EVIDEN

Identity and Access Management

DirX Identity

Integration Framework

Version 9.0.3, Edition October 2026

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

© 2026 Atos Group

All Rights Reserved

Distribution and reproduction not permitted without the consent of Atos Group.

Table of Contents

Copyright . . . ii
Preface . . . 1
DirX Identity Documentation Set . . . 2
Notation Conventions . . . 4
1. Connector Integration Framework . . . 6
1.1. Java Connector Integration Framework . . . 6
1.1.1. How the Java Connector Integration Framework Works . . . 7
1.1.2. Deploying the Framework . . . 8
1.1.2.1. Deploying the Framework as a Standalone Executable . . . 8
1.1.2.2. Deploying the Framework as a SOAP Client . . . 10
1.1.2.3. Deploying the Framework as a SOAP Service . . . 11
1.1.2.3.1. Configuring the SOAP Reader and Writer . . . 11
1.1.2.3.2. Setting Up the Web Deployment Descriptor . . . 12
1.1.2.3.3. Setting Up the Axis SOAP Deployment Descriptor . . . 13
1.1.2.4. Deploying a Connector in the IdS-J Server . . . 13
1.1.2.4.1. Component Description . . . 13
1.1.2.4.2. Deploying the JAR File . . . 15
1.1.3. About the Connector Interface . . . 15
1.1.3.1. About the DxmConnectorCore Interface . . . 16
1.1.3.2. About the DxmRequestor Interface . . . 16
1.1.3.3. About the DxmConnectorExtended Interface . . . 18
1.1.3.4. DirX Identity-Specific Conventions . . . 21
1.1.3.4.1. Attribute "customError" in SPML Response . . . 21
1.1.3.4.2. Operational Attributes in Search Request . . . 21
1.1.3.4.3. Requested Attributes in Search Request . . . 21
1.1.3.4.4. Paged Search . . . 22
1.1.3.4.5. Move Operations . . . 22
1.1.3.5. Using the Connector Interface . . . 22
1.1.3.5.1. Writing Log Entries . . . 23
1.1.3.5.2. Binary and Base64 Values . . . 25
1.1.3.5.3. Testing Request Transformations . . . 27
1.1.3.5.4. Testing the Connector Responses . . . 27
1.1.4. About the Filter Interface . . . 28
1.1.4.1. About the ConnectorFilter Interface . . . 29
1.1.4.2. About the ConnectorFilterConfig Interface . . . 30
1.1.5. Configuring the Framework . . . 30
1.1.5.1. Job . . . 32
1.1.5.2. Controller . . . 32
1.1.5.3. Operation . . . 33

1.1.5.4. User Hook . . . 33
1.1.5.5. Logging . . . 34
1.1.5.6. Port . . . 35
1.1.5.7. Filter . . . 35
1.1.5.8. Connector . . . 35
1.1.5.9. Connection . . . 36
1.1.5.10. requestTransformer . . . 37
1.1.5.11. responseTransformer . . . 38
1.1.5.12. mvProperty . . . 38
1.1.5.13. Property . . . 38
1.1.6. Configuring Default Controller Checkpointing . . . 39
1.1.7. Configuring Encryption Filters . . . 39
1.1.8. Configuring Connectors . . . 41
1.1.8.1. SPML SOAP Proxy . . . 41
1.1.8.2. LDIF Change Reader . . . 43
1.1.8.3. SPML File Reader . . . 44
1.1.8.4. SPML Loop Reader . . . 44
1.1.8.5. SPML File Writer . . . 45
1.1.8.6. LDIF File Writer . . . 46
1.1.8.7. Test Connector . . . 47
1.1.8.8. Test Writer . . . 48
1.2. C/C++ Connector Integration Framework . . . 48
1.2.1. How a C++ Connector Works . . . 49
1.2.2. About Connectors and Filters . . . 50
1.2.3. Using the C/C++ Connector API . . . 50
1.2.3.1. Creating and Compiling a Connector Library . . . 50
1.2.3.2. Implementing a Connector Class . . . 51
1.2.3.2.1. **Implementing a Filter Library** . . . 52
1.2.3.2.2. **Implementing a Filter Class** . . . 52
1.2.3.2.3. Loading and Initializing the Component Library . . . 53
1.2.3.3. About the Connector Configuration Class . . . 53
1.2.3.3.1. **About the Filter Configuration Class** . . . 54
1.2.3.4. Handling Errors and Exceptions in Connectors and Filters . . . 54
1.2.3.5. Logging Connector Messages . . . 55
1.2.3.6. About the Heap Check Mechanism . . . 55
1.2.4. About the ISR Classes . . . 55
1.2.4.1. About the Compound Classes . . . 56
1.2.4.2. About the SPML Classes . . . 57
1.2.4.3. Using ISR Classes . . . 58
1.2.4.4. Using Copy/Add/Set Methods . . . 59
1.2.4.4.1. **Using Transformation Methods** . . . 59
1.2.4.4.2. **Handling Exceptions** . . . 60

1.2.4.5. Handling Data Encoding . . . 60
1.2.5. About the Connector Server Configuration . . . 60
1.2.5.1. Server-Wide Configuration . . . 60
1.2.5.2. Logging Configuration . . . 61
1.2.5.3. Connector-Dependent Configuration . . . 61
1.2.5.4. C++ Identity Server INI File Configuration . . . 61
1.2.6. Miscellaneous Information . . . 62
1.2.6.1. Windows Platform Dependencies . . . 62
1.2.6.2. Relevant Files and their Function . . . 62
1.2.7. Sample Implementation . . . 63
2. Agent Integration Framework . . . 64
3. DirX Identity Web Application Programming Interface . . . 65
4. SPML Provisioning Web Services . . . 66
4.1. About the Provisioning Operations . . . 66
4.2. Authentication . . . 67
4.2.1. Proprietary Authentication . . . 67
4.2.2. SSO Header File Configuration . . . 69
4.2.3. XML Signature Verification Configuration . . . 71
4.3. Authorization . . . 73
4.4. Runtime Operation . . . 73
4.4.1. Deploying the Provisioning Servlet . . . 74
4.4.1.1. Understanding the Deployment Concept . . . 74
4.4.1.2. Deployment into Tomcat . . . 74
4.4.2. Configuring the Web Services Deployment . . . 75
4.4.3. Customizing the Runtime . . . 76
4.4.3.1. Providing a Service Subset . . . 76
4.4.3.1.1. Restricting Object Types . . . 78
4.4.3.1.2. Restricting Operations per Object Type . . . 78
4.4.3.2. Extending the LDAP Schema . . . 78
4.5. Common SPML Aspects . . . 79
4.5.1. Identifiers . . . 79
4.5.2. Identifying Attributes . . . 79
4.5.3. Add, Modify and Delete . . . 80
4.5.4. Search . . . 81
4.5.5. Iterate . . . 81
4.5.6. SPMLv2 Close Iterator . . . 81
4.6. User Management . . . 81
4.6.1. Add, Modify and Delete . . . 82
4.6.2. Attributes . . . 84
4.6.3. References . . . 84
4.6.4. Reference dxrRoleLink . . . 85
4.6.5. Reference dxrPermissionLink . . . 89

4.6.6. Reference dxrGroupLink . . . 91
4.6.7. Suspend Capability . . . 93
4.6.8. Password Capability . . . 94
4.7. Role Management . . . 94
4.7.1. Identifiers . . . 94
4.7.2. Attributes . . . 95
4.7.3. References . . . 95
4.7.3.1. Multivalued References . . . 96
4.7.4. Role Parameter Links and Match Rules . . . 97
4.7.5. Delete . . . 98
4.8. Permission Management . . . 98
4.8.1. Attributes . . . 99
4.8.2. Add, Modify and Delete . . . 99
Modify . . . 100
4.8.3. Permission Match Rules . . . 102
4.9. Business Objects Management . . . 103
4.9.1. Organization Management . . . 104
4.9.2. Organizational Unit Management . . . 104
4.9.3. Context Management . . . 104
4.9.4. Location Management . . . 104
4.9.5. Cost Unit Management . . . 105
4.10. Creating and Deleting Target Systems . . . 105
4.10.1. Identifiers . . . 106
4.10.2. Attributes . . . 106
4.10.3. Connection Options . . . 106
4.10.4. Environment Properties . . . 107
4.10.5. Options . . . 108
4.10.6. Add . . . 108
4.10.7. Modify . . . 109
4.10.8. Delete . . . 110
4.10.9. Search . . . 110
4.11. Accounts Management . . . 110
4.11.1. Attributes . . . 111
4.11.2. Reference dxrUserlink . . . 111
4.11.3. Reference dxrUsedBy . . . 111
4.11.4. Reference dxrPwdPolicyLink . . . 111
4.11.5. Reference dxrGroupMember . . . 111
4.11.6. Add, Modify and Delete . . . 112
4.11.7. Lookup and Search . . . 113
4.11.8. Suspend Capability . . . 113
4.11.9. Password Capability . . . 114
4.11.9.1. Authentication by User Certificate . . . 114

4.11.9.2. Authentication by Challenge/Response . . . 116
4.11.9.3. Authentication by One-Time Password . . . 116
4.11.10. Async Capability - Status Request . . . 116
4.11.11. Challenge/Response and Password Policy . . . 117
4.11.11.1. getChallenges Operation . . . 117
4.11.11.2. checkChallengeResponses Operation . . . 118
4.11.11.3. getPasswordPolicy Operation . . . 119
4.12. One-Time Password . . . 119
4.12.1. sendOtp Operation . . . 120
4.12.2. setPassword Operation . . . 121
4.12.3. Custom Error Messages . . . 122
4.13. Groups Management . . . 122
4.13.1. Identifiers . . . 123
4.13.2. Attributes . . . 123
4.13.3. References . . . 123
4.13.4. Add, Modify and Delete . . . 124
4.13.5. Lookup and Search . . . 124
4.14. Custom Objects . . . 124
4.14.1. Custom Objects Custom Context . . . 125
4.14.1.1. CustomReferenceHandler Bean . . . 125
4.14.1.2. Object Type Meta Data Beans . . . 126
4.14.1.3. BasicCustomHandler Bean . . . 126
4.14.1.4. Operation Handler Beans . . . 127
4.14.2. Custom Objects ListTargets . . . 128
4.14.3. Application Context . . . 129
4.14.3.1. ListTargetsHandler . . . 129
4.14.3.2. AddDispatcher . . . 130
4.14.4. Custom Objects Operations . . . 130
4.14.4.1. Add . . . 130
4.14.4.2. Modify . . . 131
4.14.4.3. Delete . . . 131
4.14.4.4. Lookup and Search . . . 131
4.14.4.5. Attributes . . . 131
4.14.4.6. Capabilities . . . 131
4.15. User Hooks . . . 131
4.15.1. User Hook Interface . . . 131
4.15.2. User Hook Configuration . . . 132
4.16. Using the Sample Client . . . 133
4.16.1. Getting Started with the Test Client . . . 133
4.16.2. Testing the Sample Client Functionality . . . 135
4.16.3. Generating a Client from the WSDL . . . 136
4.16.4. Using the Sample SOAP Connector . . . 137

4.16.4.1. Providing Configuration Data . . . 137
5. Workflow Management Web Services . . . 139
5.1. Operations . . . 139
5.1.1. List Definitions . . . 140
5.1.2. Create Instance . . . 141
5.1.3. Get Instance . . . 142
5.1.4. List Instances . . . 142
5.1.5. Get Worklist . . . 145
5.1.6. Modify Instance . . . 146
5.1.7. Suspend . . . 149
5.1.8. Resume . . . 149
5.1.9. Abort . . . 149
5.1.10. Activate . . . 150
5.1.11. Verify . . . 150
5.1.12. Wait for Idle . . . 151
5.1.13. Notify Participant Changed . . . 152
5.1.14. Update Certification . . . 152
5.1.15. Bulk Approval . . . 153
5.2. Implementing a Client . . . 154
5.3. Authentication . . . 157
5.4. Common Parameters . . . 158
Legal Remarks . . . 160

Preface

The *DirX Identity Integration Framework* describes a set of services and interfaces that allow extending DirX Identity It consists of the following chapters:

- ch1_connector.pdf provides information about the Connector Integration Framework, which enables customers to build their own Java agents or connectors as well as C or C++-based connectors to connect additional (legacy) target systems.
- ch2_agent.pdf introduces the agent integration framework, which allows integrating existing executables or batch files to run under the control of the DirX Identity C++-based Server.
- ch3_webAPI.pdf introduces the Identity Web API. Use this interface to the DirX Identity Services to extend a Web Center application with additional functionality.
- ch4_WebServices.pdf introduces the DirX Identity SPML Provisioning Web Services, which offer a wide range of operations to provision users, privileges, target system objects and generic LDAP objects to a DirX Identity domain.
- ch5_WFWebServ.pdf provides information about DirX Identity Workflow Management Web Services for managing request workflow instances, which allow clients to create a workflow instance, read, change, suspend and resume it.

DirX Identity Documentation Set

Version: 9.0.3 | Date: 2026-10-20 | Timestamp: 12:29:33 UTC

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 Resolution Service Setup Guide*. Use this book to set up the DirX Identity Resolution Service as a standalone service.
- *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.
- *DirX Identity REST Service 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. Connector Integration Framework

The DirX Identity Connector Integration Framework has two API-dependent variants:

- The Java Connector Integration Framework, which permits you to create connectors and agents that integrate target systems with Java APIs.
- The C/C++ Connector Integration Framework, which allows you to create connectors that integrate target systems with C/C++ APIs.

The sections in this chapter describe each framework's elements and how to configure and deploy it.

1.1. Java Connector Integration Framework

The DirX Identity Java Connector Integration Framework helps developers build DirX Identity agents and connectors for new types of target systems with a minimum of effort.Written in Java, it includes:

- Builder components to read the configuration data and transform it to Java objects and build the stack to process a DirX Identity job
- Reader classes to read input data from files or servlet requests in Services Provisioning Markup Language (SPML) or LDAP Data Interchange (LDIF) format
- Writer classes to output response data to files or servlet response in SPML or LDIF format
- Transformation classes to transform input data or output data from SPML format with Extensible Stylesheet Language (XSL) transformations
- A connector interface to incorporate third-party connector implementations
- Controller classes to run a DirX Identity job either as standalone executable or as a servlet and control the correct invocations of readers, transformers, response writers and the connector according the job configuration

In order to support the implementation of connectors for RESTful services the framework provides a template RESTful connector along with special interfaces and their simple and/or sample implementations. For more details on that see the corresponding section in the Use Case document *Java Programming*.

The information in the following sections serves two audiences:

- Administrators who want to configure DirX Identity jobs based on framework implementations. These readers need to know about the framework's deployment and configuration.
- Developers who want to implement new agents based on the framework. These readers need to know about the framework's deployment and configuration and also about the connector interface and how to use it.

1.1.1. How the Java Connector Integration Framework Works

The following figure shows the main components of the framework:

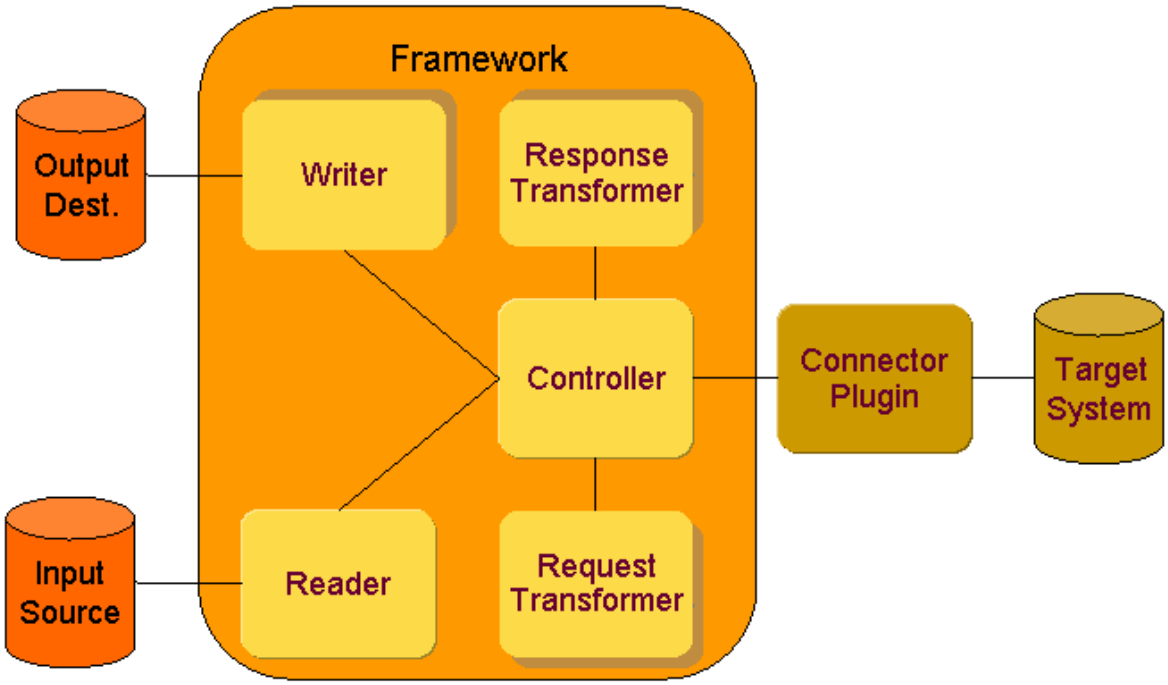

Figure 1. Java Connector Integration Framework Components

As illustrated in the figure:

- A reader reads a request from an input source. In most cases, the input source is a file - for example, an LDIF file or an SPML file - but it can also be a Java Message Service (JMS) message or an SPML request over Simple Object Access Protocol (SOAP).
- The reader passes the input data to the controller as Java objects logically compliant with SPML format.
- The controller invokes one or more read transformers to transform the input and pass the results to the connector.
- The connector operates the request against the target system and returns an SPML response, which may be transformed by one or more transformers before it is written to output destinations by writers.
- Within the framework, the data is represented as Java objects that represent SPML requests and responses. Transformers may also decrypt and encrypt the data. The connector always handles data in clear text format.
- All the participating classes, data source and destinations and operation options are configured in an XML-formatted file. On startup, this file is converted to Java objects via

a mapping file.

1.1.2. Deploying the Framework

The Java Connector Integration Framework can be deployed as a standalone executable, as a Simple Object Access Protocol (SOAP) client, or as a SOAP service in a Web container. The next sections describe how to deploy the framework in each of these environments.

1.1.2.1. Deploying the Framework as a Standalone Executable

The Java Connector Integration Framework is typically deployed as a standalone executable like the agents for batch workflows in DirX Identity. In this type of deployment, the framework executable is started by the C++-based DirX Identity server, reads its input requests from a file and writes the response to another file. The following figure shows a typical import/export deployment, where the framework acts as an executable that cooperates with the DirX Identity meta controller (**metacp**).

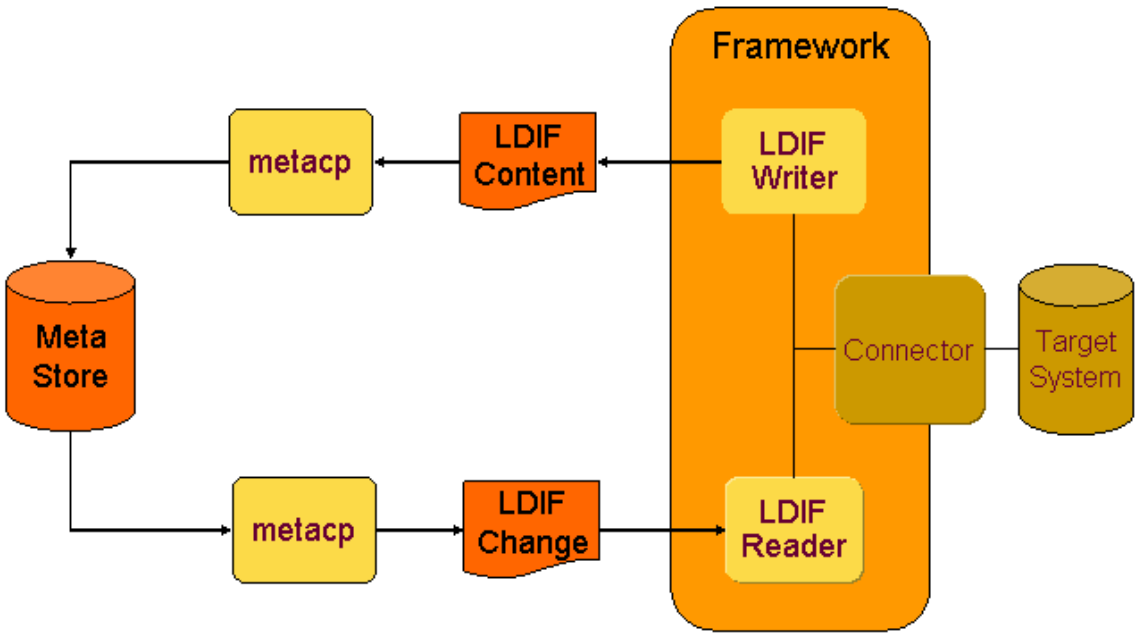

Figure 2. Import/Export Framework Deployment

In this deployment, **metacp** exports data from the identity store (an LDAP directory) and transforms it to LDIF change format. An LDIF reader within the framework converts the data to SPML requests. The connector carries out the request at the target system and returns an SPML response, which LDIF writer converts to LDIF content format. **metacp** then transforms the response and writes it back to the identity store. Note that

transformation can occur within **metacp** and also within the framework in both the input and output direction. However, it is **metacp** that determines the requests: add, modify or delete an entry.

A pure framework export scenario is slightly different, and is shown in the following figure.

