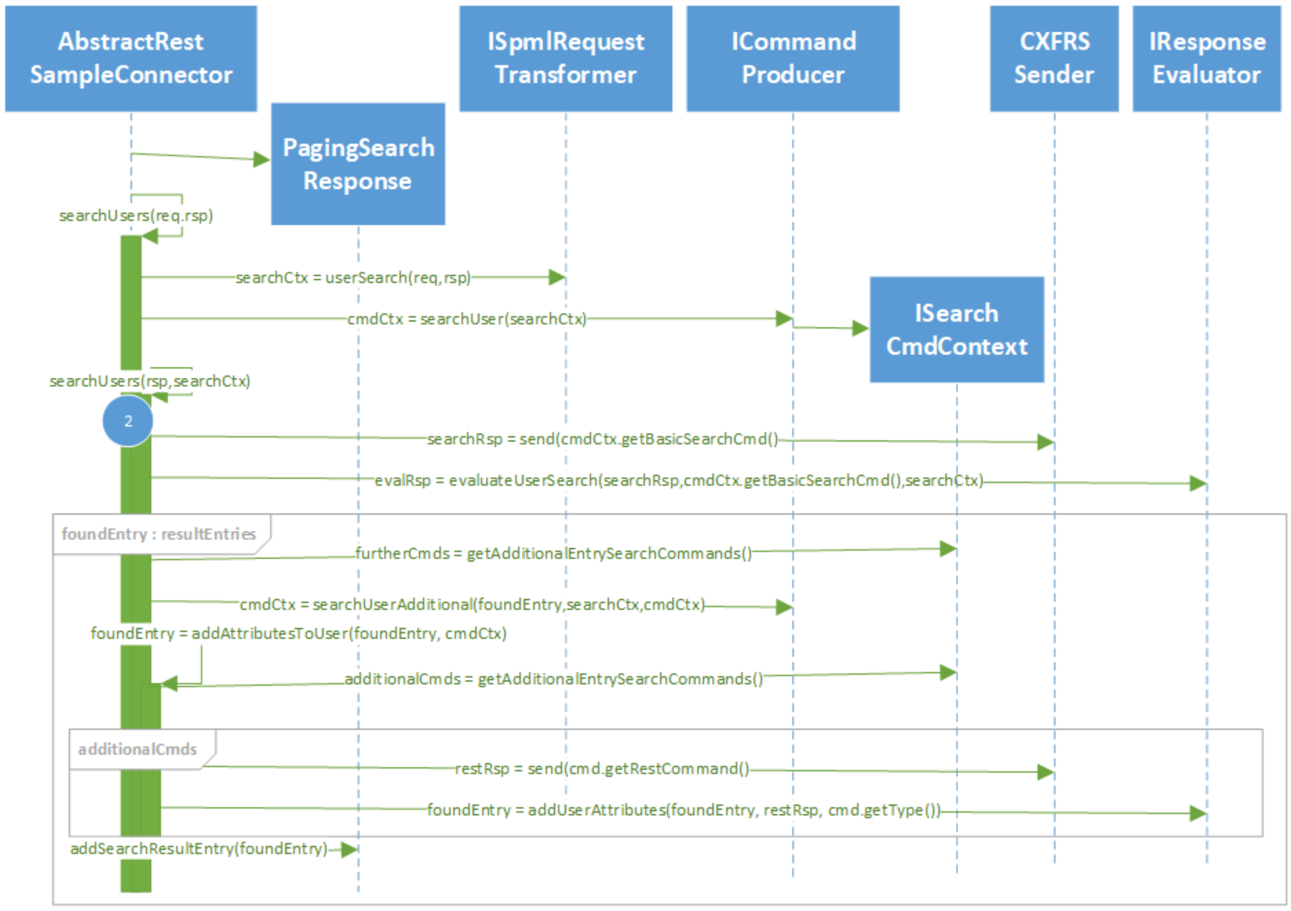

AbstractRest SampleConnector
PagingSearch Response
ISpmlRequest Transformer
ICommand Producer
ISearch CmdContext
CXFRS Sender
IResponse Evaluator
searchUsers(req,rsp)
searchCtx = userSearch(req,rsp)
cmdCtx = searchUser(searchCtx)
searchUsers(rsp,searchCtx)
2
searchRsp = send(cmdCtx.getBasicSearchCmd())
evalRsp = evaluateUserSearch(searchRsp,cmdCtx.getBasicSearchCmd(),searchCtx)
foundEntry : resultEntries
furtherCmds = getAdditionalEntrySearchCommands()
cmdCtx = searchUserAdditional(foundEntry,searchCtx,cmdCtx)
foundEntry = addAttributesToUser(foundEntry, cmdCtx)
additionalCmds = getAdditionalEntrySearchCommands()
additionalCmds
restRsp = send(cmd.getRestCommand())
foundEntry = addUserAttributes(foundEntry, restRsp, cmd.getType())
addSearchResultEntry(foundEntry)

Figure 4. searchUser Actions

First, the connector prepares a SearchResponse that is returned to the join engine at the end. To support paging, it creates a special PagingSearchResponse. The PagingSearchResponse extends the normal SPML Search Response and provides additional features for retrieving the subsequent page of result entries when the current page has been processed. This implementation ensures that always only one page is in memory so that very large result sets can be processed.

By calling the method userSearch, the connector asks the request transformer to produce a search context. A search context contains the information needed for issuing a query: filter, search base, requested attributes, and operational attributes as well as the original SPML request and the current SPML response.

With this search context, the command producer creates a search command context. A search command context contains a list of REST requests: one basic (that is, the first or main) and an optional list of additional ones. The basic search command is supposed to obtain a list of entries matching the filter with a set of attributes. The other commands serve to obtain additional attributes per entry.

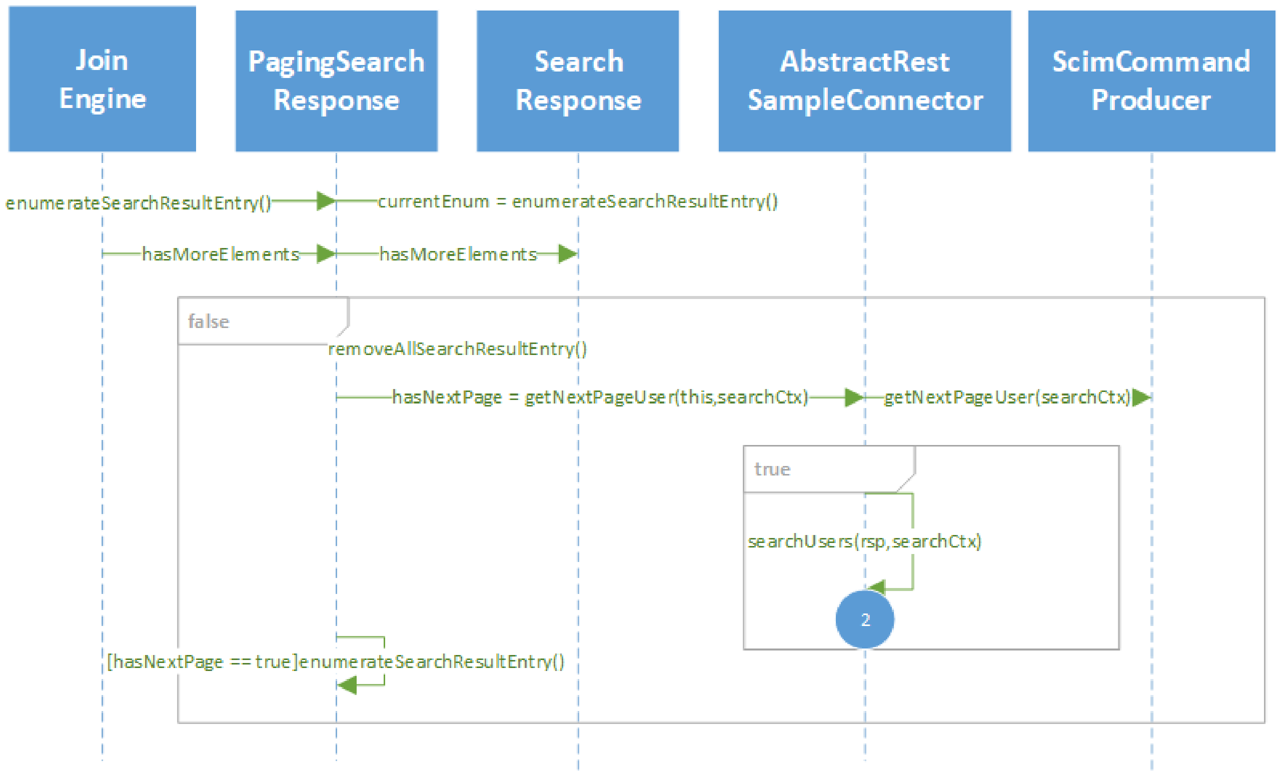
The rest of the search is handled in an extra method, searchUsers, which expects the search response and the search context as parameters. This method is also used when retrieving subsequent result pages so that the common actions can be handled by only one method.