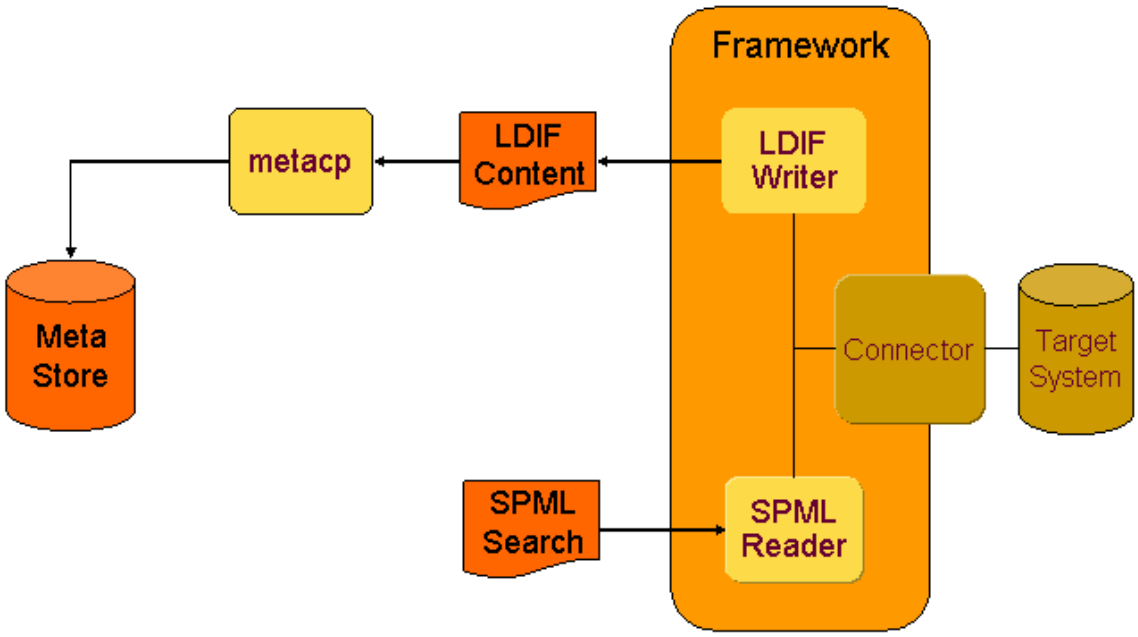

Figure 3. Pure Framework Export Deployment

In this scenario, the administrator prepares an SPML request document that contains just a search request. The framework's SPML reader passes this request to the connector, which presents it to the target system. The rest of the procedure operates just as the import/export scenario.

The following command line invokes the framework as an executable:

java -**cp classpath siemens.dxm.connector.framework.AgtSessionExe -c** *confFilename* **-p** *programName* **-v** *version* **-d** *buildDate* **-Enc** \{**ATTRIB_ADMIN_PW** | **ADMIB_PW** | **NONE**} **-Timeout** *timeout* **-Port** *port* -**CryptlogLevel** *cryptloglevel* **-AuditLevel** *auditlevel*

where

- **-c** is the name of the configuration file (mandatory).
- **-p** indicates the agent's program name.
- **-v** indicates the agent's version.

- **-d** indicates the agent's build date.
- **-Enc** is the encryption level (for details, see the DirX Identity documentation); the default is **NONE**.
- **-timeout** is the maximum time the agent is to wait to receive the key from DirX Identity server.
- **-port** is the port number for communication with DirX Identity server.
- **-cryptlogLevel** indicates the log level for encryption.
- **-auditLevel** indicates the level of writing audit logs.

> Only the **-c** option is mandatory. The **-Enc** option is only mandatory if the input data or the configuration file contains encrypted attribute values; for example, passwords.

1.1.2.2. Deploying the Framework as a SOAP Client

The standalone executable framework deployment can itself be deployed as a SOAP client that sends SPML SOAP requests over HTTP and receives SPML SOAP responses from a SOAP service. To deploy the framework in this way, just select the appropriate class as the connector. You have two options:

- DxaSpmlSoapProxy: produces SPML requests according the latest OASIS SPMLv1 standard (http://www.oasis-open.org/committees/download.php/4138/os-pstc-spml-schema-1.0.xsd).
- SpmlSoapProxy: produces SPML requests according a more recent version of SPMLv1 that is not considered the official release any more (http://www.oasis-open.org/committees/download.php/2396/cs-pstc-spml-schema-1.0.xsd).

The following snippet shows a configuration with the latest connector:

```
<connector
 role="connector"
 className="siemens.dxm.connector.framework.soap.DxaSpmlSoapProxy"
 name="SPML connector">
 <connection type="SOAP"

url="http://localhost:8080/spmlsoapservice/services/SpmlSoapService"
 >
 </connection>
</connector>
```

The proxy class sends the SPML requests to the URL specified in the connection section of this configuration. It must be adjusted to the configuration of the server side. See "Deploying the Framework as a SOAP Service" for an example.

1.1.2.3. Deploying the Framework as a SOAP Service

The framework can also be deployed as a SOAP service within a Web container. When deployed as a Web service, the framework receives SPML SOAP requests via HTTP and returns SPML SOAP responses, as shown in the following figure.

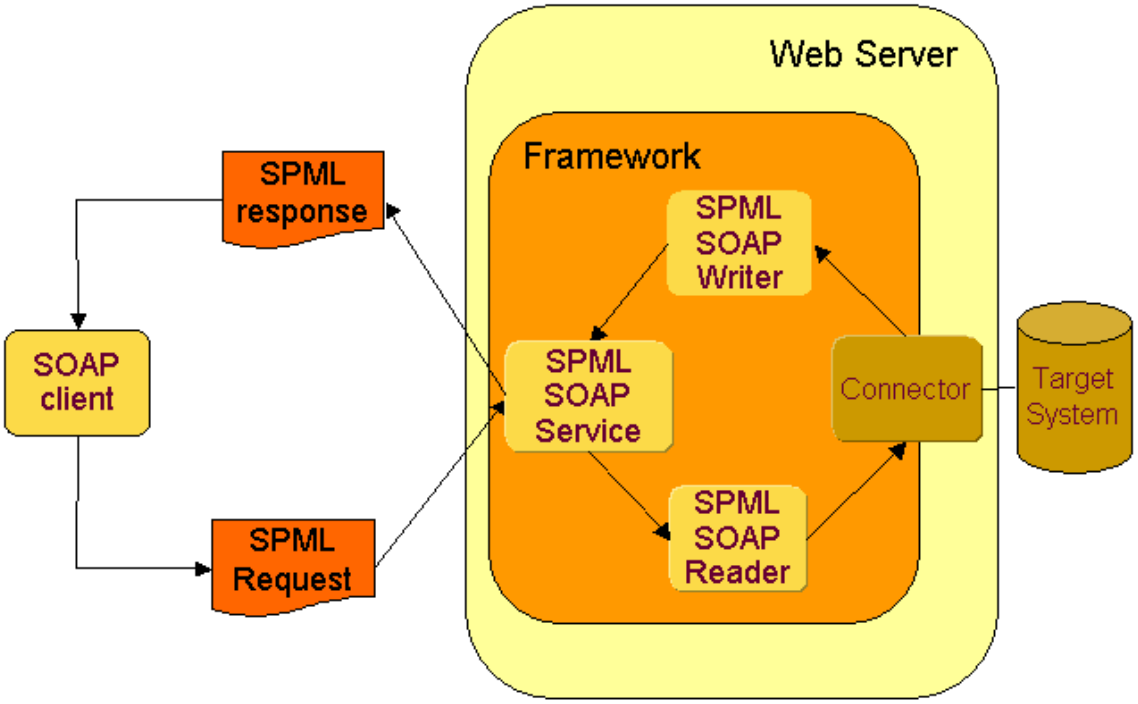

Figure 4. Framework Deployment as SOAP Service

1.1.2.3.1. Configuring the SOAP Reader and Writer

You need to specify a SOAP reader and a SOAP writer in the framework configuration file to receive the requests and send the responses. The following configuration snippet shows the important sections:

```
<job>
<connector
    name="Default Controller" version="0.1"
    role="controller"

className="siemens.dxm.connector.framework.DefaultControllerService">
    <logging ... />
</connector>
```

```
<connector
    role="reader"
    name="SPML SOAP reader"
    className="siemens.dxm.connector.framework.soap.SpmlSoapReader">
</connector>
<connector
    role="connector"
    className="your connector class name"
    name="name your connector">
    <connection ... />
</connector>
<connector
    role="responseWriter"
    name="SPML SOAP writer"
    className="siemens.dxm.connector.framework.soap.SpmlSoapWriter">
</connector>
</job>
```

Make sure you select a controller class that implements the DxmControllerService interface; for example, the DefaultControllerService of the framework.

1.1.2.3.2. Setting Up the Web Deployment Descriptor

A typical Web deployment descriptor for a web server might look like this:

```
<web-app>
<servlet>
  <servlet-name>SpmlSoapService</servlet-name>
  <servlet-
class>siemens.dxm.connector.framework.soap.AgtSessionServlet
  </servlet-class>
  <init-param>
    <param-name>confFileName</param-name>
    <param-value>confService.xml</param-value>
  </init-param>
    <init-param>
      <param-name>mappingFilename</param-name>
      <param-value>agentconf.xml</param-value>
    </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

```
<servlet-mapping>
<servlet-name>spmlsoapservice</servlet-name>
<url-pattern>/spmlsoapservice</url-pattern>
</servlet-mapping>
</web-app>
```

It defines a servlet named SpmlSoapService and the necessary initialization parameters that provide the names of the configuration and the mapping file (as absolute path names).

1.1.2.3.3. Setting Up the Axis SOAP Deployment Descriptor

The SOAP service uses the Axis framework as its SOAP engine, so you need to add the service to the Axis server's Web service deployment descriptor, for example:

```
<deployment …>
<service name="SpmlSoapService"
provider="java:MSG" style="message" use="literal" streaming="on">
 <parameter name="allowedMethods" value="spmlRequest"/>
<parameter name="className"
value="siemens.dxm.connector.framework.soap.SpmlSoapService"/>
</service>
</deployment>
```

This step defines the SOAP service's class name and the methods to be called. The method spmlRequest() within the class SpmlSoapService handles the incoming SOAP requests. It passes them to the SpmlSoapReader and via the framework controller to the class you specified as the connector. The connector's response is passed to the SpmlSoapWriter class, which produces the SOAP response message.

1.1.2.4. Deploying a Connector in the IdS-J Server

A connector implementation can be deployed as a standalone agent (see Framework as Executable) and in the IdS-J server. In the server it can run as part of a workflow in a number of threads at the same time. This requires the connector implementation to be thread-safe. This means especially to be careful with static variables and in the open() and close() methods. Especially keep in mind that this method pair could be called multiple times within a job!

1.1.2.4.1. Component Description

In order to select a connector for an activity, you need to supply a **component description**. In the configuration store, component descriptions of connectors have to be stored beneath the "*Configuration/Connector Types*" folder. Here you find a sub-tree for each type of connector. Create a new folder for your connector; for example, "*Sample*". Take one appropriate component description as a template, copy it and place it into the sub-folder

"*Component Descriptions*". The component type should be "*connector*".

The content is an XML document with the root element *<componentDescription>*. Its name attribute is important: In the configuration of an activity port, it is listed when selecting a connector. The Identity Manager stores the selected value in the port's XML content. Then it serves as the only reference from the port configuration to the component description.

The component description defines the configuration properties of your connector and how they are presented within the job configuration. It has to display the fields for the connector configuration parameters, at least the ones with may be changed. The component description is a merge between XML schema and DirX Identity object descriptions. It describes a XML structure with XML elements and attributes and <property> elements. The basic structure is the same for all connectors: it conforms to the connector configuration described in Configuration.

Important for your connector are the properties. You have three alternatives for setting them:

Set a default value within the component description

This can be done with the <value> element beneath the <property> as in this sample for the classname:

```
<property   name="className" mandatory="true"
xmlNotation="xmlAttribute" type="java.lang.String">
      <value>siemens.dxm.connector.ldap.LdapConnector</value>
</property>
```

xmlNotation="..." tells DirX Identity that this property is to be stored as an XML attribute.

Take the value from an LDAP attribute of the same or a referenced entry:

As with a lot of others, parameter values can be taken at load time from LDAP entries, either the port entry or one, which is referenced from it. See the following sample, which takes the user name from a referenced bind profile:

```
<value>${DN4ID(THIS)@dxmBindProfile-DN@dxmUser}</value>
```

The term "${DN4ID(THIS)" denotes the current LDAP entry. Each "@<attributetype>" takes the value from the LDAP attribute. If it is followed by another "@..." expression, it is interpreted as DN and DirX Identity reads the referenced LDAP entry and the next attribute.

Let the administrator set the value with DirX Identity Manager

This alternative requires displaying the property in the GUI. This is accomplished similarly, but not exactly the same way as with object descriptions. In contrast to an

object description, presentation is configured in separate elements: <presentationDescription>/<propertyDescriptions>/<property>. This <property> element must refer via its name attribute to a <property>, which has already been defined beneath an <element>, either for a <connector> or for the <connection>. The following property definition refers to the bind profile and defines the label and the editor to be used (if not standard string type):

```
<property   name="dxmBindProfile-DN"
editor="siemens.dxm.storage.beans.JnbReference"
editorparams="showpath">
     <label>Bind Profile</label>
</property>
```

1.1.2.4.2. Deploying the JAR File

The jar file that contains your connector implementation must be copied to a classpath that the IdS-J server searches when processing framework-based workflows. Copy your jar file to the system file folder *install_path*/**ids-j-***domain***-S***n***/confdb/framework/lib**. Make sure it is not deleted after upgrade installations.

1.1.3. About the Connector Interface

The connector interface marks the interface between the DirX Identity agent framework and the connector components that provision target systems. It reflects the requests of the SPMLv1 standard.

The interface consists of mandatory and optional pieces. The mandatory interfaces are:

DxmConnectorCore
: This interface consists of the methods open() and close() to read configuration data, open and close the connection to the target system.

DxmRequestor
: This interface contains the operations to read, update and delete entries in the target system: add(), modify(), delete() and search().

The optional DxmConnectorExtended interface adds the rest of the SPMLv1 requests:

- getSchema()
- status()
- cancel()
- extendedRequest()
- batchRequest() to deliver a collection of requests at one time

For more information on the interface and the input and return parameter classes, see the Java documentation provided on your DVD in the folder:

1.1.3.1. About the DxmConnectorCore Interface

The interface siemens.dxm.connector.DxmConnectorCore includes the mandatory methods to read the configuration and open and close the connection to the target system:

void open(DxmConnectorConfig *connectorConfig*)

Call the open() method after the implementing class has been instantiated. Other methods in the interface cannot be called until this method successfully returns.

The method opens a connection to the target system (called a provisioning service target (PST) in SPML). The framework delivers the configuration options in the connectorConfig parameter of type siemens.dxm.configuration.job.DxmConnectorConfig. It contains all the information that was configured in the connector element with role="connector". The most important ones are contained in the connection sub-element with address and bind credentials to access the target system.

Connector developers do not need to handle parsing the XML document: the framework handles this task and creates a Java class that corresponds one-to-one to the configuration. The attributes of the connector XML element are properties of the Connector class. You can read them by issuing a getProperty(*"optionname"*) call. The connection sub-elements are themselves a collection of Java classes, which you can read by issuing a getConnections() call.

The following code snippet shows how to access standard and custom connection options:

```
DxmConnectionConfig connection =
(DxmConnectionConfig)
connectorConfig.getConnections().firstElement();

String rspFilename = connection.getFilename();

// how to read custom configuration properties:
String dbgFilename = (String)connection.getProperty("debugfile");
void close()
```

void close()

Call the close() method to shut down the connection to the target system and release all resources. After this method is called, only an open() call is allowed.

1.1.3.2. About the DxmRequestor Interface

The interface siemens.dxm.connector.DxmRequestor includes the methods to read, update

and delete an entry:

AddResponse add(AddRequest *req*)

Call this method to perform an add request and return a SPML AddResponse. The AddRequest and AddResponse are both Java classes that conform to the SPML specification.

The following code snippet shows how to read the requestid, the identification and the attributes from the AddRequest class:

```
String requestid = req.getRequestID();
Identifier identifier = req.getIdentifier();
Attributes attributes = req.getAttributes();
```

For more details on AddRequest and AddResponse, see the Java documentation.

ModifyResponse modify(ModifyRequest *req*)

Call this method to perform a modify request and return a SPML ModifyResponse. The ModifyRequest and ModifyResponse are both Java classes that conform to the SPML specification.

The following code snippet shows how to read the requestid, the identification and the modifications from the ModifyRequest class:

```
String requestid = req.getRequestID();
Identifier identifier = req.getIdentifier();
Modifications attributes = req.getModifications();
DsmlModification[] = Modifications.getModification();
```

For more details on ModifyRequest and ModifyResponse, see the Java documentation.

DeleteResponse delete(DeleteRequest *req*)

Call this method to perform a delete request and return a SPML DeleteResponse. The DeleteRequest and DeleteResponse are both Java classes that conform to the SPML specification.

The following code snippet shows how to read the requestid and the identification from the DeleteRequest class:

String requestid = req.getRequestID();
Identifier identifier = req.getIdentifier();

For more details on DeleteRequest and DeleteResponse, see the Java documentation.

SearchResponse search(SearchRequest *req*)

Call this method to perform a search request and return a SPML SearchResponse . The

SearchRequest and SearchResponse are both Java classes that conform to the SPML specification.

The following code snippet shows how to read the requestid, the search base, the search filter and the requested attributes from the SearchRequest class:

```
String requestid = req.getRequestID();
Identifier identifier = req.getSearchBase();
Filter filter = req.getFilter();
AttributeDescriptions attrDesc[] = req.getAttributes();
```

For more details on SearchRequest and SearchResponse, see the Java documentation.

1.1.3.3. About the DxmConnectorExtended Interface

The interface siemens.dxm.connector.DxmConnectorExtended extends the DxmConnectorCore interface and includes the additional optional methods of the DirX Identity connector interface:

SchemaResponse getSchema(SchemaRequest *req*)

Call this method to return information from the target system schema. The SchemaRequest and SchemaResponse are both Java classes that conform to the SPML specification.

The following code snippet shows how to read the requestid, the optional identifications of the provider and the schema from the SchemaRequest:

```
String requestid = req.getRequestID();

//read providerID as string
String provider = (String)
req.getProviderIdentifier().getProviderID();

//read operationID as string
String operation = (String)
req.getOperationIdentifier().getOperationID();

SchemaIdentifier schemaid = req.getSchemaIdentifier();
```

The next snippet suggests how to build a schema and a schema response:

```
SchemaResponse rsp = new SchemaResponse();
rsp.setRequestID(requestid);
rsp.setResult(ResultCode.VALUE_0);  //success
```

```
Schema schema = new Schema();
schema.setProviderIdentifier(providerid);
schema.setSchemaIdentifier(schemaid);
AttributeDefinition attrDef = new AttributeDefinition();
attrDef.setName("someAttributeType");
schema.addAttributeDefinition(attrDef);
AttributeDefinitionReference attrDefRef = new
AttributeDefinitionReference();
attrDefRef.setName("someAttributeType");
attrDefRef.setRequired(false);
AttributeDefinitionReferences attrRefs = new
AttributeDefinitionReferences();
attrRefs.addAttributeDefinitionReference(attrDefRef);
ObjectClassDefinition objDef = new ObjectClassDefinition();
objDef.setName("someObjectClass");
objDef.setMemberAttributes(attrRefs);
schema.addObjectClassDefinition(objDef);
rsp.addSchema(schema);
```

For more details on SchemaRequest and SchemaResponse, see the Java documentation.

StatusResponse status(StatusRequest *req*)

When SPML operations are being executed asynchronously, call this method to query the status of a request. The connector may also return results of requests being processed, especially in case of long search result sets. The StatusRequest and StatusResponse are both Java classes that conform to the SPML specification.

The following code snippet shows how to read the requestid from the StatusRequest and set the result code "pending" in the response:

```
String requestid = req.getRequestID();
StatusResponse rsp = new StatusResponse();
rsp.setRequestID(requestid);
rsp.setResult(ResultCode.VALUE_2);    //pending
```

For more details on the StatusRequest and StatusResponse, see the Java documentation.

CancelResponse cancel(CancelRequest *req*)

When SPML operations are being executed asynchronously, call this method to cancel the request. The CancelRequest and CancelResponse are both Java classes that conform to the SPML specification.

The following code snippet shows how to read the requestid from the CancelRequest and to set the result type of the response:

```
String requestid = req.getRequestID();
CancelResponse rsp = new CancelResponse();
rsp.setRequestID(requestid);
rsp.setResult(ResultCode.VALUE_1);
rsp.setCancelResult(CancelResultsType.VALUE_1);   //cancelled
```

For more details on CancelRequest and CancelResponse, see the Java documentation.

ExtendedResponse extendedRequest(ExtendedRequest *req*)

Call this method to request services that are not specifically defined in SPML. The ExtendedRequest and ExtendedResponse are both Java classes that conform to the SPML specification.

The following code snippet shows how to read the requestid, the mandatory identifications of the provider and the operation from the ExtendedRequest :

```
String requestid = req.getRequestID();

//read providerID as string
String provider = (String)
req.getProviderIdentifier().getProviderID();

//read operationID as string
String operation = (String)
req.getOperationIdentifier().getOperationID();
```

The ExtendedResponse may contain one Attributes collection including a number of Attribute objects (as in the AddResponse).

For more details on ExtendedRequest and ExtendedResponse, see the Java documentation.

BatchResponse batchRequest(BatchRequest *req*)

Call this method to collect a number of simple requests: just AddRequest, ModifyRequest, DeleteRequest and ExtendedRequest.

When the framework encounters a batchRequest, it usually splits it up and passes the individual requests to the connector. A future operational flag will advise the framework to transfer the BatchRequest as a whole to the connector.

1.1.3.4. DirX Identity-Specific Conventions

DirX Identity defines some conventions and extensions that a connector implementation should follow. The next sections explain these items.

1.1.3.4.1. Attribute "customError" in SPML Response

The DirX Identity connector interface has extended the SPML response with the optional attribute "customError", which allows additional information to be given about the error. The following values are allowed:

FAILED.TEMPORARY: a temporary error. The request can be repeated after some waiting period.

OBJECT_ALREADY_EXISTS: the entry to be added already exists.

NO_SUCH_OBJECT: the requested entry does not exist.

NO_SUCH_ATTRIBUTE: the requested attribute or attribute to be added or updated does not exist.

ATTRIBUTE_OR_VALUE_EXISTS: the attribute or value to be added already exists.

1.1.3.4.2. Operational Attributes in Search Request

scope

The scope attribute is interpreted the same way as in LDAP requests. The following values are accepted: subtree, base, onelevel.

Here is a sample snippet of a search request:

```
<attr name="scope">
<value>subtree</value>
</attr>
```

Connectors are highly recommended to support the values "subtree" and "base". Especially, the "base" value is set automatically by the join engine in order to read a single object only given its identifier.

sizeLimit

The maximum number of entries to be returned.

timeLimit

The maximum waiting time (seconds) for the search.

1.1.3.4.3. Requested Attributes in Search Request

The <attributes> section of a search request specifies the attributes to be returned in the search result entries. SPML makes no statements on how to express that all or no attributes have to be returned. DirX Identity supports the following convention:

If no requested attributes are given, the connector is expected to return ALL attributes for compliance with the LDAP specification.

If no attributes must be returned, the operational attribute "noattr" has to be supplied, as shown in the following sample:

```
<attr name="noattrs">
<value>true</value>
</attr>
```

1.1.3.4.4. Paged Search

The OASIS SPMLv1 standard does not cover paged or iterative searches. In order to support large numbers of entries, DirX Identity uses the following operational attributes in Search Requests:

- pageSize: To be interpreted as an Integer; denotes the maximum number of entries returned per page.
- sortAttribute: the attribute type, which determines the sequence of entries in the search result.
- sortOrder: determines the sort order. Allowed values are: ASCENDING, DESCENDING.

Although the join engine might require the connector to perform a paged search, it expects one search response through which it can iterate. That means, it’s up to the connector to fill the gap: Build a response, fill with the first result page, fetch the next page, when the join engine has read all entries of the first page and iterates to the next result entry.

1.1.3.4.5. Move Operations

The SPML standard does not specifically describe how to request rename operations or a move in a hierarchical system such as LDAP. DirX Identity uses the following proprietary approach:

The operational attribute "dxrPrimaryKeyOld" contains the current identifier of the entry to be modified. If it does not match the identifier of the modify request, the identifier has been changed and the entry has to be renamed or moved.

1.1.3.5. Using the Connector Interface

This section provides information about how to use the connector interface that may be valuable for developers of connectors or transformers, including how to:

- Write log entries
- Handle string and binary attribute values
- Test request transformers
- Test connector responses

DirX Identity also provides the Java source for a sample connector that you can use as a template. The sample connector contains examples for logging and reading or writing attribute values and shows how to walk through the attribute lists (as used in Add requests and Search responses), modification lists and search filters. The sample source code is available on your DVD in the folder:

Additions\SampleConnector\java

The following files are available:

- **SampleConnector.java** - sample connector source
- **MsgID.java** - message codes for logging
- **sample.properties** - log messages in resource file

1.1.3.5.1. Writing Log Entries

The framework provides a log support that allows each connector and other framework components to write log entries in different deployment environments.

As a first prerequisite, each class must be in a package that starts with **siemens.** or **com.siemens.**, instantiate a logger and provide its own class as parameter, as shown here for the SampleConnector class:

```
private static final LogSupport logger =
LogSupport.forName(SampleConnector.class);
```

The logger provides a list of log methods for writing log entries.

The simplest way is to use the same methods that are provided in standard logging frameworks such as log4j and Java utilities: error, warn, info, finest. The logger prefixes the log message with a standard string specific for the log level. For example a

```
logger.debug("My debug message");
```

is written to file with the prefix "DBG(STG100): My debug message".

Equivalent to this is calling the log() method with the log level as first parameter followed by the message string, as shown in the following example:

```
logger.log(Level.INFO, _classname + ": No response found for request
with id: " + req.getRequestID());
```

The logger supports the following log levels (see the Level class), ranging from lowest to highest. The following table shows the level name as defined in the Level class, the corresponding number to be used in configuration, the corresponding message prefix and

some explanatory text:

Level	Number in Configuration	Message Prefix	Comment
FATAL	1	FTL	Errors that prevent the component from continuing its processing.
SEVERE, ERROR	1	ERR	For serious error messages.
WARNING	2	WAR	Error messages that should be investigated by the administrator, but usually do not disturb other requests.
INFO	5	LOG	Normal information log messages.
CONFIG	6	CON	Next higher level, usually to be used for configuration logs.
FINE	7	FIN	
FINER	8	FNR	
FINEST, DEBUG	9	DBG	Highest log level, just for debug messages.

If the configured log level is lower than the level parameter in the log method, the logger ignores the request and does not write a log entry. For example, if log level INFO is configured (<logging level="5" .../>), log messages for level DEBUG are ignored.

Producing debug messages is often very time-consuming. You can avoid this problem by first consulting the configured log level:

```
if (logger.isDebugEnabled()) {
      logger.log(Level.DEBUG, "some long message");
}
```

Instead of providing the log level as a separate parameter, you can indicate the level with a message prefix. See the previous table for corresponding levels and prefixes. The following method calls are equivalent:

```
logger.log(Level.DEBUG, "some long message");
logger.log("DBG: some long message");
```

In order to enable internationalization, it is recommended to hold message texts in resource files. This is also supported by the logging framework. Provide a message ID as first parameter and then either an object array or a list of up to 5 objects, as shown in the following sample:

```
logger.log(MsgID.LOG_CONN_NORSP, _classname,
req.getClass().getSimpleName(), req.getRequestID());
```

This requires a class MsgID, which holds the static message id fields:

```
static final String LOG_CONN_NORSP = "LOG_CONN_NORSP";
```

and a resource file with the message templates containing the placeholders:

```
LOG_CONN_NORSP=LOG:CONN003:{0}: No response found for request {1}
with id: {2}.
LOG_CONN_NORSP.explanation=Add missing response to response file!
```

The resource file should be located into the classpath, at best as part of the produced jar file. As an example, the resource file for the class siemens.dxm.connector.sample.SampleConnector is named sample.properties and located in the folder (package) "siemens.dxm.connector". In your connector class you have to register the file at the ResourceManager as in this sample:

```
ResourceManager.getDefault().addResource("siemens.dxm.connector.sampl
e");
```

If you need the message string also in exceptions or in the <errormessage> element of an SPML response, you can call the message formatter to get the localized message string, as shown here:

```
String msg = logger.getMessage(new MsgFormatter(), MsgID.DBG, new
Object[] {e.getLocalizedMessage()});
logger.log(MsgID.ERR_CONN_UNMARSHAL_EX, msg);
throw new DxmConnectorException(msg);
```

For a full sample see the **Additions/SampleConnector/java** folder of your installation DVD.

1.1.3.5.2. Binary and Base64 Values

SPML attribute values can either be strings or binary values. SPML refers to the Directory Services Markup Language (DSML) v2 standard, which allows a union of XML types: xsd:string, xsd:base64Binary and xsd:anyURI. Attributes are used in a number of Java objects, the most prominent of which are AddRequest, ModifyRequest and SearchResultEntry.

This section provides some hints on how to set and get string and binary values. The Java class that represents them is DsmlValue.

If you are sure that your values are always strings, simply call the getContent() and setContent(...) methods of the DsmlValue class. Don't worry about marshalling them to or from XML strings. Escaping and un-escaping of XML markup characters like "&" and "<" is implicitly performed by the class itself.

If you have binary values, you must base64-encode them prior to marshalling them as XML. The convenience method getDsmlValueFromBinary() creates a DsmlValue object from a byte array:

```
byte[] bytearray;
// ... set your array
DsmlValue dval = getDsmlValueFromBinary(bytearray);
```

The method encodes the byte array to a base64 string and sets the value type to 'base64'.

If you receive a DsmlValue and don't know the type, first check the type and then get either the string value or the byte array:

```
String val;
if (DsmlValue.BASE64BINARY_TYPE.equals(dval.getMemberType())) {
// value is base64 encoded: get the binary value
    bytearray = dval.getBinvalue();
}
else {
    // string or uri type
    val = dval.getContent();
}
```

To add values to an attribute, you use the methods addValue(...) for strings and addBinValue(...) for binaries and the methods removeValue(integer) and removeBinValue(integer) to delete selected values.

You don't even need to use DsmlValue classes. Just use the DSML attribute class's convenience methods to set and read string or binary values:

```
String strVal= "...";
DsmlAttr attr = new DsmlAttr();
attr.setName("mixedvalues");

// first set a binary and a string value ...
attr.addBinValue(bytearray);
```

```
attr.addValue(strVal);

// ... then read them:
byte[][] retBytes = attr.getBinValue();
String[] strvals = attr.getValue();
```

1.1.3.5.3. Testing Request Transformations

The framework offers a designated test connector to help you run automated test cases for your request transformation scripts or input readers. The test connector compares the requests it receives from the controller with requests stored in a reference file.

You configure your job for a standalone executable deployment and use the TestConnector class in the package siemens.dxm.connector as the connector. It compares the incoming requests with a reference taken from a file. Name the reference file in the configuration as shown in the following configuration snippet:

```
<connector
 role="connector"
 className="siemens.dxm.connector.framework.TestConnector"
 name="Test connector">
<connection type="file"
     filename="sampleData\sampleRsp.xml"
     >
     <property name="referencefile"
         value="sampleData\referenceRq.xml"
     />
</connection>
</connector>
```

The TestConnector returns a response for each request. Usually it mirrors the request attributes and options. If it receives a SearchRequest, it takes the search result entries from a file given in the filename attribute.

If you need to manage attribute values such as date or time that may change in each run, you can use wildcards "*" in your reference values.

1.1.3.5.4. Testing the Connector Responses

The framework offers a designated test writer to help you run automated test cases for your connector. The test writer compares the responses it receives from a connector with responses stored in a reference file.

You configure your job for a standalone executable deployment and use the SpmlTestWriter class in the package siemens.dxm.connector.test as a response writer. It

works the same way as the SpmlFileWriter, but additionally compares each response with a reference. Name the reference file in the configuration as shown in the following configuration snippet:

```
<connector
 role="responseWriter"
 name="SPML File writer"
 className="siemens.dxm.connector.framework.test.SpmlTestWriter">
 <connection type="SPML"
     filename="sampleData\receivedRsp.xml">
     <property name="referencefile"
         value="sampleData\referenceRsp.xml"
     />
 </connection>
</connector>
```

If you need to manage attribute values such as date or time that may change in each run, you can use wildcards "*" in your reference values.

1.1.4. About the Filter Interface

The filter interface represents the interface between the DirX Identity agent framework on one side and the connector component that provisions the target system on the other side. A filter is used to intercept connector calls. For example, a filter can build the delta between two consecutive search requests.

The interface consists of a mandatory and an optional interface. The mandatory ConnectotFilter interface consists of the following methods:

- open() - reads configuration data and initializes the filter component.
- close() - tidies up the filter component.
- doFilter() - processes the SPML request, passes it to the successor filter (or connector) in chain, processes the SPML response from the successor, and returns it to the initiator.
- setSuccessor() - gets the successor in the filter chain.

The optional ConnectorFilterConfig can be used if the filter must access the configuration of the associated connector; for example, if the filter must use the same connection as the connector to the target system. The method is:

- setConnectorConfiguration()

For details on the interface and the input and output parameter classes see the Java documentation provided on your DVD in the folder:

Documentation/DirXIdentity/ConnFrameWork

1.1.4.1. About the ConnectorFilter Interface

The interface siemens.dxm.connector.framework.filter.ConnectorFilter includes the following mandatory methods to read the configuration and open and close the filter.

void open(DXMFilterConfig *config*, Context *context*)

The open() method is the first call after the implementing class has been instantiated. This method must be performed successfully before any other methods can be performed.

The method opens and initializes the filter. The framework provides the configuration options in the *config* parameter of type siemens.dxm.configuration.DxmFilterConfig. The most important one is the class name of the filter.

Filter developers do not need to handle parsing the XML document. The framework handles this task and creates a Java class that corresponds one-to-one to the configuration. The attributes of the filter XML element are properties of the Filter class. You can read them by issuing a getProperty("*optionname*") call.

The following code snippet illustrates how to access standard and custom connection options:

```
String filterName = config.getName();
// how to read a custom configuration property:
String dbgFilename = (String)config.getProperty("debugfile");
```

The second parameter is the job context. From the job context the filter can get environment information and access to a connection pool manager.

void close()

The close() method tidies up the filter. After performing this method only an open() call is allowed.

It is important to call at least successor.close()in the close() method to tidy up the chain.

SpmlResponse doFilter(SpmlRequest *request*)

This is the main method of the filter. A filter can process requests and / or responses. It can process only certain requests and / or responses for example only add requests or only modify responses.

In any case it must perform a successor.doFilter()to call the successor in the chain, to get the response, and to return the response to the initiator.

See the Java documentation for details on SpmlRequest and SpmlResponse.

void setSuccessor(ConnectorFilter *successor*)

This method is performed once before the open() method. It provides the successor to the filter.

1.1.4.2. About the ConnectorFilterConfig Interface

The interface siemens.dxm.connector.framework.filter. ConnectorFilterConfig includes the optional method to get access to the connector configuration.

void setConnectorConfiguration(DxmConnectorConfig *connectorConfig*)

This method provides the configuration of the associated connector. It is performed once before the open() method.

The configuration can be used if the filter wants to use the same connection as the connector to the target system.

The following code snippet illustrates how to access the standard connection option:

```
DxmConnectionConfig connection =
(DxmConnectionConfig)
connectorConfig.getConnections().firstElement();
```

1.1.5. Configuring the Framework

The framework takes its configuration from an XML file. On startup, it parses the file and builds corresponding Java classes that the framework components interpret. The following figure shows the configuration structure.

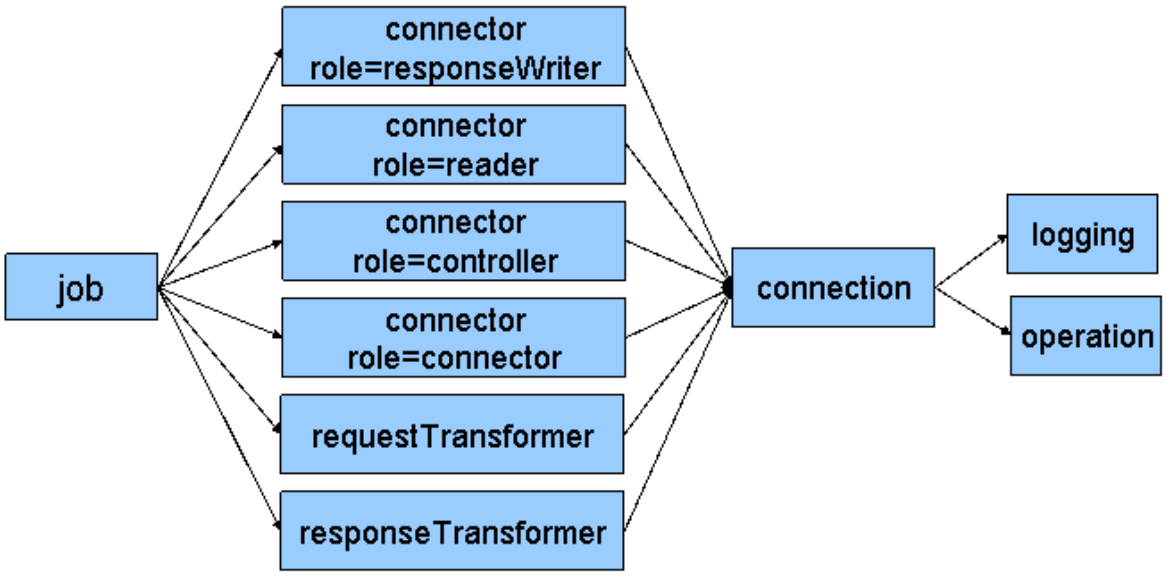

Figure 5. Java Connector Integration Framework Configuration

The root element is the job. It comprises a number of connector and optionally requestTransformer and responseTransformer elements. Each of these elements can contain a connection which in turn can contain a logging and an operation element. In addition, all of these elements can include property and mvproperty elements to carry customer defined single- or multi-value options.

The connector elements need the role attribute to distinguish between the connector types: reader, response writer, controller or connector. You can define a number of readers, writers or transformers, but only one controller and one connector. The connection elements carry information of the data sources or destinations, usually file names or directory or databases addresses.

The logging and operation elements are only evaluated by the controller and are thus ignored when entered as sub-elements of other connectors.

All of these elements comprise a list of standard attributes to cover usual and common options, such as user, password, filename, url, classname and many others. To support customer- and type-specific extensions, most of the elements can include additional options by entering them in <property> or <mvproperty> elements. A <property> element carries one value; a <mvproperty> carries a list of them.

The next sections provide more information about these elements.

1.1.5.1. Job

The <job> element represents a DirX Identity job and has no attributes. It allows the following subordinate elements:

- <controller>: 0..1; Options for the job controller.
- <connector>: 0..n; options for the reader, response writer, target system and other connectors.
- <port>: 0..n; Specifies the port to a target system. Is needed when SPML requests to a connector must be filtered. Contains the configuration for the connector and the filter. In this case, the connector must be configured subordinate to the <port> element and not subordinate to the <job>!
- <requestTransformer>: 0..n; options for the optional component that transforms the SPML input request before it is processed by the controller.
- <responseTransformer>: 0..n; options for the optional component that transforms the SPML response returned by the (target system) connector before it is passed to the response writer(s).
- <basename>: 0..1; Only evaluated in real-time workflows within the Identity server. It denotes the basename of the file folder from where the classes of the job implementation are loaded. The parent folder is: *install_path***/ids-j-***domain***-S** *n***/confdb/jobs**.
- <class>: 0..1; Only evaluated in real-time workflows within the Identity server. Contains the classname of the job implementation. For framework jobs this is com.siemens.idm.framework.Job.
- <mapFilename>: 0..1; Only evaluated in real-time workflows within the Identity server. Contains the path to the mapping file required to read the job configuration. Should be: *install_path***/ids-j-***domain***-S***n***/confdb/connector/agentconf.xml**.

1.1.5.2. Controller

Within a job configuration there is only one controller element. It represents the controller implementation. An equivalent configuration would use the <connector> element with the role="controller" attribute. The <controller> element allows the following subordinate elements:

- <operation>: 0..1; controls the operation of the controller.
- <logging>: 0..1; controls logging of the job. Only evaluated in standalone deployments, not in real-time workflows running in DirX Identity server.
- <property>: 0..n
- <mvproperty>: 0..n

It supports the following standard attributes:

- className: mandatory; the name of the implementing class.
- usePSE: optional; boolean (true / false), which tells the framework that this controller is allowed to crypt and encrypt data.

- name: optional; name of the controller; currently not evaluated.
- version: version of controller implementation.

The identity integration framework currently provides the following controller implementation classes:

- siemens.dxm.connector.framework.DefaultControllerStandalone - a simple "pass-through" controller for agent jobs in stand-alone deployments. Use this class in a configuration for jobs that read SPML or LDIF input files, process them against a target system and write the SPML response back to a file. This class is delivered with the framework as part of dxmConnector.jar.
- com.siemens.dxm.join.JoinAccount - the event manager in real-time workflows that handle password change events.
- com.siemens.dxm.join.SetPassword - used in real-time workflows to set the password at a target system.
- com.siemens.dxm.join.ErrorActivity - used in error activities of real-time workflows.

1.1.5.3. Operation

The <operation> element controls the operational options of the job controller. The options that are evaluated depend on the controller implementation. The pass-through controller (DefaultControllerStandalone) evaluates these options for checkpointing.

In general, the operation element accepts the following subordinate elements:

- <userhook>: 0..n
- <property>: 0..n

It does not currently support any standard attributes.

1.1.5.4. User Hook

The <userhook> element specifies a user extension called a "user hook" that is called during job processing.

The administrator can provide a JavaScript (or a Java class in future releases). Selected job controller implementations (currently SetPassword join engine, error activity, join account) call them at dedicated points while processing an SPML request. For information about how to write a user hook, see the samples delivered with the release.

The <userhook> element accepts the following subordinate elements:

- <data>: exactly 1. Contains only the JavaScript as its content enclosed in ![CDATA[...]] tags.
- <property>: 0..n. Simple properties to be evaluated by the user hook implementation; for example, the script.

The <userhook> element allows the following standard attributes:

- implementationLanguage: mandatory; specifies the language of the user hook implementation. Allowed values: "javascript".
- name: mandatory; the name of the user hook. Used to identify several user hooks within one job. The names must agree with the job controller. The current controllers read only the "mapping" hook.

Here is a typical user hook configuration:

```
<userhook implementationLanguage="javascript" name="mapping">
 <property name="pwdResetAttr" value="pwdLastSet"/>
 <data><![CDATA[... your javascript ...    ]]></data>
</userhook>
```

1.1.5.5. Logging

The <logging> element specifies the logger and some of its options, especially the log level.

It allows the following subordinate elements:

- <property>: 0..n

It supports the following standard attributes:

- logger: optional; class name of logger; currently not evaluated!
- filename: name of logging file
- level: integer between 0 (no log) to 9 (debug log)

The logger accepts the following property elements:

timestamp

If set, each log entry is prefixed with a timestamp according the format:

<property name="timestamp" value="*EEE MMM d HH:mm:ss.SS yyyy***" />**

where *EEE* denotes day-of-week, *yyyy* the year, *MM* the month, *d* the day, *HH:mm:ss* hour, minutes and seconds and *SS* milliseconds.

The format string is evaluated by the JDK class java.text.SimpleDateFormat.

modules

Provides a list of the classes to be logged. Enter the full class names. Wildcards can be included, as shown in the following example:

```
<property name="modules" value="*.AgtSessionExe, *.LdifFileWriter" />
```

If not set, the log level applies to all classes.

1.1.5.6. Port

The <port> element combines a list of <filter> elements with one <connector>. It specifies the port to a target system and is needed when SPML requests from the controller to a connector must be filtered. The <port> contains the configuration for the connector and the filter(s). In this case, the connector must be configured subordinate to the <port> element and not subordinate to the <job>!

The <port> element does not support any standard or custom properties, but just the following sub-elements:

- <filter>: 0..n; list of filter configurations.
- <connector>: 1. The corresponding connector.

1.1.5.7. Filter

A filter intersects the data flow from the job controller to a connector. Typical implementations encrypt or decrypt SPML requests and responses or build the delta between two consecutive search requests.

The configurations for the filter and the corresponding connector are subordinate to a <port> element. Thus different filters can be specified for different connectors. Several filters can be concatenated. The order of the configuration defines the sequence in which they are processed.

The <filter> element supports the following standard attributes:

- className: mandatory; the class name of the filter implementation.
- name: optional; name of the filter (to optionally distinguish between a list of filters).
- usePSE: optional; boolean (true / false), which tells the framework that this component is allowed to decrypt and encrypt data. A value of "true" is mandatory for the crypto filter!

The <filter> element only supports the standard sub-elements to specify custom single- and multi-value properties:

- <property>: 0..n
- <mvproperty>: 0..n

1.1.5.8. Connector

Each <connector> element represents a connector implementation such as a reader, writer, controller or target system connector identified by its role name. It allows the following subordinate elements:

- <connection>: a list is possible, but only the first is evaluated.
- <logging>: 0..1; currently only evaluated when subordinate to connector with

role=controller. (deprecated!)

- <property>: 0..n
- <mvproperty>: 0..n

It supports the following standard attributes:

- name: string. Used to distinguish between a number of connectors of the same type. Must be unique in within one job! Standard names in real-time jobs are: error, retry, audit. They identify the corresponding connectors for the error, retry and audit channels.
- role: mandatory attribute to specify the role of the connector within the framework. See this topic for the list of evaluated roles. Note that the roles that are evaluated and used depend on the controller implementation.
- className: the name of the implementing class.
- id: the identification of the connector.
- version: for future use.

Some framework classes accept properties beyond the standard ones. They are entered as <property> sub-elements in the following format:

<property name="*some property name***" value="***the value for your property***"/>**

The classes are:

- Ldif Change Reader
- SPML File Reader
- SOAP Connector

List of role names currently evaluated:

- reader: 1..n; currently only the first is evaluated. Reads the input requests (file or channel in a real-time workflow), transforms to SPML and passes to job controller.
- responseWriter: 1..n. Accepts SPML responses and outputs to destination (file or channel in a real-time workflow). May transform to other format; for example, LDIF content.
- connector: 1..n. A connector implementation that receives SPML requests and returns SPML responses. is supposed to process the SPML requests against the target system. A number of connectors are distinguished by their names. They must be unique and agree with the controller implementation.
- controller: 1. The construct <connector role="controller" .../> is deprecated. Use the equivalent <controller> element instead.

1.1.5.9. Connection

Each <connection> element provides information for a connector to connect to the data source: a file, a directory, a database or the like. It allows the following subordinate elements:

- <property>: 0..n
- <mvproperty>: 0..n

It supports the following standard attributes:

- port: port number of the target system.
- server: server name or IP address of the target system.
- filename: in case data is input or output to/from a file.
- url: URL of a Web service, if the target system is reached via HTTP.
- user: user name used for the connection.
- password: password used for binding to the target system.
- ssl: Boolean (true / false); use SSL/TLS connection.
- authentication: specifies the authentication method; for example, simple, strong.
- certificate: reserved for input of user certificate for binding.
- type: just descriptive
- trustStore: the path to the trust store file containing the certificate of the server to be used for SSL/TLS server-side authentication.
- trustStorePassword: the password that is needed to read the server certificate from the trust store.
- keyStore: the path to the key store file containing the private key or certificate to be used for SSL/TLS client authentication.
- keyStorePassword: the password that is needed to read the key from the key store.
- keyStoreAlias: the alias name to identify the private key in the key store.
- driverDBType: Only relevant for JDBC connector. Denotes the Java type for a class that characterises the database supported SQL data types and their handling through the JDBC connector. The following values are currently supported:

  siemens.dxm.connector.jdbc.DefaultDriverDB: the default, applies to most databases.

  siemens.dxm.connector.jdbc.SQLServerOverMicrosoftDriver: for MS SQL Server.

  siemens.dxm.connector.jdbc.AccessOverJdbcOdbcDriver: for MS-Access, HSQLDB.

1.1.5.10. requestTransformer

This element has the same attributes and subordinate elements as the connector element. The framework provides an XSLT transformer. It expects SPML requests, applies an XSL stylesheet on each request and passes the result back to the framework. The controller (specifically, the default controller class siemens.dxm.connector.framework.DefaultControllerStandalone) hands the transformed request to the next configured transformer or to the connector, if there are no more configured transformers.

The transformer is configured as shown in the following snippet:

```
    <requestTransformer
      role="requestTransformer"
      name="XSL transformer"
      className="siemens.dxm.connector.framework.SpmlXslTransformer">
      <connection type="XSL-stylesheet"
          filename="tf-request.xsl">
      </connection>
    </requestTransformer>
```

1.1.5.11. responseTransformer

This element has the same attributes and subordinate elements as the connector element. The framework provides an XSLT transformer. It expects SPML responses, applies an XSL stylesheet on each response and passes the result back to the framework. The controller (exactly: the default controller class siemens.dxm.connector.framework.DefaultControllerStandalone) hands the transformed response either to the next configured transformer or to the response writer, if there is no more.

The transformer is configured as shown in the following snippet:

```
    <responseTransformer
      role="requestTransformer"
      name="XSL transformer"
      className="siemens.dxm.connector.framework.SpmlXslTransformer">
      <connection type="XSL-stylesheet"
          filename="tf-response.xsl">
      </connection>
  </responseTransformer>
```

1.1.5.12. mvProperty

The <mvproperty> element is used to define agent-specific multi-valued attributes. It supports the following attributes:

- name

The option values are entered as <value> subordinate elements:

- <value>: 1..n

1.1.5.13. Property

The <property> element is used to define agent-specific additional attributes. It supports

the following attributes:

- name
- value

1.1.6. Configuring Default Controller Checkpointing

The default pass-through controller (DefaultControllerStandalone) evaluates the following operation properties to perform checkpointing:

- checkpoint_frequency - an integer value that specifies how often to write checkpoints.
- checkpoint_value - the value of the last checkpoint; an empty value specifies that no checkpoint was written in the last run.

A checkpoint frequency greater than 0 enables checkpoint handling. The controller ignores all incoming requests (add, modify, delete, not search) until one matches the configured checkpoint value. The subsequent requests are processed normally. Each time checkpoint_frequency requests have been handled, the controller writes a checkpoint to the checkpoint file "DeltaOutputData.txt". The checkpoint is usually the identifier of the request. Only when no identifier exists (which is just possibly for an AddRequest), the controller concatenates the first up to 5 attributes to the checkpoint.

If the job succeeds, the controller deletes the checkpoint file.

1.1.7. Configuring Encryption Filters

The filter class siemens.dxm.connector.framework.filter.CryptFilter decrypts request attributes and encrypts response attributes as they pass between the job controller and the connector.

When a job receives a request with encrypted data or reads encrypted attributes from some target system or file, the crypto filter decrypts the values and passes them to the connector in clear-text form. The same operation occurs on the return path: when the connector returns a SPML response, the crypto filter can encrypt the attributes. This allows the data to remain encrypted as long as possible and only be decrypted just before it is written to the target system or evaluated by the join engine. No connector needs to decrypt itself or call any decryption module; the framework does this automatically, if properly configured.

The filter does not decrypt/encrypt all attributes, just those that are configured. Note that the filter not only handles the attributes and modifications in the "normal" <attributes> and <modifications> section, but also the operational attributes.

The configuration options are set in a <filter> element within a <port> and at the same level with the associated <connector> element. The <filter> needs the following standard attributes:

- className: for the crypto filter this is "siemens.dxm.connector.framework.filter.CryptFilter"

- usePSE: needs to be set to "true" for the Crypto Filter.
- name: optional; some identifying string.

The Crypto Filter evaluates the following custom properties in the <filter> section:

decryptRequest (optional):

If this boolean property is set to "true", the filter decrypts selected attributes from the request.

decryptAttributes (optional):

This property contains the comma separated list of attributes that must be decrypted from the SPML request. When it receives a request, the filter also checks the operational attribute "encryptedAttributes". If one of these lists is empty, it decrypts all attributes of the other list. Else it decrypts only the attributes appearing in both lists. This means, the controller can restrict the set of attributes to be decrypted.

encryptResponse (optional):

If this boolean property is set to "true", the filter encrypts selected attributes from the response.

encryptAttributes (optional):

This property contains the comma-separated list of attributes that must be decrypted from the SPML response. The filter also checks the operational attribute "encryptedAttributes" in a response. For each of the attributes listed there, it assumes that it is already encrypted and thus ignores it. It encrypts only the rest of the configured attributes.

Here is a sample configuration snippet that directs the filter to decrypt the attribute "unicodePwd" in the request and encrypt the attribute "password" in the response:

```
<port mode="inout" name="TS">
      <filter
className="siemens.dxm.connector.framework.filter.CryptFilter"
name="CryptSupport" usePSE="true">
              <property name="decryptRequest" value="true"/>
              <property name="decryptAttributes" value="unicodePwd"/>
              <property name="encryptResponse" value="true"/>
              <property name="encryptAttributes" value="password"/>
      </filter>
      <connector ... />
</port>
```

1.1.8. Configuring Connectors

This section describes the configuration properties of the connectors provided with the Java Connector Integration Framework.

1.1.8.1. SPML SOAP Proxy

The class siemens.dxm.connector.framework.soap.SpmlSoapProxy is a connector that sends the SPML request as a SPML/SOAP request message to the URL specified in the configuration. The message conforms to the SPML/SOAP binding standard.

The SPML SOAP client sends SPML SOAP requests over HTTP to the configured endpoint and receives SPML SOAP responses from a SOAP service. To deploy the framework in this way, just select the appropriate class as the connector. You have two options:

- DxaSpmlSoapProxy: produces SPML requests according the latest OASIS SPMLv1 standard (http://www.oasis-open.org/committees/download.php/4138/os-pstc-spml-schema-1.0.xsd).
- SpmlSoapProxy: produces SPML requests according a more recent version of SPMLv1 that is not considered the official release anymore (http://www.oasis-open.org/committees/download.php/2396/cs-pstc-spml-schema-1.0.xsd).

The connector evaluates the following standard and non-standard properties.

Standard attributes:

url

optional. The endpoint where to send the request; for example, **http://localhost:8080/spml/spmlservice**.

You either have to specify the SOAP endpoint completely in this url parameter or provide the parts in the properties 'server', 'port', 'ssl' and the non-standard property 'path'.

> A protocol selector of "https" requests SSL/TLS protocol. In this case, you must ensure that the certificate of the addressed Web server is imported in the trust store of the Java runtime. See the JDK documentation (**keytool**) for details.

server

optional. If no url is given, this property provides the server part of the url. In the above sample, this would be "localhost".

port

optional. If no url is given, this property provides the port of the url. In the above sample, this would be "8080".

ssl

optional. If no url is given, this property tells the connector which protocol to use. In case of 'true', 'https' is selected. Otherwise, the connector sets 'http'. In the above sample, a

missing ssl property or the value 'false' would apply.

user

optional; the user name used for HTTP basic authentication.

password

optional; the password used for HTTP basic authentication.

trustStore

optional; the path to the trust store file, which contains the certificate of the server to be used for SSL/TLS server-side authentication.

trustStorePassword

optional; the password that is required to read the certificate from the trust store.

keyStore

optional; the path to the key store file that contains the private key or certificate to be used for SSL/TLS client authentication.

keyStorePassword

optional; the password that is needed to read the key from the key store.

keyStoreAlias

optional; the alias name to identify the private key in the key store.

The SOAP connector evaluates the following non-standard properties beneath the <connection> element:

maintainSession

optional, boolean (true / false); if set to "true" (the default), maintains the HTTP session to the target Web service between consecutive requests and thereby saves performance.

path

optional. If no url is given, this property provides the path of the url. In the above sample, this would be "spml/spmlservice".

timeout

optional. The socket timeout in seconds. Default is 60 sec.

Here a configuration sample using the url property to denote the SOAP endpoint:

```
<connector    role="connector"
className="siemens.dxm.connector.framework.soap.DxaSpmlSoapProxy"
name="SPML connector">
<connection type="SOAP"

url="http://localhost:8080/spmlsoapservice/services/SpmlSoapService"
```

```
/>
</connector>
```

The following is an alternative with the older SPML implementation using the properties server, port, path and ssl to denote the SOAP endpoint.

```
<connector  role="connector"
className="siemens.dxm.connector.framework.soap.SpmlSoapProxy"
   name="SoapConnector">
   <connection type="SOAP"
       server="localhost" port="8080" ssl="false"
   >
       <property name="path"
value="spmlsoapservice/services/SpmlSoapService"/>
   </connection>
</connector>
```

1.1.8.2. LDIF Change Reader

The framework class siemens.dxm.connector.framework.LdifChangeReader reads LDIF change requests from file. It does not evaluate any connector attribute or property, but needs the following standard attribute in the <connection> element:

- filename: mandatory; denotes the name of the input file.

The LdifChangeReader class accepts the following non-standard properties in the <connection> element:

- IdentifierType - specifies the identifier type to be used in SPML requests sent to the connector. Usually the reader uses the DN type. The values must be compliant with the SPML standard. For using the generic string type, set

  <property name="IdentifierType"
  value="urn:oasis:names:tc:SPML:1:0#GenericString"/>

  In this case, the other properties "ExtractRdn" and "IncludeNamingAttribute" also must be set.

- ExtractRdn - directs the reader to use either the entire DN as the identifier (value="false") or just the naming part (value="true"). If set to false, the property "IncludeNamingAttribute" is not evaluated.
- IncludeNamingAttribute - directs the reader to include the naming attribute (value="true") or just use the value of the naming attribute (value="false").

This sample illustrates how these properties function:

```
<connection type="LDIF"
 filename="sampleRq.ldif">
 <property name="IdentifierType"
value="urn:oasis:names:tc:SPML:1:0#GenericString"/>
  <property name="ExtractRdn" value="true"/>
 <property name="IncludeNamingAttribute" value="false"/>
</connection>
```

Using this configuration, the reader transforms the DN value:

"cn=John Doe, cn=PowerUsers,... "

to the SPML identifier "John Doe".

1.1.8.3. SPML File Reader

The framework class siemens.dxm.connector.framework.SpmlFileReader reads SPML requests from file. It does not evaluate any connector attribute or property, but needs the following standard attribute in the <connection> element:

- filename: mandatory; denotes the file name of the input request file.

and the following non-standard properties (<property name="..." value="..."/>):

- validate: optional, boolean (true / false); if set to "true", the reader turns on validation checking for the request, i.e. checks against the SPML schema. In case of validation errors it stops parsing the request.

Sample configuration snippet:

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

1.1.8.4. SPML Loop Reader

The framework class siemens.dxm.connector.framework.test.SpmlLoopFileReader reads SPML requests from file. In contrast to the SPML file reader, this connector processes the input file in a loop.

In the <connector> element, it evaluates the following non-standard property:

- counter: optional, integer. The integer value tells the reader how often to process the requests of the input file.

Furthermore, it evaluates the following standard attribute in the <connection> element:

- filename: mandatory; denotes the file name of the input request file.

and the following non-standard properties (<property name="..." value="..."/>):

- validate: optional, boolean (true / false); if set to "true", the reader turns on validation checking for the request; that is, it checks against the SPML schema. In case of validation errors it stops parsing the request.

Sample configuration snippet:

```
<connector role="reader" name="SPML loop file reader"
className="siemens.dxm.connector.framework.test.SpmlLoopFileReader">
    <property name="counter" value="2"/>
    <connection type="SPML" filename="request.xml">
        <property name="validate" value="true"/>
    </connection>
</connector>
```

By using a placeholder in the input file you can arrange for different data per loop.The reader replaces a variable *${counter}* with the value of the respective loop.Here a sample snippet for an input request:

```
<spml:attr name="employeeNumber">
    <dsml:value>EN-${counter}</dsml:value>
</spml:attr>
```

1.1.8.5. SPML File Writer

The framework provides two response writers that write SPML responses to a file:

siemens.dxm.connector.framework.DxaSpmlFileWriter: SPML response conforms to the latest OASIS SPMLv1 standard (http://www.oasis-open.org/committees/download.php/4138/os-pstc-spml-schema-1.0.xsd).

siemens.dxm.connector.framework.SpmlFileWriter: produces SPML output according to an older version of SPMLv1 that is not considered the official release any more (http://www.oasis-open.org/committees/download.php/2396/cs-pstc-spml-schema-1.0.xsd).

The writers do not evaluate any connector attribute or property, but the following from the

<connection> element.

- filename: mandatory; denotes the file name of the output file.

Non-standard properties:

- validate: (optional); if set to 'true', the writer validates the given response.

Sample configuration snippet:

```
<connector role="responseWriter" name="SPML File writer"
className="siemens.dxm.connector.framework.DxaSpmlFileWriter">
    <connection type="SPML" filename="response.xml">
     </connection>
</connector>
```

1.1.8.6. LDIF File Writer

The framework class siemens.dxm.connector.framework.LdifFileWriter transforms SPML responses to LDIF content format and writes them to the configured file. It does not evaluate any connector attribute or property, but the following attributes and properties from the <connection> element.

Standard attribute:

- filename: mandatory; denotes the file name of the output file.

Non-standard properties:

- **NamingAttribute** (optional) - this property is needed when the response identification is not a DN type or when the response contains no identification. In this case it prefixes the DN value with the RDN built from the configured attribute type and the attribute value. The attribute value is either taken from the response identification or from the attribute with the configured type. Assume you have configured ...

<connection ...

<property name="NamingAttribute" value="employeeNumber"/>

</connection>

i. the LDIF output DN would start with

dn: employeeNumber=123,...

- **contenttype** (optional) - denotes the LDIF format to be written to the output file. Allowed values are: "LDIF-CHANGE" (default), "LDIF-CONTENT".

The LDIF writer processes SPML responses as follows:

- For search responses, the writer produces LDIF content with the list of result entries.
- For add responses, the writer takes the DN value from the response identifier, outputs the list of attributes in the response and sets the changetype to "add", if contentype LDIF-CHANGE is configured.
- For modify responses**,** the writer tries to construct a DN from either the response identification, which it prefixes with "cn=" or from an operational attribute named "dn". If format "LDIF-CHANGE" is configured, the writer outputs the response with changetype "modify" and the list of attribute modifications. Otherwise it simply outputs LDIF content format with the list of attribute values taken from the modification elements.
- For delete responses, the writer tries to construct a DN as with the modify response processing. If format LDIF-CHANGE is configured, it outputs the entry with changetype "delete". Otherwise it outputs an attribute type "isDeleted" with value "1".

1.1.8.7. Test Connector

The framework provides a connector just for testing purposes: siemens.dxm.connector.framework.TestConnector. The test connector outputs SPML requests to file and returns responses taken from a response file. If a reference request file is configured, it compares each request with the reference and logs errors in case of mismatch. A wildcard "*" in the reference value matches with all values. It does not evaluate any connector attribute or property, but the following attributes and properties from the <connection> element.

Standard attribute:

- filename: (optional); denotes the name of the file that contains a search response to be returned. If not configured, the connector throws an exception in case it receives a search request. For other requests, the connector returns default responses.

Non-standard properties:

- debugfile: (optional); if configured outputs each received request to the file in SPML format.
- listfile: (optional); if configured outputs each received request to the file in LDIF like pseudo format.
- referencefile: (optional); this file contains the reference requests. The connector tries to find a reference for each received request. It associates both with the requestid. If the received request does not contain an id, it simply takes the next reference request of the same type. If the received request and the reference do not match, the connector logs an error.

Here is a sample configuration snippet for the test connector:

```
<connector role="connector"
className="siemens.dxm.connector.framework.TestConnector" name="Test
connector">
    <connection type="file" filename="sampleRsp..xml>
```