The connector passes the basic search command to the REST sender and passes the returned REST response to the response evaluator along with the search command and the search context. The evaluator transforms the result to a list of SPML SearchResultEntries and returns them in an evaluated response.

For each result entry, the connector asks the command producer to create further commands for retrieving additional attributes of the entry (here: user). The connector passes them to the REST sender and asks the response evaluator to extract them from the REST response to the found SPML result entry. After the last command, the connector adds the result entry to the SPML search response.

10.3.3.2. Retrieving Subsequent Pages

For convenience, the join engine of the provisioning job (aka job controller) obtains the list of result entries one by one by calling the method enumerateSearchResultEntry of the search response. This gives the search response class control and allows it to re-issue subsequent searches after the first chunk has been processed. As a disadvantage, it might not know the total number of result entries at the beginning.


Join Engine
PagingSearch Response
Search Response
AbstractRest SampleConnector
ScimCommand Producer
enumerateSearchResultEntry()
currentEnum = enumerateSearchResultEntry()
hasMoreElements
hasMoreElements
false
removeAllSearchResultEntry()
hasNextPage = getNextPageUser(this,searchCtx)
getNextPageUser(searchCtx)
true
searchUsers(rsp,searchCtx)
2
[hasNextPage == true]enumerateSearchResultEntry()

The PagingSearchResponse delegates enumeration of entries to the underlying SPML Search Response. As long as it has more elements, it just returns them to the join engine.

If the first chunk is exhausted, the PagingSearchResponse requests the next chunk from the connector by calling its method getNextPageUser (for users) and passes the current search context.

The connector asks the command producer for the command to retrieve the next page (here: of users). If there is one, it calls the same method searchUsers as in processing the original search request. When it returns, the entries of the current page are in the search response.

Now the PagingSearchResponse can again enumerate over the result entries and pass them to the join engine.

10.3.4. Read an Entry

As mentioned above, reading a single entry should be implemented by a search request with a search base. The search base represents the identifier of the entry.

The connector performs the same actions as outlined above. Just setting the response should be a bit different. If the entry cannot be found, the response should have an error code NOSUCHIDENTIFIER and a custom error code NO_SUCH_OBJECT. This is important for the provisioning join engine; in that case it continues trying to find the joined entry with the next join expression.

10.4. Sample Connector

We provide the sources for a sample connector **BasicScimConnector** as a sample. The sample extends the **AbstractRestSampleConnector** template and just sets the OAuth Authentication control and a simple response evaluator.

Note that despite its name, the connector does not (yet) implement the SCIM specification. It takes some elements of SCIM to provide a simple demonstration rather than invent a new REST protocol.

10.5. Authentication

Authentication is handled within the REST sender.

The (derived) connector can set an authentication control in the REST sender. An authentication control must implement the interface **IAuthenticationControl**.

IAuthenticationControl methods include:

open – used for passing the configuration parameters.

getAuthenticationHeaders – allows the authentication control to return the authentication- and authorization-relevant HTTP headers that must be added to a REST request.

getDynamicEndpoint – allows the authentication control to return an application endpoint generated dynamically from the authentication service, thereby instructing the REST sender to use it for the next REST request.

evaluateErrorResponse – allows the authentication control to check the result code and error message of a JAX-RS response to determine whether or not to refresh an access token for the next request.

If no authentication control is passed, the sender uses the user name and password from the configuration for basic authentication.

If an authentication control is available, the connector asks the control for additional (authentication and authorization) headers and then adds them to the REST command. The control may also provide an endpoint to which requests are subsequently to be sent. This parameter supports scenarios where the address of the REST service is determined by the authentication service.

The framework provides an authentication control for OAuth2, the **OauthPasswordAuthenticator**. It obtains a new token when required and a refresh token once the previous is expired and adds it in a HTTP header "Authorization Bearer".

For accessing the OAuth2 service, the control evaluates the configuration options *authorizationServerUrl* (address), *clientId* and *clientSecret* (see OAuth2): identification or user name of the connector as a client application and its password.

10.6. ISpmlRequestTransformer Interface

An implementation of **ISpmlRequestTransformer** must evaluate SPML requests for users, groups and other entry types and then return appropriate attribute sets:

For add, modify and delete requests - an object that implements **ISeparatedAttributes**. A sample implementation is **BasicSeparatedAttributes**. This kind of object provides the attributes in various sets:

Basic attributes for the first or main operation to create or modify an entry.

Attributes for either adding a user to a group or removing it.

The identifier of the (new) entry.

The list of additional attribute sets, when more operations are needed to create, update or delete an entry.

For search requests – an object that implements **ISearchContext**. A search context provides the search filter, requested attributes, a search base and operational attributes. A sample implementation is **BasicSearchContext**.

In the **open** method, the connector passes the configuration options for the entire connector that might also be used by the request transformer.

A sample implementation for a request transformer is **SimpleSpmlRequestTransformer**. It assumes that the attribute "memberOf" is the one that contains the groups of which a user is member and provides them as groupMemberToAdd/Delete. All the other attributes go into the basic attribute set. It also respects a list of multi-valued attributes. The values for these attributes are provided as lists. The names of multi-valued attributes can be set from the connector.

From a search request, it extracts the search base and the requested attributes, but not (yet) the filter.

10.7. ICmdProducer Interface

An implementation of the **ICmdProducer** interface produces the REST commands for creating, updating, deleting and searching for users, groups and other entries. The main inputs are the attribute sets and search contexts produced by the request transformer.

In the **open** method, the connector passes the configuration options for the entire connector that may also be used by the command producer.

For each entry type (user, group or other) there are the following methods:

*Add / modify / delete*_Type_ – for creating / updating or deleting the basic attributes of an entry. The method is expected to produce a REST request with either a POST, PUT, PATCH or DELETE operation, the relative path that needs to be added to the application endpoint, the query parameters, the headers and the (preferably JSON) payload.

ProcessUserToGroups – for returning the REST requests for adding a user to and removing it from groups. The returned list typically has a request per user-group membership.

Add / modify / delete*Type*Additional – for performing additional requests for managing special attributes or privileges such as licenses, profiles, etc. The command producer returns a (potentially empty) list of REST requests that are processed one after the other.

*Search*Type – for finding all entries (of the specific type user, group or some proprietary one) matching the filter. Typically, the searches will try to find all entries of a given type (for example, all users) or only one entry associated with the given source entry (called the joined entry). In this case, the filter will just reference a unique identifier (the primary key) or a few basic attributes. These methods expect a search context as input and return a search command context.

Search*Type*Additional – for finding the additional attributes of a given entry. The method expects the search context, the search command context produced by the previous Search*Type* method and the evaluated REST response associated with the previous search as input parameters. It returns a search command context with explicitly the additional typed search requests populated. The type allows for distinguishing several attribute sets (for example, groups, licenses). Unfortunately, the identifier of the individual entry is not known to the producer at the time this method is called. As a result, the connector must currently insert it somehow into the REST command.

*GetNextPage*Type – for retrieving the next page (chunk) of result entries if there are some left. The information needed to decide that should be put to the search context by the response evaluator.

The **SimpleCommandProducer** class provides a straightforward implementation of a command producer inspired by SCIM. It can be used as a copy/paste starting point for specific command producers. The framework also provides implementations for search command context and REST requests that are basically property holders.

10.8. IRESTSender Interface

An implementation of the **IRESTSender** interface sends REST requests according to REST command objects passed to it and returns corresponding REST responses. The sender handles authentication. The connector may pass an authentication controller, which the sender then uses to obtain authentication headers and/or query parameters.

10.8.1. Interface Methods

In the **open** method, the connector passes the configuration options for the entire connector that can also be used by the sender.