```
        <property name=" debugfile" value="dbgOut.xml"/>
        <property name="listfile" value="listing.txt"/>
        <property name="referencefile" value="referenceRq.xml"/>
    </connection>
</connector>
```

1.1.8.8. Test Writer

The framework provides a response writer just for testing purposes: siemens.dxm.connector.framework.test.SpmlTestWriter. The test writer outputs SPML responses to file. If a reference file is configured, it compares each response with the reference and logs errors in case of mismatch. A wildcard "*" in the reference value matches with all values. It does not evaluate any connector attribute or property, but the following attributes and properties from the <connection> element.

Standard attribute:

- filename: (mandatory); the name of the file, where to write the SPML response.

Non-standard properties:

- referencefile: (optional); this file contains the reference responses. The writer tries to find a reference for each received response. It associates both with the requestid. If the received response does not contain an id, it simply takes the next reference response of the same type. If the received response and the reference do not match, the writer logs an error.
- validate: (optional); if set to 'true', the writer validates the given response.

Here is a sample configuration snippet for the test writer:

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

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

1.2. C/C++ Connector Integration Framework

The DirX Identity C/C++ Connector Integration Framework helps developers build DirX

Identity connectors for new types of target systems with a minimum of effort.

The C++ connector server (which is part of the C++-based Identity Server) is a framework for using DirX Identity connectors written in C++.While Java connectors are integrated into the Java-based DirX Identity server as Java classes, C or C++ connectors are integrated into the C++-based Identity Server as shared libraries.The following figure illustrates the C/C++ Connector Integration Framework architecture.

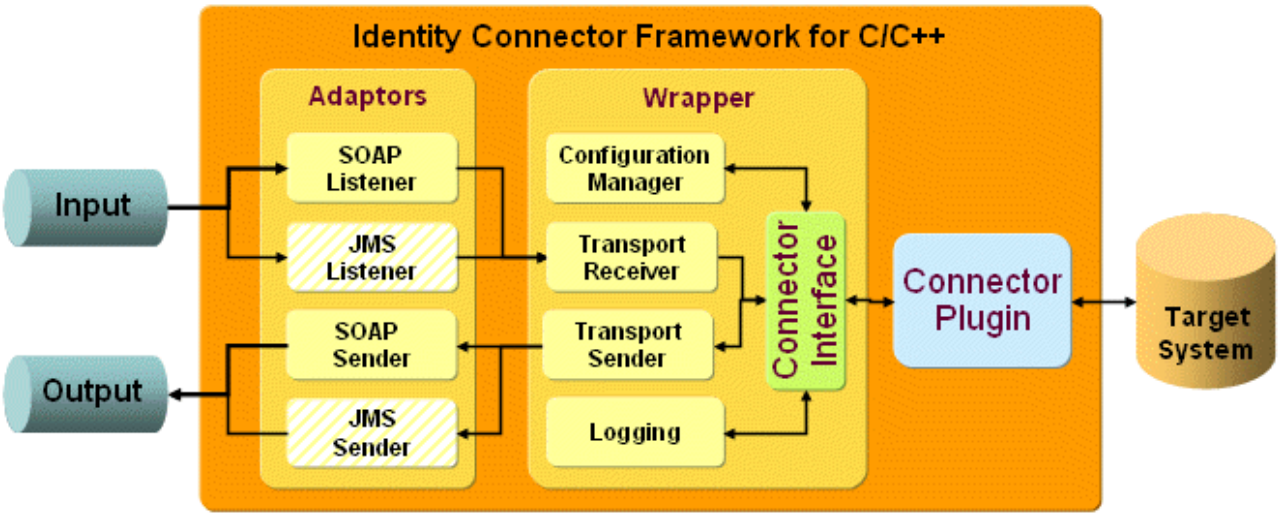

Figure 6. C++ Connector Integration Framework Architecture

DirX Identity delivers a set of C and C++ connectors and allows you to create your own custom connector to provide special functionality or access to a special target system.You implement a C/C++ connector as a shared library using the connector server API, deploy the library to the installation path and insert this library name (and other related configuration data) into the C/C++ server configuration.

1.2.1. How a C++ Connector Works

The C++ connector server is a multithreaded application that communicates with the DirX Identity Java server using the SPML-over-SOAP interface. Connectors are inserted into the server as “plugins” in the form of shared libraries. Connectors run in separate threads inside the server (the thread count for each connector is configurable). Each connector receives SPML requests that are addressed to it. These requests represent operations that should be carried out in the target system to which the connector belongs. After accomplishing the requested operation in the target system, the connector forms an SPML response that describes the results of the operation and returns the response to the connector server, which then returns it to the client.

The following types of error can occur:

- An error occurs in the operation that the connector carries out in the target system but the error has no influence on the connector’s ability to run; it only means that the current request cannot be handled. For this type of error, the connector should use the error fields inside the SPML response; that is, it should return an "SPML response with error" result.
- A serious error occurs in the operation of the connector and/or target system that

prevents the connector from processing further requests. For this type of error, the connector should throw an exception (of a certain type, discussed later in this section). This exception will be caught in the server. The SOAP request will be answered with an error and, depending on the contents of the exception, the thread in which the connector is running may be finished, the connector can be suspended from operation, and so on.

- An exception of an unknown or unexpected type occurs during the connector operation; this type of error will be caught on the server level. The server attempts to terminate the affected thread and continue working.

The process of connector server operation and co-operation with connectors runs inside the multithreaded application; that is, all the process steps run independently of each other. This means, for example, that if one connector blocks to carry out a time-consuming operation the other connectors can continue working. However, this independence is limited as you can see from the unknown/unexpected error case. Even when connectors run in separate threads, they are still part of a single application and fatal errors and exceptions may have application-wide effect.

1.2.2. About Connectors and Filters

A C/C++ connector is a pluggable component implemented in a shared library whose task it is to carry out operations on a particular target system. A C/C++ connector can also have another pluggable component optionally associated with it - this component is called a filter.

A C/C++ connector can have zero or any number of filters associated with it. Filters are ordered into a chain. The SPML requests coming to a connector are handed to all of its associated filters in the chain and the SPML responses are handed to all these filters in backward order. Each filter carries out some transformation of the requests / responses; that is, the filter's task is to transform the request (according to the filter's purpose and configuration), hand the request to the next filter in the chain, receive the response from that filter, transform it back and return to the caller.

DirX Identity delivers a set of standard filters; for example, the encoding transformation filter that transcodes the strings inside the SPML classes from one character encoding to another. You can also implement your own filters; this task is described in the connector API section.

1.2.3. Using the C/C++ Connector API

This section describes the components of the API that the C/C++ connector server defines for the development of custom connectors and filters.

1.2.3.1. Creating and Compiling a Connector Library

As mentioned in an earlier topic, you insert a C/C++ connector into the C/C++ connector server as a dynamically-loaded shared library. The connector itself is a class inside this library. A connector library is a special case of the connector server component library. It is a shared library that contains one dynamically-loaded component (in this case, the connector class).

The component library must observe the following rules:

- It must contain and export exactly one component (connector) class implementing (that is, inheriting from) the component (connector) interface (abstract predecessor class) – CConnectorInterface.
- It must contain the class method's implementation for the component interface classes. One class method is used as a factory method for producing instances of the actual component (connector) classes. Other class methods are used for initializing and uninitializing the whole library (if needed for multithreaded use of the library). For a more detailed description, see the "Implementing a Connector Class" topic.

The library can contain any number of other classes or functions in addition to the connector class.

To compile the component library, you include the main API header file (which includes other header files needed for the library) and link it with the API shared libraries – see the list of files in "Additional Information". Compile and build the library using compatible compiler and linker environment and settings with the connector server. Note that these implementation details may change with newer versions of the framework.

1.2.3.2. Implementing a Connector Class

The connector is an instance of the connector class which must be contained in and exported from the connector shared library. The connector class must be inherited from the abstract class CConnectorInterface. It provides the following methods:

- Constructor / destructor - call these methods when the instance is created using the GetInstance method or deleted using the delete operator.
- Open - call this method before the connector handles any requests. The method receives an instance of the current configuration data (see the "About the Connector Configuration Class" section) and should open and prepare its connection to the target system. Note that the open method can be called more than once during the lifetime of the object instance. The method close will always be called between two calls of open. This method pair defines the state of the object as opened or closed. Calls to methods other than open are only allowed in the opened state. The open method prepares the object's connection to the target system and initializes associated internal structures so that after the method completes, the operation on the target system can be successfully performed without delays.
- Add / Modify / Delete / Search - call an "operation" method to carry out the operation described by the request class and return the operation result using the response class. The methods receive an internal SPML representation (ISR) class (see the "About the ISR Classes" section for details) that contains the representation of the SPML request and response.
- Close - call this method when no more operation methods are to be called. It can be called at any point in the connector server operation. Open/close methods can be called more than once during the lifetime of a connector instance but it is guaranteed that no operation method is called after close and before open.

In addition to the connector class, the connector shared library must contain the

implementation of the following CConnectorInterface methods:

- The factory method of the abstract interface class - CConnectorInterface::GetInstance. This method (which must be implemented in each connector shared library) produces the instances of the desired connector every time the server requires it. This method must instantiate the appropriate connector class (that is implemented in the library) and returns the pointer to the newly created instance (see the declaration of the factory method).
- The constructor and the destructor of CConnectorInterface - they should be empty.

Note: depending on the connector configuration, more than one connector class instance may exist at one time, each instance running independently in a separate thread. If you intend to configure the connector class this way, appropriate mechanisms should be used when implementing the class to ensure the multithreading capability of the class.

1.2.3.2.1. Implementing a Filter Library

Filters are implemented in the same way as connectors: as dynamically-loaded classes inside a shared component library. Most of the rules that apply to connector libraries also apply to filter libraries.

The filter shared library must contain exactly one filter class which inherits from the abstract filter interface class CFilterInterface defined in the filter API header (see "Additional Information"). The library can contain any number of other classes and functions in addition to the filter class.

Compile and link a filter library in the same way as a connector library.

1.2.3.2.2. Implementing a Filter Class

The filter class contained in the filter shared library must be inherited from the abstract CFilterInterface base class. It provides the following methods:

- Constructor and destructor - call these methods when the instance is created using the GetInstance method or deleted using the delete operator.
- Open – call this method to initialize the filter; the rules for the connector open method apply to this method, too. The method receives the configuration class, the PSE context class and the NextInChain argument, which points to the next filter instance in the chain that must be used in the DoFilter method. The open method should store this pointer for later use.
- DoFilter – call this main filter method to transform the requests and responses. The method has one input/output argument of type CRequestResponse. This compound class can contain both the SPML request and the response. When the DoFilter method is called, the compound object contains only the request. The method may first need to cast the compound object to some of its descendants depending on the operation defined in the request. Use the casting methods described in "About the ISR Classes". The method should then transform the SPML request class (use the modification methods) depending on the configuration data received in the open method. Next, the descendant filter's DoFilter method should be called to handle the compound object with the request as the input argument. When the descendant's DoFilter returns, the

compound object now contains the response (created by the connector associated to the filter chain). The filter must now transform the response back and return it in the input/output compound object argument. The compound object and the contained request and response remain the same (instance) during this process; only their contents change (using their modification methods).

- Close – call this method to place the instance in the closed state; the rules for the connector close method apply to this method, too.

You can use the ISR class transformation methods in the filters to transform all strings of a request/response. See the "About the ISR Classes" section for details.

The filter class must also implement the factory method and constructor/destructor class methods defined in the base interface class.

As in the connector case, more instances of a filter class will typically exist. They may be part of one filter chain (and so exist inside the same thread) or they may be parts of different chains of two connector instances. In the latter case, the instances will exist in separate threads. Multithreading capability of the filter class is therefore required.

1.2.3.2.3. Loading and Initializing the Component Library

Previous topics described the component shared libraries that are used to plug the filters and connectors into the DirX Identity servers. There are also features that are common to all such component libraries. The abstract interface classes of filters and connectors are derived from the common base class CLibComponentInterface. This base class defines the InitLibrary and TermLibrary methods for initializing the entire shared library. These methods are called only once each, after the library is loaded and before it is unloaded. They are intended for initializing internal structures of the library, especially those used for multithreading.

The component libraries are loaded in the server's initialization phase or when it is reconfigured in runtime. If an error occurs during the initialization phase (this includes the case when the InitLibrary method throws an exception), the associated components (the connector and its filters) are placed in an error state (you will see a message in the log) and will not be operational until the server is restarted or reconfigured.

If the server is reconfigured in runtime, the state of the connectors and filters is adapted to the new configuration. Connectors and filters that no longer exist are removed and new instances are added. The configuration parameters of existing components are changed and the server will try to initialize the components that are in an error state.

When a component library changes (its contents) during server operation, it will not be reloaded until the server starts again.

1.2.3.3. About the Connector Configuration Class

The instance of the class CConfigData is given as an argument into the connector's open method. This class is called the connector configuration class and is declared in the configuration header file (see "Additional Information" for a list of relevant files). It contains "get" methods for retrieving particular configuration parameters that are in effect for the

current connector (like user, password, type, name, version, server, port) and for retrieving properties, which are a set of name/value pairs. See also the contents of the configuration header file.

1.2.3.3.1. About the Filter Configuration Class

Analogous to the connector configuration class is a filter configuration class named CFilterConfigData which is declared in the filter configuration header (see "Additional Information"). Use its "get" methods to retrieve the configuration data from the class. An important part of the functionality is carried out by methods inherited from the base class CCommonConfigData. This base class is common for both filter and connector configuration classes.

1.2.3.4. Handling Errors and Exceptions in Connectors and Filters

There are two ways to react to errors inside a connector or a filter:

- Return an error result and error message inside the SPML response returned by the connector's or filter's operation method. This is an appropriate way to react to errors related to individual requests (for example, the object to which the operation should be applied does not exist in the target system).
- Throw an exception of type ECntrException. Use this mechanism when there is a problem with running the connector that may last for some period of time and is not bound to the current request - for example, when the target system has a serious error or is inaccessible. Depending on the flags (see the exception class declaration) you include in the exception and the count of such exceptions thrown by the given connector, the connector server decides what to do with the connector and its thread. The server usually attempts to close and open the connector again and, if this action does not help, may terminate the thread or place the connector in an error state.

> ECntrException is the only exception type that a connector or filter can throw. This rule has the following important consequences for connector and filter developers:

- Connector and filter code must catch and handle all exceptions that may be thrown from any C++ components used inside the connector or filter (possibly by throwing an ECntrException but not necessarily). This rule includes ISR classes (see the "Exceptions" section in "ISR Classes") and exceptions thrown by various I/O operations (like file manipulation). You typically find the thrown exceptions in the API of these components, in comments or directly in the source code. If you do not have such information available, consider adding general catch handlers at the places where the components are called so that the cause of the problem can be more easily found when the call fails.
- Interpret each exception by determining its cause and writing an understandable message about the cause of the error into ECntrException or into the error message in the SPML error response you produce. The error message should be clear to a user - who is typically not a developer - so don't mention the exception itself, as it's only an internal mechanism for error handling and not the cause of the error.

If you fail to catch all exceptions inside the connector or filter, the server will consider any

thrown exception as a fatal C++ exception (because the server has no way of determining the cause of the exception). In this case, the server will place the connector in the error state and it will not be operational until the server is restarted or reconfigured.

1.2.3.5. Logging Connector Messages

The connector server contains a framework for convenient logging of application messages. This framework is accessible for the connector implementation using the macro DXM_MSG_PRINTF. As arguments for this macro you must insert:

- The connector or filter logging number - every connector and filter has its own logging number, which is set in the server configuration and can be retrieved inside the connector implementation by calling the getLoggingNo method of the configuration class.
- The message type - messages have several degrees of “fatality” defined, from fatal errors over warning to several debug messages levels. See the main API header file for details.
- A message string - a string that describes the message.

The logged messages will be written into the DirX Identity C++ server log.

1.2.3.6. About the Heap Check Mechanism

Memory management is one of the most important quality aspects for C++ applications, and memory leaks represent the worst problem in this field. Memory leaks are pieces of memory that were allocated using various allocation mechanisms (new, malloc, STL allocators and so on) but which were not released once they were no longer needed. In a continuous-run application like the connector server, memory leaks cause it to consume all operational memory of the computer on which it’s running and then crash. As a result, you need to develop connectors and filters that do not contain memory leaks. The connector server API offers a heap check mechanism for debugging memory leaks. It is intended for classes defined inside the connector library (and not for classes from third-party libraries like STL).

The connector and filter interface classes are inherited from the CCSvrObjBase class, which implements the heap check mechanism. When allocating instances of a class inherited from CCSvrObjBase, you must use the macro CSNEW instead of the standard new operator. For deallocating (destruction) of such object instances you use the normal delete operator. The heap checking mechanism records all of these allocations and logs the object instances that have not been deallocated at the end of the server run. In a normal server termination, these instances represent memory leaks. You can obtain the line number at which the allocation occurred from the server log – logging component **csvr**, level debug 1.

The heap checking mechanism is turned off by default so that it does not adversely affect the server’s performance. You can turn it on for debugging purposes; see the server-wide configuration section for details.

1.2.4. About the ISR Classes

Internal SPML representation (ISR) is a submodule of the C++ connector server interface library that provides a set of C++ classes that implement various SPML constructs. ISR is a

custom SPML implementation that includes all SPML constructs supported by the DirX Identity connectors.

The DirX Identity C++ connector server uses these classes to communicate with the connectors that are running within it. SPML requests are received in textual form by the server from other system components via various communication channels. The C++ connector server unmarshalls each request and creates the respective ISR object for it. This ISR object is then sent to the connector (and its filters) via the standard connector interface (as an argument of the connector interface method).

The C++ connector or filter that receives the object can use its methods to retrieve the information from the request. According to this information the connector provides the requested services (writes or reads some data etc.). After that the connector creates an ISR response object using other methods of the original ISR object and returns the object back to the server through the connector interface.

In this chapter explains how to use ISR classes. See the ISR header files for the class structure in detail. ISR classes implement the SPML classes defined in the XSD specification of SPML. In the class comment in the header files, you'll find a reference to the XSD class that the ISR class represents. Note that the SPML class names sometimes differ only in plural (that is, the ending "s"). You'll also find some comments in the header file that describe conditions and constraints for using the ISR classes and their methods.

1.2.4.1. About the Compound Classes

There are two groups of ISR classes: compound classes and SPML classes. The compound classes are wrappers for an SPML request/response pair. In normal operation, the server receives an SPML request and reacts with an SPML response, while the connectors actually handle the SPML data object - the requests are the input data for the connector and the responses are the output data. However, responses are logically bound to the requests. They constitute one logical data entity (which can be described as the data of one connector transaction).

From the logical point of view, the connector receives (in the call to modify, add, delete or other interface methods) one instance of the compound ISR object. This compound object contains only the SPML request object. The connector handles the data from the request and then stores the results of its operation in an SPML response object and inserts it into the original compound object. This object is then returned to the server. One operation of the connector can thus be described as reading and modifying the received ISR compound object and acting according the information in it.

Examine the base compound class CRequestResponse. Its relevant methods include:

- Casting methods – these methods are used to downcast the object; that is, to convert it to a pointer to the appropriate descendant type. You will need these methods in filters before you can modify the contents of the requests/responses. The correct method must be called (according to GetType), otherwise an exception is thrown.
- Methods that return the contained object instances - GetRequest and GetResponse. These methods return pointers to the contained SPML request/response objects (or NULL if no such object is stored in the compound object). The user is expected to

change the returned instance (by calling its methods) but is not allowed to destroy the returned instance (since the compound object still owns it).

- Method that creates the response sub object - CreateResponseFromRequest. This method creates the SPML response object from the contained request object by copying some important data structures (like RequestID or Identifier) from the request to the response. Other data fields of the response remain empty or have default values.
- Methods that set common data fields in the response sub object - SetErrorMessage, SetErrorCode, SetResult. The response sub object must exist before these methods are called.

Other compound classes are derived from the base class. These derived classes implement the particular main SPML operation types. Look at the class CModifyRequestResponse. it contains the following methods (in addition to the inherited methods from the common compound base class):

- Methods that add new data items directly to the response sub object - AppendModification, AppendOperAttr. The response sub object must exist before the methods are called. See also Response classes.

1.2.4.2. About the SPML Classes

The SPML classes provide a C++ implementation of the standard SPML classes, and include the following types:

- Base classes - these classes contain methods that are common to a certain group of SPML classes. These base classes are not called directly; they are accessed through their methods inherited in descendant classes.
- Request and response classes - these classes are called directly and implement SPML requests and responses. Their objects typically contain sub objects (which represent various SPML substructures, for example, attributes or modifications). An example of this group of classes is CModifyRequest
- SPML substructure classes - these classes are not self-standing SPML structures, but they are contained in the request and response objects. For example, CSpmlAttributes or CSpmlIdentifier.

Look at the base classes. The class CSpmlBase is the base class common to all SPML classes (request, response and substructure classes). The following methods are of interest:

- Marshall - this method appends the textual form (XML) of the respective SPML object to the given string stream
- Equals - this method returns true if the given object is equivalent with the method owner. Equivalent means that it can differ only by the sequence of attributes / modifications and so on.

The class CSpmlReqRespBase is the base class common to all request and response classes and inherits the methods from CSpmlBase. The following methods are of interest:

- Unmarshall and UnmarshallFromFile - these methods are class methods; that is, they are used to create a new instance of a request or response. Unmarshall parses the XML

from a memory buffer, UnmarshallFromFile from a file. The methods return an instance of type CSpmlReqRespBase, but you should use the appropriate casting method to get the correct request or response type.

- Casting methods - these methods are named **To***Operation***Request** and **To** *Operation***Response** (example, **ToModifyRequest**). You call these methods after you get the CSpmlReqRespBase pointer on a new SPML object instance from the unmarshall methods. The real type of the object is, however, not CSpmlReqRespBase but one of the request/response types (descendants of CSpmlReqRespBase). You call the respective casting method on a CSpmlReqRespBase instance to cast it to the desired class request/response instance.
- Content reading methods - GetOperAttribs, HasOperAttribs, GetIdentifier, HasIdentifier and GetRequestId - these methods return the SPML substructures of a request/response object
- Copy methods - these methods copy the SPML substructures from other objects.
- Add/set methods - these methods programmatically add new SPML substructures into the instance.

There is also a base class common to all request types (that is, not responses). This class CSpmlRequest inherits from CSpmlReqRespBase and has only some minor "get" methods. For responses, there is the CSpmlResponse base class, which inherits from CSpmlReqRespBase. It contains methods for handling SPML result and error codes.

Request objects (for example, CModifyRequest) inherit from CSpmlRequest and contain additional methods for handling special SPML substructures contained only in the respective type of request. For CModifyRequest the Modifications element is an example. Handling of other SPML subelements is inherited from CSpmlRequest and CSpmlReqRespBase respectively. Analogous with the response classes (e.g. CModifyResponse).

1.2.4.3. Using ISR Classes

The ISR classes are normally created within the C++ connector server after the SPML requests are received from various communication protocols. Once a request is received, it is unmarshalled into the request type and inserted into the appropriate compound object (for example, CModifyRequestResponse).

The compound object is then sent to the connector methods (Add/Modify/Delete/Search). The rest of this section describes the actions that should be performed inside the connector to handle the request correctly and return the correct response. Here the actions will be explained in the context of a modify request, the use of other request types is analogous.

First you must get the request from the compound object, for example, by calling GetModifyRequest. From the CModifyRequest pointer you read the SPML data that the connector needs for its operation. This is the request ID (for logging purposes), the identifier, operational attributes and modifications. You use the "get" methods of the request class (CModifyRequest) or its base class methods. These methods often return pointer to sub objetcs (for example, CSpmlAttributes). You should not modify or delete these objects that you get from the request object. These objects are only meant to be

read.

After getting the information, you handle the actions in the connector; that is, write some data into the target system, change or delete them. The results of this data operation must be written into the response object.

First you must create a response object instance inside of the compound ISR object. For this action, call the CreateResponseFromRequest method. Subsequently, you can get the pointer to the response object (by calling the GetModifyResponse method) and use the Add/Set/Copy methods of the response class or its base classes to add the data to the response.

You should insert the results of a correct connector operation as well as the error information in the response. In case of a serious exception, you can also throw an exception; see the section "Handling Exceptions".

The request and response instances must remain in the compound object. With the GetModifyRequest and GetModifyResponse, you only receive pointers to these instances so that you can read and modify them. The instances are returned within the compound object after the Modify (or other) connector method returns. Do not delete any SPML object instances you receive from the compound object from inside the connector. They all remain inside the connector and are deleted by the server later on.

1.2.4.4. Using Copy/Add/Set Methods

The response classes provide several methods for adding or modifying the contents of the object. They copy some substructures from other objects (mainly from the requests), add new substructures or set them to a given value.

The methods are generally used to add new substructures to the SPML objects. Sometimes, however, replacing information inside the SPML objects may be required. In this case, you should be aware of the Set/Add/Copy methods' replace semantics. Some of the methods expect to be used to add a new content to the object and not to replace the old content, because replacing the old content could be unintentional and could lead to information loss. These methods will not allow you to directly replace the contents. You should first explicitly remove the substructure from the SPML object and then add a new instance (otherwise you incur an exception). In other cases, the implicit replace semantics are allowed (where it is natural or does not impose a risk for unintentional data loss).

To determine a method’s replacement semantics, consult the comments at its declaration

1.2.4.4.1. Using Transformation Methods

The SPML classes have a built-in mechanism that allows you to transform all text contents of an SPML request/response in a defined way. The ISR classes provide the method TransformStrings for this purpose. This method has two arguments:

- Which – denotes the set of strings that should be transformed inside the ISR instance
- Transformer – a reference to the instance of the transformer object that should be used for the transformation

To carry out a particular transformation of the strings inside a ISR request or response, you must first implement the transformer class. You create your own class derived from the CIsrStringTransformerBase base class. You implement the inherited abstract method Transform, which receives the string as the argument and returns the result of the transformation. Next, you create an instance of this new class and call the TransformStrings method of the ISR class whose strings you want to transform. As the Transformer argument you give the reference to your transformer object instance. The ISR class will apply the defined transformation on all strings that it contains using the transformer.

1.2.4.4.2. Handling Exceptions

ISR classes use their own exception type for reporting errors – EIsrException. Nearly every ISR method can throw this exception. In some cases, the header file comments describe the cause of throwing the exception. You should always catch and interpret these exceptions inside of a connector or filter. When it's unclear as to why the exception was thrown, we recommend that you insert the EIsrException's text message into the generated error message and include with it the identification of the location at which exception was caught (for example, while changing some attribute contents, and so on). You should not rely on the assumption that the exception will not be thrown in a particular case. The exceptions are used to guard the internal consistency of the classes and you can't be sure that a violation will not occur in runtime.

The connectors and filters should never let the EIsrException out to the server.

1.2.4.5. Handling Data Encoding

Note that all character values (strings) that the SPML and ISR class methods return are encoded in ASCII or UTF-8. Similar if you set values you must use ASCII/UTF-8 encoded character values. You cannot use ISO8859-1 (latin-1) or any other encoding. There is, however, a standard filter defined for converting some encodings (or you may write your own filter for that).

1.2.5. About the Connector Server Configuration

The following sources of configuration options are in effect for the C++ connector server:

- Configuration data stored in the directory (LDAP) server
- Configuration data in other configuration files, for example, the logging configuration file

This section describes the configuration data that applies to the connector server and its subcomponents.

1.2.5.1. Server-Wide Configuration

The following options control the connector server as a whole. They are stored with one exception in the directory and consist of the following data:

- Number and list of configured connectors
- Ports for communication over SOAP/HTTP and SOAP/HTTPS

- Accept and receive timeout - control the TCP transmission timeouts for the SOAP communication
- Thread minimum and maximum count for receiver threads - how many threads should receive SPML requests from the network and deliver it to the connectors
- Queue maximum - the maximum number of requests that can wait in a receiver thread queue. If the queue is full, the server will not accept any more requests
- SSL key file path and key password - for using SOAP over HTTPS
- Heap check mechanism - turned on / off (see "About the Heap Check Mechanism" and "C++ Identity Server INI File Configuration")

1.2.5.2. Logging Configuration

The connector server uses the same logging mechanism as the DirX Identity C server. Each logged message is assigned to a particular component and a particular logging level. You can specify which components and logging levels inside them should be logged in the logging configuration file. Separate levels exist for errors and warnings and several debug logging levels are used for debug messages. The following components are relevant for the connector server: cutl, csvr, cwrp, cmgr. "cutl" is a core component of the server; it handles various low-level tasks like thread synchronization or dynamic library loading. "cmgr" is a module for reading configuration data from LDAP or files. "cwrp" is a module that controls the activity of connectors. "csvr" is the central server module that controls the network interface as well as the flow of data through the server.

Connectors log their messages as 'con*n*' components where *n* represents the logging number of the respective connector. See the description of connector configuration.

1.2.5.3. Connector-Dependent Configuration

The following options are available for every configured connector:

- Maximum number of threads - the maximum number of connector class instances that can exist at one time in separate threads. If you have a connector that is not thread safe, use 1 as the maximum.
- Maximum size of the queue - the maximum number of requests that can wait in the connector class's queue. When the queue is full, the server will not accept further requests for the respective connector.
- User, password, type, name, version, server, port - data needed for the connector operation and for accessing the associated target system.

1.2.5.4. C++ Identity Server INI File Configuration

Some rarely-used configuration options are configured in the C++ DirX Identity server's INI file:

- Section [settings], option CConnServer, value 1 or 0 - turns the connector server on general on and off.
- Section [settings], option CConnServer-MemCheck, value 1 or 0 - turns the heap checking mechanism on and off (see the "About the Heap Check Mechanism" section.)

1.2.6. Miscellaneous Information

This section provides some additional information about the C/C++ Connector Integration Framework.

1.2.6.1. Windows Platform Dependencies

The current DirX Identity connector server version was compiled using Microsoft Visual Studio 6. It uses the STLport implementation of the STL library (www.stlport.com). Because STL objects are part of the connector server API, it is necessary to compile the connectors using STLport as well. To find out how to compile and link your source with this library, see the product related documentation. You must ensure that the standard Microsoft STL header files are ignored and instead the STLport headers are included. The same applies to the STL runtime libraries. Note: Do not forget to use the namespace "_STL" instead of "std".

When compiling a DLL on Windows, you must typically define a preprocessor symbol for the DLL linkage (is usually defined in the header file and differently redefined for the import / export case). As a result, the main API header file contains the symbol DXMCCONINTERFACE_EXPORTS. This symbol should be defined when compiling the connector library.

You also need to include some standard compilation symbols. You can find them in the sample connector implementation discussed in the section "Sample Implementation".

Use the "Multithreaded DLL" library.as the C runtime library.

You can also compile the server under Visual Studio .NET 2005 environment as an unmanaged C++ application. This environment has its own multithreading-capable STL implementation so you won't need to use the STLport implementation.

1.2.6.2. Relevant Files and their Function

This document refers to the following files:

- Main API header file ("dxmcconinterface.hpp") - should be included into the connector source. Includes all other required header files.
- Filter API header file ("dxmcconfilterif.hpp") – should be included into the filter source.
- Heap check header file ("dxmcconmembase.hpp") - is automatically included from the main API header file. Contains declarations of classes and macros for the connector server's built-in heap checking mechanism (see the section "About the Heap Check Mechanism")
- Configuration header file ("dxmcconfigmgr.hpp") - contains the configuration classes; see the section "Using the C++ Connector API".
- Filter configuration header file ("dxmcconfigfilter.hpp") – contains the filter configuration class, see the section "Using the C++ Connector API".
- API shared libraries ("libdxmcconinterface.**", "libdxmcconfigmgr.**") – shared libraries (the file extension depends on the target platform) that are necessary to link to any component shared library (connector or filter)

- Logging configuration file (“dirxlog.cfg” under *install_path**/server/conf*) - contains the configuration of the logging subsystem.
- Server INI file (“dxmmsssvr.ini” under *install_path*/**server/conf**) - contains configuration settings for the server. See the section "C++ Identity Server INI File Configuration".
- ISR header files (“dxmSPMLisr.hpp”, “dxmcconctr.hpp”, “dxmisrsup.hpp”) - contain the declarations of ISR classes; see the section "About the ISR Classes".

1.2.7. Sample Implementation

Additional files are available on your DVD in the folder

Additions\SampleConnector\cpp

This folder contains a ZIP file that contains all necessary header and lib files to build your own connector. It also contains an example test connector implementation that you can build with Microsoft Visual Studio C++ 6.0. For more information, see the file ReadMe.txt.

2. Agent Integration Framework

The agent integration framework allows you to integrate custom agents into DirX Identity. You can configure these agents via job descriptions that are controlled by workflows and activities within the C++-based Identity server. Start workflows via schedules or with the **runwf** tool.

The following DirX documentation explains how to integrate your custom agent into the DirX Identity Agent Integration Framework

- *DirX Identity Tutorial* → Follow-on Tutorials → "Integrating a Customer-Specific Agent" - this section of the tutorial provides procedures and examples that explain how to use all of the features that permit custom agents to be integrated into DirX Identity.
- *DirX Identity Customization Guide* → "Customizing Connectivity Objects" - the sections in this chapter provide reference information about the agent integration features illustrated in the tutorial, including how to:
  - Use virtual object extensions to extend the DirX Identity objects with custom attributes.
  - Use design mode to hide custom attributes at specific object instances.
  - Find detailed information about automatic references that can be used to transfer your custom attributes into any type of control file (for example, INI files).
  - Create your own wizards for smooth and easy-to-use configuration of C++-based batch workflows.
  - Use the DirX Identity naming conventions, which make it easier to understand complex object structures.

3. DirX Identity Web Application Programming Interface

You can customize the DirX Identity Web Center application as described in the *DirX Identity Web Center Customization Guide*. You can also build new application parts that use the DirX Identity configuration information and data via the DirX Identity Web API. This API is a guaranteed interface. We strongly recommend that you use the Web API. We do not recommend using direct LDAP access because LDAP structures may change over time. The Web API hides these low-level changes.

You can find the Identity Web API Java documentation in the *DirX Identity Web Center Reference*.

4. SPML Provisioning Web Services

DirX Identity provides Web services that offer a wide range of operations to provision users, privileges, target system objects and business objects to a DirX Identity domain. For an overview of the DirX Identity object model and its basic concepts, see the *DirX Identity Introduction*. For more details, see the *DirX Identity Provisioning Administration Guide*.

The services are deployed into a (Tomcat) Web container. The Web Service interface is provided as a Web Service Description Language (WSDL) file along with a number of imported XML schemata in the **WEB-INF/etc** subfolder.

Using the WSDL and the XML schemata, it is easy to implement clients in various technologies (for example, Java, C#, PHP, and so on). A variety of tools exist to generate client stubs from the WSDL description and XML schemata. DirX Identity delivers a ready-to-run Java SOAP client and Java classes that represent the data entities exchanged at the WSDL interface.

This chapter describes the SPML Provisioning Web Services operations and provides information about other aspects of their operation, like authentication and authorization.

4.1. About the Provisioning Operations

The SPML Provisioning Web Services operations are structured according to the entity types of the DirX Identity model.For each entity type, at a minimum the operations add, modify, delete, lookup and search are supported.

The provisioning operations adhere to the OASIS SPMLv2 standard (see http://www.oasis-open.org/committees/provision/docs/).For a formal definition of the operations and their types, see the WSDL file **provisioning-service.wsdl** and imported files.DirX Identity supports the SPMLv2-DSMLv2 profile (see http://www.oasis-open.org/committees/provision/docs/, document pstc-spml2-dsml-profile-os.pdf).For information about SPMLv2 concepts, see the SPMLv2 standard document (pstc-spml2-os.pdf) and its associated XML schemata.

SPMLv2 requires the following mandatory requests: add, modify, delete, lookup and listTargets.The listTargetsResponse contains the object types (targets), the associated schema and capabilities.In addition to the mandatory requests DirX Identity typically supports the search and reference capabilities for all its objects.For some object types DirX Identity supports further capabilities, such as the match rule capability for roles.See the chapters for the specific object types for more details.

The DirX Identity schema is open for customer extensions.The customer can extend each object type with custom attributes transparent for DirX Identity services.Except when extending the LDAP schema and the object descriptions, nothing special needs to be done for the provisioning services.In contrast to the SPMLv2 concept, the services do not check the attributes against the schema defined in the respective listTargets response.

While the attributes are open, the target identifiers are not.The DirX Identity target identifiers denote the object types (currently users, roles, permissions, targetSystems, accounts, groups and all business objects).See the object type-specific sections for the

appropriate target identifier.

If you need to manage additional custom object types via the Provisioning Web Services, you can deploy custom handlers to extend the Web Service Spring configuration files and leverage existing generic handler Java classes.See the section on "Custom Objects" for details.

For details about the operations, see the subsequent sections, which are organized according to the object types and their specifics.

4.2. Authentication

The SPML Provisioning Web Services support HTTP basic authentication, proprietary authentication based on asymmetric key cryptography or digest authentication based on SSL/TLS protocol and Single Sign-On (SSO) authentication via HTTP header.They also support XML signature verification.The next sections describe how to configure this support in more detail.

4.2.1. Proprietary Authentication

SPML clients can use a proprietary authentication scheme based on asymmetric key cryptography.In this scheme, an SPML client authenticates itself to the SPML Provisioning Web Services servlet with its private key and then forwards only the name of the current user.The servlet verifies that it has a valid certificate of the presented private key and if so, accepts the username without further validation.This scheme is also used for Web Center with a setup of SSO to the Request Workflow server.See the *DirX Identity Web Center Reference* for details.In this case, you need a private key for a client, which is a key generated solely for this purpose; see the section "Creating a Keystore and a Truststore" below for details.

The sample client available in *install_path*/**provisioningServices** can use this type of authentication.The keystore access and the alias of the private key must be configured as a part of the configuration for a special authenticator class (see the sample configuration file *install_path*/**provisioningServices/spmlv2/conf-proprietary-auth.xml**):