The **setAuthenticationControl** method expects an authentication controller implementing the interface **IAuthenticationControl**. For details on authentication, see the "Authentication" section.

In the **send** method, the sender sends a REST request according the given REST command (interface **IRESTRequest**; see the section "ICmdProducer Interface". The sender returns a

REST response object that holds the HTTP result code, the full original response, the payload, the headers and potentially an error message.

10.8.2. CXFRSSender Implementation

The framework provides the **CXFRSSender** class as an implementation of the **IRESTSender** interface. CXFRSSender uses Apache CXF for sending REST requests and evaluating the responses.

The sender accepts only JSON REST responses. It also assumes JSON for the payload of requests, which can be overridden by the request media type. The Jackson JSON provider helps transform the message payload to and from Maps. Consequently, the connector and the other components don’t need to manage JSON handling; they simply produce and consume Java Maps.

When the connector doesn’t set an authentication control, CXFRSSender assumes basic authentication and uses the configured user and password for producing the authentication header.

When a proxy host is configured, the sender instructs the CXF Web client to send the requests to the proxy host and port from the configuration. Currently, no authentication for the proxy is supported.

10.9. IResponseEvaluator Interface

A response evaluator implements the **IResponseEvaluator** interface and needs to check the REST responses, extract identifiers and attributes from the payload, transform them to SPML and then feed them into the evaluated response.

The interface methods are organized according to entry types (user, group, other) and operations (add, modify, delete, search). In the **open** method, they receive the configuration parameters for the connector.

All the methods must check the response for errors. If the HTTP result code indicates failure, the REST sender sets the result of the internal REST response to failure. All the methods get the REST request and the corresponding REST response with the payload.

Some of the methods (modify, delete) normally do not contain a payload or the payload is not evaluated by the join engine. So the evaluator can simply ignore them.

When an entry is created, the response payload most often contains the identifier of the new entry. In the methods **evaluate***Type***Add**, the evaluator must extract it and put it into the evaluated response.

The payload of GET operations contains the matching entries as a list or a single entry. The evaluator receives the payload in two groups of operations:

evaluate*Type***Search** – the payload is expected to contain the basic attributes of the matching entries. The evaluator must transform each of them into a SPML result entry and then put them into the search result entries list of the evaluated response. Each result entry needs to have an identifier and can have a list of single- or multi-valued attributes.

For paging support, the evaluator must also read response attributes that tell whether the result is complete or how to retrieve the next chunk. This information should be stored in the search context.

add*Type***Attributes** – the payload contains additional attributes of an entry. The evaluator must add them to the existing SPML entry. The entry with the attributes already obtained is passed as an input parameter. Another parameter passes a type that if present indicates the attribute set that is expected in the payload. A typical type will indicate the groups of which a user is a member.

The **BasicResponseEvaluator** class provides a simple implementation of the interface and returns the simple implementation **BasicEvaluatedResponse**. This evaluator checks the REST response and throws an exception on failure. A specific implementation can extend it and then must implement the **evaluate*Search** methods at a minimum, and frequently the **add*Attributes** methods, too. The **ScimRspEvaluator** implementation shows how to extract simple String attributes from the payload and create SPML search result entries and the SPML identifier. It also reads the number of total results, start index and items per page that can be used to decide whether a request to retrieve a subsequent chunk of entries makes sense.

10.10. REST Utilities

The **RESTUtils** class provides some utility methods that are helpful for sending and evaluating REST requests and responses.

Some methods implement transformation between JSON strings and maps and lists or to base64 strings.

Other methods check the HTTP status code with respect to the HTTP operations.

The remaining methods support logging requests and responses.

10.11. SPML and Framework Utilities

The connector framework provides some utilities that can help to manage SPML requests and responses. You can find them in the Java package **com.siemens.dxm.spml.util**.

The **ResponseCreator** provides methods for creating a success or failure response that accept the SPML request and an error message or an exception.

The **Serializer** serializes a SPML request and response to a string, which can be useful for logging.

The **SpmlUtils** class provides methods for:

- Creating attributes from a name and value or list of values.
- Obtaining attributes as a map, get values as map or list.

- Obtaining the first value from an attribute map or list.
- Obtaining the identifier string or produce an identifier from a string value.
- Getting or setting some specific operational attributes, such as the scope or the object type.

10.12. Examples

The sources for all the REST-related interfaces, implementing classes and utilities are provided in the folder **RestFramework** on the product media.

11. Using the User LDAP Lock

The purpose of the User LDAP lock is to prevent two or more applications / programs / threads from updating the same user in parallel.

The implementation has drastically changed in V8.10. In order to minimize locking times and to relieve custom code from lock handling, the following major changes have been made:

- Components, namely custom clients and Java Scripts do not have to set and release the lock explicitly. Instead, it is done within the *SvcUser* method *'storePrepared'* and only when it is necessary. Locking is considered necessary, if a privilege assignment is created, changed, or deleted or if a permission parameter attribute of the user is changed.
- User resolution is not performed anymore in the client applications (such as Web Center, REST and SOAP Services, consistency rules, etc), but in an extra component, the new resolution adapter running in the Java server IdS-J.
- Clients, including Java Scripts, should always use the method *checkAndSave(true)* for storing their changes to a user. This method checks the changes for their relevance to access rights, lock the user and send a resolution event if an access-right relevant change has been detected.

To support upgrade, the old methods in class *SvcSession* (such as *createLdapLock*) are kept and do not perform any changes. They are marked deprecated, so please take care to adapt your custom clients to not use them anymore. They likely might be dropped in a subsequent version.

Your client code still must consider that the method *checkAndSave* – or *storePrepared*, if you use it – might fail because it is not able to acquire the lock for changing the user. This will be indicated by the result code 15 (SVC_ENTRY_LOCKED) in the returned SvcSummary object.

DirX Product Suite

The DirX product suite provides the basis for fully integrated identity and access management; it includes the following products, which can be ordered separately.

DirX Identity

DirX Identity provides a comprehensive, process-driven, customizable, cloud-enabled, scalable, and highly available identity management solution for businesses and organizations. It provides overarching, risk-based identity and access governance functionality seamlessly integrated with automated provisioning. Functionality includes lifecycle management for users and roles, cross-platform and rule-based real-time provisioning, web-based self-service functions for users, delegated administration, request workflows, access certification, password management, metadirectory as well as auditing and reporting functionality.

DirX Directory

DirX Directory provides a standards-compliant, high-performance, highly available, highly reliable, highly scalable, and secure LDAP and X.500 Directory Server and LDAP Proxy with very high linear scalability. DirX Directory can serve as an identity store for employees, customers, partners, subscribers, and other IoT entities. It can also serve as a provisioning, access management and metadirectory repository, to provide a single point of access to the information within disparate and heterogeneous directories available in an enterprise network or cloud environment for user management and provisioning.

DirX Access

DirX Access is a comprehensive, cloud-ready, scalable, and highly available access management solution providing policy- and risk-based authentication, authorization based on XACML and federation for Web applications and services. DirX Access delivers single sign-on, versatile authentication including FIDO, identity federation based on SAML, OAuth and OpenID Connect, just-in-time provisioning, entitlement management and policy enforcement for applications and services in the cloud or on-premises.

DirX Audit

DirX Audit provides auditors, security compliance officers and audit administrators with analytical insight and transparency for identity and access. Based on historical identity data and recorded events from the identity and access management processes, DirX Audit allows answering the “what, when, where, who and why” questions of user access and entitlements. DirX Audit features historical views and reports on identity data, a graphical dashboard with drill-down into individual events, an analysis view for filtering, evaluating, correlating, and reviewing of identity-related events and job management for report generation.

For more information: support.dirx.solutions/about

Eviden is a registered trademark © Copyright 2026, Eviden SAS – All rights reserved.

Legal remarks

On the account of certain regional limitations of sales rights and service availability, we cannot guarantee that all products included in this document are available through the Eviden sales organization worldwide. Availability and packaging may vary by country and is subject to change without prior notice. Some/All of the features and products described herein may not be available locally. The information in this document contains general technical descriptions of specifications and options as well as standard and optional features which do not always have to be present in individual cases. Eviden reserves the right to modify the design, packaging, specifications and options described herein without prior notice. Please contact your local Eviden sales representative for the most current information. Note: Any technical data contained in this document may vary within defined tolerances. Original images always lose a certain amount of detail when reproduced.