```
<!-- Proprietary authentication -->
<authenticator
handler="com.siemens.dxm.provisioning.framework.soap.authentication.P
roprietaryAuthentication" >
  <property name="username" value="cn=Morton Gabriela,o=My-
Company,cn=Users,cn=My-Company"/>
  <property name="keyStore" value="C:/Program Files/Atos/DirX
Identity/ssl/provisioningservices-keystore-localhost"/>
  <property name="keyStoreAlias" value="localhost"/>
  <property name="keyStorePassword" value="<password>"/>
</authenticator>
```

The value of the **username** property is used as an authenticated user at the server side and will be used as an effective user for any invoked operation.

The properties **keyStore** and **keyStorePassword** are used by the test client to access the keystore with the private key used during authentication.

The **keyStoreAlias** property is the alias of the private key stored in the configured keystore. This property allows the client to identify the correct private key within the keystore. For details about the sample client provided with DirX Identity, see the section "Using the Sample Client".

You need to import a certificate that corresponds to the client's private key into the Provisioning servlet truststore named as **provisioningservices-truststore**. The truststore password must match the one with the **truststore** key in the **password.properties** file. Note that the **truststore** key can be missing if the servlet is being redeployed or upgraded. Check to see if the **truststore** key in the **password.properties** exists and add it manually if necessary. Both files must be placed in the following directory:

- *install_path*/**web/instances/provisioningService**-*technical-domain-name*/**WEB-INF** for the servlet deployed to the stand-alone Tomcat.

If the client and the SPML Provisioning Web Services run on different hosts, the keystore must reside on the machine with the client and the truststore on the one hosting the servlet.

You can share the same key for each client, or one key per client, or anything in between. When using different keys, choose a unique alias for each key and add a certificate for each key to the servlet's **provisioningservices-truststore**.

The use of proprietary authentication does not require any special steps except for creating key material for the client and server sides and configuring the **password.properties** file. The authentication itself is already enabled by default when the servlet configuration file **config.xml** located in the folder **WEB-INF** contains following section under the element **authenticators**:

```
<!-- Default Authentication (performs HTTP basic and proprietary
authentication) -->
<default>

<class>com.siemens.idm.service.provisioning.auth.DefaultAuthenticator
</class>
  <trustStore>provisioningservices-truststore</trustStore>
  <trustStorePassword>${password:truststore}</trustStorePassword>
</default>
```

Creating a Keystore and a Truststore

The batch file *install_path*/**ssl/generate_ProvSvcKeystore.bat** (or **.sh**) provides a

convenient way to:

- Create a keystore with a private key for a client *install_path*/**ssl/provisioningservices-keystore-alias**.
- Export a self-signed certificate of that key to the file *install_path*/**ssl/provisioningservices-alias.crt**.
- Add the certificate to the truststore *install_path*/**ssl/provisioningservices-truststore**.

If the truststore doesn't exist, it is created. The keystore and the certificate file are overwritten if they already exist.

Before you run the batch file, open it and set the following values:

- **dname** - a distinguished name intended to identify the client instance(s) the key is generated for.
- **alias** - a unique name for the key. Since the alias gets part of the keystore file name, don't use any special characters in the alias.
- **keystorePassword** - the keystore password; any password you like. The client needs to be configured to use it for access to the keystore.
- **truststorePassword** - the truststore password; any password you like. The servlet needs to be configured to use it for access to the truststore.

Next, move the created keystore to the machine hosting the client and then adapt the client's configuration as described in this section.

Move the truststore to the servlet's **WEB-INF** directory and adapt the **password.properties** file accordingly.

When using the batch file again to generate a key for another client, choose a different **dname** and **alias**.

4.2.2. SSO Header File Configuration

The SSOHeaderFilter is a servlet filter that extracts SSO credentials from an HTTP header and stores them in a session-scoped variable.

The filter can act as a stand-alone SSO module if the client provides unencrypted credentials via an HTTP header. Its other purpose is to copy credentials provided by another SSO module (for example, a Tomcat valve) to session-scope.

Note that the filter does not validate the source of the credentials. So deploying it as a stand-alone SSO module may cause a security breach if untrusted clients can access the Provisioning Web Services directly.

The filter must be defined in the application's deployment descriptor **web.xml** as a filter, as shown in the following example:

```
<filter>
```

```
   <filter-name>
            SSOHeaderFilter
   </filter-name>
   <display-name>
            SSO Header Filter
   </display-name>
   <filter-class>

com.siemens.idm.service.provisioning.sso.filter.SSOHeaderFilter
    </filter-class>
    ... initialization parameters ...
</filter>
```

The filter can be customized via the filter initialization parameters as in this sample:

```
<!-- where to find the authenticated user -->
<init-param>
    <param-name>headerName</param-name>
    <param-value>MyAuthenticatedUser</param-value>
</init-param>
<!--defines what headername contains: user / account -->
<init-param>
    <param-name>type</param-name>
    <param-value>user</param-value>
</init-param>
```

The parameter “headerName” denotes the HTTP header field from where the filter takes the identifier. The value of the header is obtained by calling

- request.getUserPrincipal().getName(), if the header name is UserPrincipal
- request.getRemoteUser(), if the header name is Authorization or RemoteUser
- request.getHeader(headerName), otherwise

Depending on the given “type” parameter, the filter either searches for a user or for an account with the appropriate filters defined in the context parameters in the deployment descriptor. Here is an example:

```
<context-param>
    <param-name>com.siemens.provisioning.auth.userFilter</param-name>
   <param-value>(&amp;(objectclass=dxrUser)(sn=%USER_ID))</param-
value>
```

```
</context-param>

<context-param>
    <param-name>com.siemens.provisioning.ldap.baseDN</param-name>
    <param-value>cn=My-Company</param-value>
</context-param>

<context-param>
    <param-
name>com.siemens.provisioning.auth.targetSystemFilter</param-name>
    <param-
value>(&amp;(objectclass=dxrTargetSystem)(dxrTSDomainName=%DOMAIN))</
param-value>
</context-param>

<context-param>
    <param-name>com.siemens.provisioning.auth.accountFilter</param-
name>
    <param-
value>(&amp;(objectclass=dxrTargetSystemAccount)(dxmADsSamAccountName
=%ACCOUNT))</param-value>
</context-param>
```

If the header identifies a user, the filter takes the *userFilter* search expression. Otherwise, it searches for an account in the configured target system with the accountFilter. It finds the corresponding user DN via the account's dxrUserlink attribute.

4.2.3. XML Signature Verification Configuration

For password reset scenarios, clients can sign some requests - especially setPassword or checkChallengeResponses - with the end user's private key. This mechanism serves not only to authenticate, but also to ensure that the new password is for the same user as the one who produced the signature.

To use this Provisioning Web Services feature, XML signature verification needs to be enabled and configured. Verification is implemented as Apache CXF WS-Security interceptors based on the Apache WSS4J module. For more details on identifying attributes, see the subsection "Identifying Attributes" in the section "Common SPML Aspects".

The feature is activated with three servlet initialization parameters in the deployment descriptor WEB-INF/web.xml. By default, the feature is disabled. The following snippet shows the relevant parameters, which you only need to uncomment and adapt:

```
<init-param>
       <param-name>wsSecurityEnabled</param-name>
       <param-value>true</param-value>
</init-param>
<init-param>
       <param-name>wsSecuritySignatureIsOptional</param-name>
       <param-value>false</param-value>
</init-param>
<init-param>
       <param-name>wsSecuritySignaturePropFile</param-name>
       <param-value>crypto.properties</param-value>
</init-param>
```

Set the **wsSecurityEnabled** parameter to **true** to activate XML signature verification for the servlet; the default is **false** (feature disabled).

Set the **wsSecuritySignatureIsOptional** parameter to **true** if you expect and accept both signed and non-signed requests. If it is set to **false** (the default), the interceptor rejects non-signed requests; if set to **true**, it leaves the proper handling to the corresponding Web Service operation. This setting is especially important for challenge/response scenarios, where the user authenticates with their challenges rather than with a certificate. See the sub-section "Authentication by Challenge/Response" (in the section "Accounts Management") for details.

The **wsSecuritySignaturePropFile** parameter references the file **crypto.properties** (the default value); you may specify a different file name here. In a default deployment, this file is located in the WEB-INF/classes folder. The file format is defined by Apache WSS4J. It especially sets the class name of the Crypto provider (in this example: **MyMerlin**) and the location of the server key store. Here are the relevant sample snippets:

```
org.apache.wss4j.crypto.provider=com.siemens.idm.service.provisioning
.wssecurity.MyMerlin
org.apache.wss4j.crypto.merlin.file=C:\\DirXIdentity\\ssl\\server-
keystore
```

You need to adapt the path to the server key store.

> The property key prefix changed from **org.apache.ws.security.crypto.** to **org.apache.wss4j.crypto.** as part of the migration of the Provisioning Web Services from Apache Axis 1.x/JAX-RPC to Apache CXF/JAX-WS (with WSS4J 4.x). If you are upgrading an existing deployment, update the keys in your **crypto.properties** file accordingly.

When the XML signature verification is successful, the WS-Security interceptor checks that

the certificate matches the identifying attributes in the request. You need to configure the attribute names in the certificate that must be compared with the identifying attributes in the setPassword request. This information is stored in the file WEB-INF/identifierMatcherConfig.xml. The element <identifyingAttributes> contains a XPATH expression to the sub-element in the request, which contains the identifying attributes. There is no need to change the default "identifyingAttributes". The attribute names need to be configured in <matchRule> elements, as shown in the following snippet:

```
<matchRules>
        <matchRule op="equals" subset="all">
              <op1 source="cert"  name="rfc822name" extract="(.*)"/>
              <op2 source="psoID" name="user.mail" extract="(.*)"/>
  </matchRule>
        <matchRule op="equals" subset="all">
              <op1 source="cert"  name="subject.serialnumber"
extract="(.*)"/>
              <op2 source="psoID" name="serialnumber"
extract="(.*)"/>
        </matchRule>
 </matchRules>
```

Enter a <matchRule> for each attribute.The XML attribute **op** sets the comparison operator, typically "equals".Alternatives are: "startsWith", "endsWith" and "contains".In <op1>, configure the attribute name in the certificate and a Java regular expression for extracting the required value - typically **all**.In <op2>, do the same thing for the identifying attribute in the SPML request.Note the prefix (here: **user**) that indicates to the Web Service that this attribute identifies the user.

4.3. Authorization

An SPML Provisioning Web Service processes the request on behalf of the authenticated user.It especially applies the access policies of the Identity domain.The domain administrator needs to make sure that access policies exist that entitle the client user to modify the requested users and to grant the privileges.

4.4. Runtime Operation

This section describes the SPML Provisioning Web Services runtime, including how to:

- Deploy the Web Services runtime
- Adapt Web Services Runtime Configuration
- Customize the Web Services runtime

4.4.1. Deploying the Provisioning Servlet

This section describes how to deploy the Provisioning Servlet.There are the following deployment options:

- Deployment into Tomcat - use this deployment type in productive environments.

4.4.1.1. Understanding the Deployment Concept

The Provisioning Servlet installation includes these template folders in the folder *install_path*/**provisioningWebServices**:

- **provisioningServlet.template** - for creating a domain-specific Provisioning Servlet instance and then deploying this instance into a Tomcat server.

These folders contain files with placeholders in the **WEB-INF** subfolder:

- **application.xml** contains the placeholder @DOMAIN@.
- **config.xml** contains the placeholders @DOMAIN@, @HOST@, @PORT@, @SSL@.
- **web.xml** contains the placeholder @DOMAIN@.
- **password.properties** contains the placeholder @TECHNICAL_PW@.

The placeholders and their meanings are:

- @DOMAIN@ - the domain name
- @HOST@ - the host for accessing the Provisioning store
- @PORT@ - the port number for accessing the Provisioning store
- @SSL@ - the SSL flag (true or false)
- @TECHNICAL_PW@ - the password of the technical user.

To deploy the servlet for a domain *domain*, with technical domain name *technical-domain*:

- Create a deployment instance folder in *install_path*/**web/instances**. For deployment into Tomcat, this is a copy *provisioningServlet-***technical_domain** **of** ***provisioningServlet.org**.
- Replace the placeholders in the relevant files - not in the template folders, but in the deployment instance folder.

4.4.1.2. Deployment into Tomcat

Create the folder structure *tomcat_install_path*/**conf/Catalina/localhost** (case sensitive) if it does not yet exist. For UNIX platforms, ensure that the related folders have the appropriate permissions (755). The file **WEB-INF/password.properties** must be readable and writable for Tomcat (and the password administrators) but should not be readable by anyone else.

Run the Configuration Wizard, selecting **Provisioning Web Service Configuration** as a step. The domain name *domain* is then obvious from the subsequent configuration dialog. For complete details on this task, see the "Configuration" chapter in the *DirX Identity*

Installation Guide. The Configuration Wizard:

- Stops Tomcat (on Windows only).
- Creates a deployment instance folder *install_path*/**web/instances/provisioningServlet-***technical-domain-name*.
- Replaces relevant placeholders in the relevant files according to the deployment concept, storing the relevant password in a safe way.
- Creates a copy *tomcat_install_path*/**conf/Catalina/localhost/ProvisioningService-***technical_domain_name***.xml** of **ProvisioningService.xml** and replaces the placeholder @DOCBASE@ in that copy with the full path of the instance folder, using forward slashes as pathname separators for all platforms.
- Runs the pre-processed script **postInstallPSV.bat** (**postInstallPSV.sh** for UNIX) in the subfolder **endorsed** of the deployment instance folder and checks the output in **postInstallPSV.log**.

If you intend to use SSL connections (server-side SSL) between Web Service clients and the Web Services, perform the related steps described in "Establishing Secure Connections with SSL" in the *DirX Identity Connectivity Administration Guide*.

To test your deployment:

- Restart Tomcat.
- A WSDL should be displayed when requesting the URL **http://***tomcat_host:tomcat_port*/**ProvisioningService-***technical_domain_name*/**services/Spmlv2RequestService?wsdl** (or **https://***tomcat_host:tomcat_secure_port*/**ProvisioningService-***technical_domain_name*/**services/Spmlv2RequestService?wsdl** for SSL connections, respectively)

4.4.2. Configuring the Web Services Deployment

In this step, you adapt the files in your deployment instance folder to your requirements. In this section, relative file locations are relative to the deployment instance folder. To configure a Web Services deployment:

- Stop Tomcat or the Java-based Server, respectively.
- If the Identity Store is based on DirX Directory, the number of pagedReads per Directory installation is limited. This limit is configurable, as are the paging policies with respect to paging timeout. Consult the DirX Directory documentation for information about paging policies and supported controls for further details.
- Configure log level and bind parameters in the file **config.xml**. It is located in the folder **WEB-INF**.
- Configure the file **web.xml** with respect to these initialization parameters:
- Parameter **cleanupInterval** - controls the periodic cleanup of paged search results overhead for the Web Services.
- Parameter **pageTimeout** - specifies the maximum lifetime in seconds of paged result

overhead by the Web Services. If a paged result has not been accessed for a period longer than **pageTimeout**, the related overhead will be cleaned up during the next cleanup cycle (see also **cleanupInterval**).

- Parameter **listTargetsResponseFile** - this parameter is no longer evaluated. Instead, the list of file names is stored in the Spring configuration file **applicationContext.xml.**
- Parameter **spmlv2.defaultpagesize** - specifies the maximum number of entries to be returned in a page of a paged search result. This parameter is only valid for SPMLv2 search requests.
- Parameter **passwordFile** - specifies the properties file containing the passwords. To use a file from within the Provisioning Servlet folder, specify the file name relative to the folder, like **WEB-INF/password.properties** or **/WEB-INF/password.properties** (which is also the default location). To use a file located outside the Provisioning Servlet folder, specify the absolute path name. This way, you can use a common password file with the IdS-J server or Web Center. When specifying a non-default location, ensure that the <init-param> definition for the password file is no longer enclosed in XML comments. The following sample would reference the IdS-J file *install_path*/**ids-j-***domain*-**S** *n*/**private/password.properties**.
- Configure the file **config.xml** so that the password placeholder in the technical bind section is consistent with the **passwordFile** specification in **web.xml**. For the default or a Web Center password file, the correct placeholder is **${password.ldap}**. For an IdS-J password file, the correct placeholder is **${password.domain}**.

> We recommend creating backup copies of the **config.xml** and **web.xml** files outside any template folder or deployment instance folder so that they are not overwritten on an update installation or a repeated deployment.

To test the deployment, start Tomcat and then test the deployment as indicated in the section "Deploying the Provisioning Servlet".

4.4.3. Customizing the Runtime

This section describes how to customize Web Services runtime to:

- Provide a subset of the offered services
- Extend the LDAP schema with your attributes

4.4.3.1. Providing a Service Subset

A default deployment of the Provisioning Web Services provides all implemented operations on all supported object types or Provisioning Service targets. In some circumstances, you might want to provide only a subset of the functionality to the Web Service clients, for example:

- Allow changing of only a subset of the object types.
- Allow only some of the operations per object type.

The following figure illustrates the configuration files:

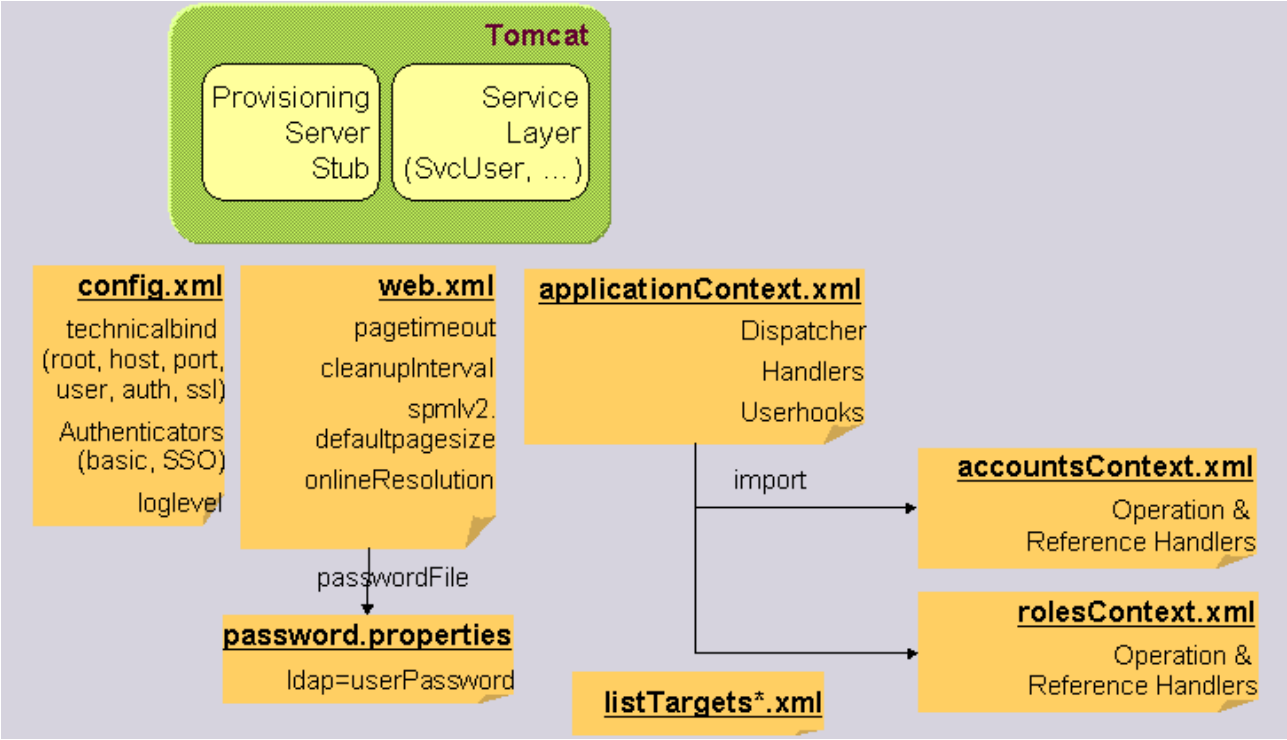

Figure 7. Provisioning Web Services Configuration Files

The options in the files **config.xml** and **web.xml** are described in the section "Configuring the Web Services Deployment".

The supported object types and their handlers are configured in the **applicationContext.xml** and the other context files, which it imports. This configuration is based on the Open Source dependency injection framework "Spring". It defines the request dispatchers and operation handlers as Spring beans. Dispatching an incoming request is realized in 2 stages:

- The SPMLv2 dispatcher forwards the request according the operation; for example, an addRequest is forwarded to the AddDispatcher, a modifyRequest is forwarded to the ModifyDispatcher, and so on.
- The operation specific dispatcher forwards the request according the target identifier; that is, the object type. As an example, the AddDispatcher inspects the targetID of the request and if this equals "users", it forwards the request to the "AddUser" handler.

The dispatcher beans are configured in the context files. The main source for dispatching are tables (or Java maps). The Spmlv2Dispatcher works on a handler map with the operation as key and the name of the associated handler as value. The operation dispatchers work on a handler map with the target identifier as key and the handler bean name as value. All handlers responsible for an object type are configured in a context file for this object type. For example, the handlers for user management are defined in the file **usersContext.xml**.

The only exception to this is the handler for the listTargets request. This is the only request without a target identifier. Its handler is configured in the handler map for the SPMLv2 dispatcher, and it contains a map for all listTargets response files. In case of a listTargets request, the handler concatenates the content of all configured response files and returns

it in one huge listTargets response.

4.4.3.1.1. Restricting Object Types

Suppose, for example, that you want to provide services only for the management of users and not for any other objects types such as roles or business objects. You can achieve this by simply reducing the list of list target response files in the Spring bean configuration file **applicationContext.xml** in the deployment instance.

This list is configured for the bean “ListTargetsHandler” in the property “target2ResponseMap”. The key for each entry in the map is the target identifier, the value is the corresponding name of the listTargets response file. In response to a SPMLv2 listTargetsRequest, the Web Service returns the accumulated targets of these responses. Each pathname is relative to the deployment folder instance. The related file must exist, and it must be syntactically correct - otherwise the Web Service will fail when starting up.

Optionally, you can also remove the target-specific context files and their references from within **applicationContext.xml**. But be careful to remove all references to the skipped files.

4.4.3.1.2. Restricting Operations per Object Type

Suppose you do not want to support all available operations for an object type; for example, you do not allow the password capability for users. Then you must remove the associated handlers from the dispatcher map(s). If you do not want to support password management at all, you can remove the respective handlers from the SPMLv2Dispatcher’s handler map: these are the dispatchers for "setPasswordRequest" and "validatePasswordRequest".

If you want to support password updates for accounts, but not for users, you need to change the handler map for the "SetPasswordDispatcher": remove the line for the "users" key.

4.4.3.2. Extending the LDAP Schema

If you extend the LDAP schema with your custom attributes, make sure that these extensions are reflected in the corresponding object description and list targets response.

In the following example, the role objects are extended with an attribute "myAttribute":

- In the element "target/schema/schema" of the file **listTargetsRspRoles.xml** specify the new attribute with the following XML element:

```
<spmldsml:attributeDefinition name="myAttribute"/>
```

- Then add the attribute to the object class definition of “dxrRole” with the following XML element:

```
<spmldsml:attributeDefinitionReference name="myAttribute"/>
```

4.5. Common SPML Aspects

The next sections describe aspects that are common to the Provisioning Web Services' use of SPML.

4.5.1. Identifiers

The identifiers in all requests are interpreted as the DNs (LDAP distinguished names) of the objects or containers.

If the add request provides an identifier (element <psoID>), the service tries to create the object or container with this DN.If the add instead contains a <containerID>, the service takes it as the DN of the parent container and creates an object beneath it.The RDN (relative DN) must either be passed as an attribute in the request or the object description must calculate a default value.

4.5.2. Identifying Attributes

In some cases, the identifying DN is not known; this is especially true when the user has forgotten the password. In this case, you can set up to pass a list of identifying attributes as a sub-element of the psoID. Here is a sample snippet as part of a setPasswordRequest:

```
<pass:psoID targetID="accounts">
    <idattr:identifyingAttributes>
           <dsml:attr name="user.uid"><dsml:value>user’s
GID</dsml:value></dsml:attr>
           <dsml:attr
name="user.mail"><dsml:value>…</dsml:value></dsml:attr>
           <dsml:attr
name="dxrName"><dsml:value>…</dsml:value></dsml:attr>
           <dsml:attr
name="ts.dxrTSDomainName"><dsml:value>…</dsml:value></dsml:attr>
      </idattr:identifyingAttributes>
</pass:psoID>
```

In this sample, the attributes are used to identify an account within a target system. The account is identified by the attribute "dxrName", the target system by "dxrTSDomainName" – for Active Directory, the domain name.

Prefixes in the attribute names are used to identify related entries. For accounts, the prefix "ts." denotes a target system. The prefix "user." denotes the associated user of an account.

For cases where the client does not know about the LDAP attribute names, the Web Service provides an attribute name mapping. This mapping is configured in the Spring bean definition of the accounts handlers. Because a couple of handlers need them, the mapping is separated into an extra bean, which is referenced by all the handlers that need

it.

The section "Customizing the Runtime" gives an overview of the Web Service customization. The settings for changing an account password are configured in the file **accountsContext.xml**. The bean **AttributeNameMapping** defines the attribute mapping, as shown in this example:

```
<bean id="AttributeNameMapping" class="java.util.HashMap">
     <constructor-arg>
           <map>
                 <!--
                 key: name in request excluding prefix in lower case
(!!!),
                 value: name in LDAP without prefix
                 -->
                <entry key="gid" value="uid"/>
                <entry key="domain" value="dxrTSDomainName"/>
                <entry key="logonname" value="dxrName"/>
           </map>
     </constructor-arg>
</bean>
```

The mapping is configured as a Java map. For the key, you enter the attribute name as sent by the client, but in lowercase format. For the value, you enter the name of the attribute in LDAP format. Note that the names in the map must not contain the prefixes. So "ts.domain" in the request is translated to a target system attribute “dxrTSDomainName”.

Beans that need this mapping reference it as shown here for the setPassword handler:

```
<bean id="SetPasswordAccountsHandler"
class="com.siemens.idm.service.provisioning.spmlv2.accounts.SetPasswo
rdHandler"
scope="prototype" parent="BasicAccountsHandler">
        <property name="attributeNameMapping"
ref="AttributeNameMapping"/>
</bean>
```

4.5.3. Add, Modify and Delete

Besides the normal list of attributes or modifications, add and modify requests may contain the reference capabilities.

To simplify deployment, attribute names are not cross-checked with the definitions made

in the respective listTargets response. Hence - not in line with the OASIS SPMLv2 specification - you don't have to change the listTargets response file, once you extend the LDAP schema and add new attributes or object classes.

For the management of completely new entry types not supported by the DirX Identity domain, see the section "Custom Objects".

You may want to define additional custom object descriptions that extend existing default object descriptions. A sample might be to distinguish between several types of users. In this case keep in mind that the Provisioning Service uses the default object description for creating a new object. For users, this is the "dxrUser" object description. This means that you can create your custom extended user entry, if you provide the appropriate object classes and attributes. Rules for calculating default attribute values are only taken from this default object description, not from your custom one.

When the Service processes a modify or delete, it obtains the appropriate object description the same way as Web Center or Identity Manager by applying the matching rules defined in the object description. This means especially, it associates the proper object description and its rules for attribute values.

4.5.4. Search

Search requests are always processed as LDAP simple paged search requests. The default page size may be changed in the web application's initial parameters.

4.5.5. Iterate

The **iterate** request allows requesting the next pages of a search result.The prerequisite is an initial search having returned a search response that contains an iterator identifier.The request must contain the iterator identifier obtained from a previous search or iterate response.

As it is for a search response, a successful **iterate response** contains the entries of the next page and - optionally - an iterator identifier for requesting the next pages.Absence of the iterator identifier indicates that no more pages are available and the initial search result can no longer be iterated upon.

4.5.6. SPMLv2 Close Iterator

The SPMLv2 **closeIterator request** allows you to inform the Web Services that the related search result is no longer needed.On receipt of such a request, the Web Services will delete the internal overhead related to this result.It is good style for a client to send a closeIterator request when a search result is no longer needed.

The **closeIterator response** returns success or an error message in case of failure.

4.6. User Management

Users of a DirX Identity domain can be managed by SPMLv2 requests using the target identifier "users".In addition to the mandatory operations add, modify, delete and lookup

the provisioning service supports the following capabilities for users:

- Search, namespaceURI = “urn:oasis:names:tc:SPML:2.0:search”.The SPMLv2 search capability applies both to users and users containers.As user containers you can have: organizations, organizational units, countries, locations.
- Reference capability.The user service supports a lot of reference types:
- Simple references; for example, to organizations, manager.See the listTargets response file for a complete list.
- dxrRoleLink: user-role assignments.They include attributes such as start and end date and also role parameters.
- dxrPermissionLink: user-permission assignments with start and end date.
- dxrGroupLink: user-group assignments with start and end date.
- Password capability.This capability allows setting and validating passwords.
- Suspend capability. This capability allows activating and disabling users.

DirX identity provides samples for requests and responses in the folder:

install_path/**provisioningServices/spmlv2/users**.

You must provide appropriate DirX Identity access policies for the requesting user.

4.6.1. Add, Modify and Delete

Add

Besides the normal attributes and references to other entities (see the sections on references), the add request supports a sub-element <operationalAttributes>. The operational attributes are defined for the XML namespace "urn:siemens:dxm:provisioning:operAttrs:1:0" in the schema file "operAttrs.xsd". As in SPMLv1 they define a list of DSML attributes that have to be evaluated by the addressed web service.

The user web service evaluates the attribute "**timeout**":

```
<spml:addRequest returnData="identifier" requestID="usAdd-01"
targetID="users">
<opat:operationalAttributes>
<dsml:attr name="timeout"><dsml:value>300</dsml:value></dsml:attr>
</opat:operationalAttributes>
<spml:containerID  ID="o=testOrg,cn=Users,cn=My-Company"/>
...
</spml:addRequest>
```

It is of interest when the user creation is delegated to a request workflow. In case a timeout

is given, the service waits a maximum of "timeout" seconds until the end of the workflow. If the workflow is not yet finished, the service returns the status "PENDING". Note that - in contrast to the SPMLv2 specification - the client cannot ask for the status of the request or cancel it. In other words: the operations "cancel" and "status" are not supported.

A user can be created *without passing its DN, parent DN or a naming attribute*. By default, this user will be put into the users subtree (cn=Users,cn=<domain>) with the generated dxrUid as the naming attribute. The folder where the user should be created can be changed in the Spring configuration: in the file usersContext.xml, set the defaultRelUserBase property of the AddUser bean to the required relative DN (DN omitting the domain root), as shown in the following snippet:

```
<property name="defaultRelUserBase">
      <value>cn=Users</value>
</property>
```

You can override the default value for the naming attribute by configuring a naming rule in the user's object description. Here is an example:

```
<property name="cn" mandatory="true" type="java.lang.String"
dependson="sn, givenName" >
   <extension>
       <namingRule>
           <reference baseObject="SvcUser" attribute="givenName" />
               <fixedValue value=" "/>
               <reference baseObject="SvcUser" attribute="sn" />
               <fixedValue value=" "/>
               <reference baseObject="SvcUser" attribute="dxrUid" />
        </namingRule>
   </extension>
</property>
```

For the definition of the naming rule, keep in mind that a temporary value is set when the entry is created in memory and before the other attributes are set. Therefore, the property must depend on all the attributes that are used in the rule.

Delete

The delete operation can be applied both to users and to containers. If a container is not empty, you have to supply the attribute "recursive" with a value of "true". Otherwise the request is rejected. If auditing is enabled for users, the user entry is not immediately deleted. Instead its state is set to DELETED, the delete date is set and an audit record is stored in its history attribute. Only when the history attribute is read and cleared, a subsequent consistency rule will delete the entry physically.

The following gives an example of a delete request for a non-empty container:

```
<spml:deleteRequest requestID="usDelOrg-01" recursive="true">
<spml:psoID ID="o=testOrg,cn=Users,cn=My-Company"  targetID="users"/>
</spml:deleteRequest>
```

4.6.2. Attributes

There are no mandatory attributes anymore; even the naming attribute cn and the object class can be omitted. The object class attribute helps to distinguish users and containers in add requests. An object class value of "dxrUser" identifies a user. Otherwise the object is expected to become a container. If no object class is given, the entry is supposed to be a user.

Other attributes are optional. Standard attributes are given in the listTargets response. If you extend the LDAP schema with your custom user attributes, make sure you also extend the user's object description. Then you will be able to update and read them via the provisioning Web Services.

An important standard attribute is dxrState. For the supported values and their meaning, see the overview in the *DirX Identity Provisioning Administration Guide*.

4.6.3. References

All SPML operations support the reference capability. References cover relationships to other objects in the DirX Identity domain.

Most of the references are simple ones. They consist simply of the DN link to the other object. In most cases the name of the reference type equals the LDAP attribute where it is stored.

Some of the references are attributed and support specific reference data. The most important are the user-privilege assignments with their start and end date and account-group memberships with their state. See the respective sections for more information.

In order to search for users with a specific reference, use the <hasReference> clause within a search query:

```
<spmlsearch:query scope="subTree" targetID="users">
<spmlsearch:basePsoID ID="cn=Users,cn=My-Company"/>
<spmlsearch:and>
   <dsml:filter>
       <dsml:equalityMatch name="objectclass">
           <dsml:value>dxrUser</dsml:value>
       </dsml:equalityMatch>
   </dsml:filter>
```

```
   <spmlref:hasReference typeOfReference="dxrSecOrganizationLink">
       <spmlref:toPsoID ID="o=My-
Company,cn=Companies,cn=BusinessObjects,cn=My-Company"/>
   </spmlref:hasReference>
</spmlsearch:and>
<dsml:attributes>
   <dsml:attribute name="cn"/>
</dsml:attributes>
</spmlsearch:query>
```

The above query searches for users with a reference to the organization "My-Company".

If you want to see reference data in the lookup or search response you have to supply the element <includeDataForCapability> for the reference capability.

4.6.4. Reference dxrRoleLink

The reference "dxrRoleLink" controls user-role assignments. The role assignment supports the attributes start and end date and role parameters.

When you want to add a start or end date within the reference data you just have to supply them as a normal DSML attribute. See the following snippet for a sample:

```
<dsml:attr
name="dxrStartDate"><dsml:value>20071231230000Z</dsml:value></dsml:at
tr>
```

If you want to modify a start or end date, provide a DSML modification in the reference data as in the following snippet:

```
<modification modificationMode="replace">
<capabilityData
capabilityURI="urn:oasis:names:tc:SPML:2:0:reference">
<spmlref:reference
xmlns:asgpar="urn:siemens:dxm:provisioning:asg:param:1:0"
         typeOfReference="dxrRoleLink">
<spmlref:toPsoID ID="cn=Signature Level 2,cn=General,cn=Corporate
Roles,cn=RoleCatalogue,cn=My-Company"/>
         <spmlref:referenceData>
             <dsml:modification name="dxrStartDate"
operation="delete">
             </dsml:modification>
```

```
            <dsml:modification name="dxrEndDate" operation="add">
                <dsml:value>29991231235959Z</dsml:value>
            </dsml:modification>
        </spmlref:referenceData>
</spmlref:reference>
</capabilityData>
</modification>
```

Note that role assignments with role parameters might not always be uniquely identified by the distinguished name of the assigned role, so it might be necessary to add a "uid" attribute identifying the assignment. You can find an example in the following snippet:

```
<modification modificationMode="replace">
<capabilityData
capabilityURI="urn:oasis:names:tc:SPML:2:0:reference">
<spmlref:reference
xmlns:asgpar="urn:siemens:dxm:provisioning:asg:param:1:0"
        typeOfReference="dxrRoleLink">
<spmlref:toPsoID ID="cn=Project Member,cn=Project
Specific,cn=Corporate Roles,cn=RoleCatalogue,cn=My-Company"/>
        <spmlref:referenceData>
            <dsml:modification name="dxrStartDate"
operation="delete">
            </dsml:modification>
            <dsml:modification name="dxrEndDate" operation="add">
                <dsml:value>29991231235959Z</dsml:value>
            </dsml:modification>
            <ns1:paramAsg
xmlns:ns1="urn:siemens:dxm:provisioning:asg:param:1:0"
                uid="uid-a5c14a4-7888ad42-152b222a1ac-7e7f">
            </ns1:paramAsg>
        </spmlref:referenceData>
</spmlref:reference>
</capabilityData>
</modification>
```

In this case, the value of the "uid" is the value of the "cn" attribute of the assignment object containing the assignment data in LDAP.

The reference data of role parameters follow the XML schema with the namespace "urn:siemens:dxm:provisioning:asg:param:1:0". You'll find the schema in the list targets response file for users and in "paramAsg.xsd".

In an add request, you supply a role parameter in a <paramAsg> element as in this sample. Note that "asgpar" is the namespace prefix for the above mentioned schema namespace, which you have to define in the XML document somewhere before:

```
<paramAsg>
   <asgpar:paramValue dn="cn=Project,cn=My-
Company,cn=RoleParams,cn=Customer Extensions,cn=Configuration,cn=My-
Company"
                 uid="uid-7f001-cee271-fed935aa6d--7eb4">

<asgpar:value>cn=OptimizeIT,cn=Projects,cn=BusinessObjects,cn=My-
Company</asgpar:value>
    </asgpar:paramValue>
</paramAsg>
```

The "dn" attribute denotes the distinguished name of the role parameter, the "uid" its "dxrUid" attribute. Only one of the two attributes is necessary to identify the role parameter. Add one or more values in <asgpar:value> elements.

Modifications to a role parameter must be supplied in <paramAsgModification> elements as in the following sample:

```
<asgpar:paramAsgModification uid="uid-a5c14a4-7888ad42-152b222a1ac-
7e7f">
  <asgpar:paramValueModification uid="uid-7f001-cee271-fed935aa6d--
7eb4" operation="add">

<asgpar:value>cn=MoreCustomers,cn=Projects,cn=BusinessObjects,cn=My-
Company</asgpar:value>
  </asgpar:paramValueModification>
</asgpar:paramAsgModification>
```

The "uid" attribute identifies the assignment (attribute "cn" in LDAP), whereas the "uid" attribute of the element <paramValueModification> identifies the role parameter (that is, its attribute "dxrUid"). As with normal SPML / DSML modifications, the "operation" attribute tells the service whether to add, replace or delete the value(s). Note that a delete operation without any value deletes all existing values.

a delete modification with only the <toPsoID> element given deletes all assignments for this role:

```
<modification modificationMode="delete">
```

```
<capabilityData
capabilityURI="urn:oasis:names:tc:SPML:2:0:reference">
<spmlref:reference typeOfReference="dxrRoleLink">
         <spmlref:toPsoID  ID="cn=Project Member,cn=Project
Specific,cn=Corporate Roles,cn=RoleCatalogue,cn=My-Company"/>
</spmlref:reference>
</capabilityData>
</modification>
```

The modification shown above deletes all assignments for the role "cn=Project Member".

The modification of dxrRoleLink with the replace operation has a special semantic. It can be used either for update operation of an existing role assignment defined by the "dn" or "uid" attribute or for creation of a new role assignment if supplied with common attributes as shown in the following snippet:

```
<modification modificationMode="replace">
<capabilityData
capabilityURI="urn:oasis:names:tc:SPML:2:0:reference">
<spmlref:reference
xmlns:asgpar="urn:siemens:dxm:provisioning:asg:param:1:0"
        typeOfReference="dxrRoleLink">
<spmlref:toPsoID ID="cn=Marketing Tasks,cn=Department
Specific,cn=Corporate Roles,cn=RoleCatalogue,cn=My-Company"/>
        <spmlref:referenceData>
            <dsml:attr name="dxrEndDate">
                <dsml:value>29991231235959Z</dsml:value>
            </dsml:attr>
        </spmlref:referenceData>
</spmlref:reference>
</capabilityData>
</modification>
```

The replace operation for dxrRoleLink does not delete any existing role assignments.

To search for users with a specific permission assignment, use the <hasReference> clause within a search query:

```
<spmlsearch:query scope="subTree" targetID="users">
<spmlsearch:basePsoID ID="cn=Users,cn=My-Company"/>
<spmlsearch:and>
    <dsml:filter>
```

```
        <dsml:equalityMatch name="objectclass">
            <dsml:value>dxrUser</dsml:value>
        </dsml:equalityMatch>
    </dsml:filter>
    <spmlref:hasReference typeOfReference="dxrRoleLink">
        <spmlref:toPsoID ID="cn=Project Member,cn=Project
Specific,cn=Corporate Roles,cn=RoleCatalogue,cn=My-Company"/>
    </spmlref:hasReference>
</spmlsearch:and>
<dsml:attributes>
    <dsml:attribute name="cn"/>
</dsml:attributes>
</spmlsearch:query>
```

The filter shown above searches for users who have the role "cn=Project Member". Note that reference data such as start or end date are not supported.

For complete samples of how to manage role references, see especially the file "sampleRequestRoleLink.xml".

4.6.5. Reference dxrPermissionLink

The reference "dxrPermissionLink" controls user-permission assignments. The permission assignment supports the attributes start and end date.

When you want to add a start or end date within the reference data, you just need to supply them as normal DSML attributes. See the following snippet for a sample:

```
<dsml:attr
name="dxrStartDate"><dsml:value>20071231230000Z</dsml:value></dsml:at
tr>
```

If you want to modify a start or end date, provide a DSML modification in the reference data as in the following snippet:

```
<modification modificationMode="replace">
<capabilityData
capabilityURI="urn:oasis:names:tc:SPML:2:0:reference">
<spmlref:reference  typeOfReference="dxrPermissionLink">
<spmlref:toPsoID ID="cn=Signature Level 2,cn=General,cn=Corporate
Permissions,cn=Permissions,cn=My-Company"/>
          <spmlref:referenceData>
```

```
                <dsml:modification name="dxrStartDate"
operation="delete">
                </dsml:modification>
                <dsml:modification name="dxrEndDate" operation="add">
                    <dsml:value>29991231235959Z</dsml:value>
                </dsml:modification>
            </spmlref:referenceData>
</spmlref:reference>
</capabilityData>
</modification>
```

The modification shown above replaces the assignment to the permission "cn=Signature Level 2". It deletes the previous start date and sets the end date to the end of the year 2999.

The modification of dxrPermissionLink with the replace operation has a special semantic. It can be used either for update operation of an existing permission assignment defined by the "dn" attribute or for creation of a new permission assignment if supplied with common attributes, as shown in the following snippet:

```
<modification modificationMode="replace">
<capabilityData
capabilityURI="urn:oasis:names:tc:SPML:2:0:reference">
<spmlref:reference
xmlns:asgpar="urn:siemens:dxm:provisioning:asg:param:1:0"
        typeOfReference="dxrPermissionLink">
<spmlref:toPsoID ID="cn=Standard Class,cn=Suppliers,cn=B2B
Permissions,cn=Permissions,cn=My-Company"/>
        <spmlref:referenceData>
            <dsml:attr name="dxrEndDate">
                <dsml:value>29991231235959Z</dsml:value>
            </dsml:attr>
        </spmlref:referenceData>
</spmlref:reference>
</capabilityData>
</modification>
```

The replace operation for dxrPermissionLink does not delete any existing permission assignments.

To search for users with a specific permission assignment, use the <hasReference> clause within a search query:

```
<spmlsearch:query scope="subTree" targetID="users">
<spmlsearch:basePsoID ID="cn=Users,cn=My-Company"/>
<spmlsearch:and>
       <dsml:filter>
           <dsml:equalityMatch name="objectclass">
               <dsml:value>dxrUser</dsml:value>
           </dsml:equalityMatch>
       </dsml:filter>
       <spmlref:hasReference typeOfReference="dxrPermissionLink">
           <spmlref:toPsoID ID="cn=Signature Level
2,cn=General,cn=Corporate Permissions,cn=Permissions,cn=My-Company"/>
       </spmlref:hasReference>
</spmlsearch:and>
   <dsml:attributes>
       <dsml:attribute name="cn"/>
   </dsml:attributes>
</spmlsearch:query>
```

The filter shown above searches for users with the permission "cn=Signature Level 2". Note that reference data such as start or end date are not supported.

For complete samples of how to manage role references, see especially the file "sampleRequestPermissionLink.xml".

4.6.6. Reference dxrGroupLink

The reference "dxrGroupLink" controls user-group assignments. The group assignment supports the attributes start and end date.

When you want to add a start or end date within the reference data, supply them as normal DSML attributes. See the following snippet for a sample:

```
<dsml:attr
name="dxrStartDate"><dsml:value>20071231230000Z</dsml:value></dsml:at
tr>
```

If you want to modify a start or end date, provide a DSML modification in the reference data, as shown in the following snippet:

```
<modification modificationMode="replace">
<capabilityData
capabilityURI="urn:oasis:names:tc:SPML:2:0:reference">
```

```
<spmlref:reference  typeOfReference="dxrGroupLink">
<spmlref:toPsoID ID="cn=Internal,cn=General,cn=Accounts and
Groups,cn=Windows Domain Europe,cn=TargetSystems,cn=My-Company"/>
         <spmlref:referenceData>
             <dsml:modification name="dxrStartDate"
operation="delete">
             </dsml:modification>
             <dsml:modification name="dxrEndDate" operation="add">
                 <dsml:value>29991231235959Z</dsml:value>
             </dsml:modification>
         </spmlref:referenceData>
</spmlref:reference>
</capabilityData>
</modification>
```

The above modification replaces the assignment to the group "Internal" in the target system "Windows Domain Europe". It deletes the previous start date and sets the end date to the year end 2999.

The modification of dxrGroupLink with the replace operation has a special semantic. It can be used either for update operation of an existing group assignment defined by the "dn" attribute or for creation of a new group assignment if supplied with common attributes, as shown in the following snippet:

```
<modification modificationMode="replace">
<capabilityData
capabilityURI="urn:oasis:names:tc:SPML:2:0:reference">
<spmlref:reference
xmlns:asgpar="urn:siemens:dxm:provisioning:asg:param:1:0"
        typeOfReference="dxrGroupLink">
<spmlref:toPsoID
ID="cn=Auditors,cn=Groups,cn=DirXmetaRole,cn=TargetSystems,cn=My-
Company"/>
        <spmlref:referenceData>
            <dsml:attr name="dxrEndDate">
                <dsml:value>29991231235959Z</dsml:value>
            </dsml:attr>
        </spmlref:referenceData>
</spmlref:reference>
</capabilityData>
</modification>
```

The replace operation for dxrGroupLink does not delete any existing group assignments.

To search for users with a specific permission assignment, use the <hasReference> clause within a search query:

```
<spmlsearch:query scope="subTree" targetID="users">
<spmlsearch:basePsoID ID="cn=Users,cn=My-Company"/>
<spmlsearch:and>
     <dsml:filter>
         <dsml:equalityMatch name="objectclass">
             <dsml:value>dxrUser</dsml:value>
         </dsml:equalityMatch>
     </dsml:filter>
     <spmlref:hasReference typeOfReference="dxrGroupLink">
         <spmlref:toPsoID ID="cn=Internal,cn=General,cn=Accounts and
Groups,cn=Windows Domain Europe,cn=TargetSystems,cn=My-Company"/>
     </spmlref:hasReference>
</spmlsearch:and>
 <dsml:attributes>
     <dsml:attribute name="cn"/>
 </dsml:attributes>
</spmlsearch:query>
```

The above filter searches for users who have the group "Internal" of the target system "Windows Domain Europe". Note that reference data such as start or end date are not supported.

For complete samples of how to manage role references, see especially the file "sampleRequestGroupLink.xml".

4.6.7. Suspend Capability

The suspend capability allows enabling and disabling a user.

The capability updates the attributes dxrState, dxrDisableStartDate and dxrDisableEndDate. But note that the state is also affected by other operations and attributes, namely by the dxrStartDate and dxrEndDate. A suspend request sets the user's state to DISABLED, a resume request to ENABLED. A user is considered active if his state is ENABLED.

In suspend or resume requests you can provide the attribute "effectiveDate". The effective date in a suspend request denotes the disable start date. As default, the service takes the day before. The effective date in a resume request denotes the disable end date. If it is not given, the service deletes both disable start and end date.

The following sample shows a suspend request with an effective date of January 1st, 1990:

```
<spmlsuspend:suspendRequest  effectiveDate="1990-01-
01T00:00:00+00:00">
<spmlsuspend:psoID ID="cn=John Doe 9876,o=testOrg,cn=Users,cn=My-
Company"  targetID="users"/>
</spmlsuspend:suspendRequest>
```

The service accepts a number of formats for the date representation; especially it supports the IETF standard.Internally it uses the *parse* method of the JDK class "java.util.Date".See there for more details on the formats.

In a search request you can use the <isActive> query clause to search for enabled users: Simply include the following element into your query filter: <spmlsuspend:isActive/>.

4.6.8. Password Capability

The password capability implementation only supports the requests “validatePassword” and “setPassword”.Reset and ExpirePassword are answered by an error response.

The password is not only stored in the attribute userPassword but also in the attribute dxmPassword.It is encrypted if the service has a certificate; otherwise, it is hashed.To read the certificate from the connectivity configuration tree, the service needs to authenticate as domain administrator.It derives the user name from its own bind credentials.Make sure that the service is bound as domain administrator!

4.7. Role Management

This section assumes that you are familiar with the SPMLv2 standard.It presents only aspects that are specific for DirX Identity.

Roles and role containers can be managed by SPMLv2 requests.The target ID “**roles**” identifies them.The service supports the following capabilities:

1. Search, namespaceURI = “urn:oasis:names:tc:SPML:2.0:search”.The SPMLv2 search capability applies both to roles and role containers.
2. Reference, namespaceURI = “urn:oasis:names:tc:SPML:2.0:reference”.The SPMLv2 reference capability only applies to roles.
3. Matchrule, namespaceURI = “urn:siemens:dxm:provisioning:role:matchrule:1:0”.The matchrule capability only applies to roles.The content adheres to the DirX Identity proprietary schema, which you find in the file “rpmatchrule.xsd”.See below for more details.

4.7.1. Identifiers

The identifiers in all requests are interpreted as the DNs (LDAP distinguished names) of the objects. An add request may, but need not contain an identifier.

If the add request provides an identifier (element <psoID>) the service tries to create an object with this DN. If the add instead contains a <containerID>, the service takes it as the DN of the parent node and creates an object beneath it. The RDN (relative DN) must either be passed as an attribute in the request or the object description has to calculate a default value. If neither a <psoID> nor a <containerID> are provided, the service creates the new object direct beneath the roles root (*cn=rolecatalogue,cn=<your domain>*).

4.7.2. Attributes

Mandatory attributes are the naming attribute *cn* and the object class. In add requests, roles and role containers can only be distinguished by the object class attribute. A value of “dxrRole” identifies a role, the value “dxrContainer” a role container.

Other attributes are optional and have to be among the list given in the listTargets response. Note that the operational links to junior roles and permissions and the role parameter related attributes are treated as capabilities and are silently ignored, when provided in simple attributes.

4.7.3. References

All role links to other objects within the Identity domain must be provided as SPMLv2 references. A role object supports the following references:

- dxrRoleLink: DN link to a junior role.
- dxrPermissionLink: DN link to a permission.
- dxrWorkflowLink
- dxrDelWorkflowLink
- dxrModWorkflowLink
- dxrPrivAssDelWorkflowLink
- dxrPrivAssWorkflowLink
- dxrReappWorkflowLink
- dxrSODWorkflowLink

For role parameter links, see the following match rule section.

Here is a sample snippet for a dxrPermission reference within an addRequest:

```
<capabilityData
capabilityURI="urn:oasis:names:tc:SPML:2.0:reference">
    <spmlref:reference typeOfReference="dxrPermissionLink">
        <spmlref:toPsoID ID="cn=Project Manager,cn=Project
Specific,cn=Corporate Permissions,cn=Permissions,cn=My-Company"
targetID="permissions"/>
    </spmlref:reference>
```

```
</capabilityData>
```

If these links are among the attribute list, they are silently ignored.

4.7.3.1. Multivalued References

To replace the values of a reference attribute with multiple new values, specify the new value references as children of the same capabilityData element. You may put values of different reference attributes below a common capabilityData element, as shown in the following example:

```
<spml:modification modificationMode="replace">
   <capabilityData
capabilityURI="urn:oasis:names:tc:SPML:2:0:reference">
       <spmlref:reference typeOfReference="dxrGroupLink">
           <spmlref:toPsoID  ID="cn=Users,…,cn=TargetSystems,cn=My-
Company"/>
       </spmlref:reference>
       <spmlref:reference typeOfReference="dxrGroupLink">
           <spmlref:toPsoID  ID="cn=Admins,…,cn=TargetSystems,cn=My-
Company"/>
       </spmlref:reference>
       <spmlref:reference typeOfReference="owner">
           <spmlref:toPsoID  ID="cn=Abele Marc,cn=Users,cn=My-
Company"/>
       </spmlref:reference>
       <spmlref:reference typeOfReference="owner">
           <spmlref:toPsoID  ID="cn=Dalmar
Christopher,cn=Users,cn=My-Company"/>
       </spmlref:reference>
   </capabilityData>
</spml:modification>
```

Or you may have one capabilityData element per attribute, as shown in the next example:

```
<spml:modification modificationMode="replace">
   <capabilityData
capabilityURI="urn:oasis:names:tc:SPML:2:0:reference">
       <spmlref:reference typeOfReference="dxrGroupLink">
           <spmlref:toPsoID  ID="cn=Users,…,cn=TargetSystems,cn=My-
Company"/>
```

```
        </spmlref:reference>
        <spmlref:reference typeOfReference="dxrGroupLink">
            <spmlref:toPsoID  ID="cn=Admins,…,cn=TargetSystems,cn=My-
Company"/>
        </spmlref:reference>
    </capabilityData>
    <capabilityData
capabilityURI="urn:oasis:names:tc:SPML:2:0:reference">
        <spmlref:reference typeOfReference="owner">
            <spmlref:toPsoID  ID="cn=Abele Marc,cn=Users,cn=My-
Company"/>
        </spmlref:reference>
        <spmlref:reference typeOfReference="owner">
            <spmlref:toPsoID  ID="cn=Dalmar
Christopher,cn=Users,cn=My-Company"/>
        </spmlref:reference>
    </capabilityData>
</spml:modification>
```

4.7.4. Role Parameter Links and Match Rules

If a role requires a role parameter, the DN of the role parameter and the corresponding match rule must be provided in the *matchrule* capability. Here is a sample for a match rule capability:

```
<capabilityData
capabilityURI="urn:siemens:dxm:provisioning:role:matchrule:1:0">
    <matchrule
xmlns="urn:siemens:dxm:provisioning:role:matchrule:1:0">
        <uid>uid-7f001-cee271-fed935aa6d--7eb4</uid>
        <roleparamDN>cn=Project,cn=My-
Company,cn=RoleParams,cn=Customer Extensions,cn=Configuration,cn=My-
Company</roleparamDN>
        <operator>=</operator>
        <type>Group</type>
        <attribute>dxrproject</attribute>
    </matchrule>
</capabilityData>
```

The <roleparamDN> and the <uid> denote the referenced role parameter object. Within a request one of them is sufficient. The <uid> refers to the dxrUid attribute of the role

parameter. It is automatically generated with the role parameter object and does not change when the parameter is moved. Currently, the <type> value has to be “Group” and the <attribute> refers to the group attribute, which in the role resolution process is matched according the <operator> with the DN of the selected role parameter value. For more details see the appropriate section in the manual.

The following sample gives the modification for a match rule within a modify request:

```
<spml:modification modificationMode="replace">
    <capabilityData
capabilityURI="urn:siemens:dxm:provisioning:role:matchrule:1:0">
        <matchrule
xmlns="urn:siemens:dxm:provisioning:role:matchrule:1:0">
            <uid>uid-7f001-cee271-fed935aa6d--7eb4</uid>
            <operator>=</operator>
            <type>Group</type>
            <attribute>dummy</attribute>
        </matchrule>
    </capabilityData>
</spml:modification>
```

The given value replaces all existing role parameter links and match rules in the role.

4.7.5. Delete

The DirX Identity service does not support a delete request with *recursive=”true”*.That is, the service deletes only the object with the given DN and rejects the request, if the entry has children.

4.8. Permission Management

Permissions of a DirX Identity domain can be managed by SPMLv2 requests using the target identifier "permissions".In addition to the mandatory operations add, modify, delete and lookup the provisioning service supports the following capabilities for permissions:

- Search, namespaceURI = “urn:oasis:names:tc:SPML:2.0:search”.The SPMLv2 search capability applies both to permissions and permission containers.
- Reference capability, namespace URI = "urn:oasis:names:tc:SPML:2:0:reference".The permission service supports the following simple reference types:
- dxrGroupLink: permission-to-group reference.
- dxrProject: reference to a project.
- dxrWorkflowLink
- dxrDelWorkflowLink

- dxrModWorkflowLink
- dxrPrivAssDelWorkflowLink
- dxrPrivAssWorkflowLink
- dxrReappWorkflowLink
- dxrSODWorkflowLink
- Permission match rule capability. It is identified by the namespace URI "urn:siemens:dxm:provisioning:permission:matchrule:1:0" and allows to manage the match rules of a permission.

DirX identity provides samples for requests and responses in the folder:

install_path/**provisioningServices/spmlv2/permissions**.

You must provide appropriate DirX Identity access policies for the requesting user.

4.8.1. Attributes

Mandatory attributes are the naming attribute "cn" and the object class. The object class attribute distinguishes permissions and containers in add requests. An object class value of “dxrPermission” identifies a permission. Otherwise the object is expected to become a container.

Other attributes are optional. Standard attributes are given in the listTargets response. If you extend the LDAP schema with your custom permission attributes, make sure you also extend the permission’s object description. Then you will be able to update and read them via the Provisioning Web Services. You don’t need to extend the list targets response.

4.8.2. Add, Modify and Delete

Add

Besides the normal attributes and simple references to other entities, the add request supports the capability for match rules. For a detailed explanation of the XML schema see the separate chapter on match rules.

Here’s the excerpt of a sample add request containing a reference capability to a group and the match rule capability:

```
    <spml:addRequest returnData="identifier" requestID="permAdd-01"
targetID="permissions">
  <spml:containerID ID="cn=wsTestPermissions,cn=Permissions,cn=My-
Company"/>
  <spml:data>
      <dsml:attr name="cn">
          <dsml:value>Permission1</dsml:value>
      </dsml:attr>
```

```
...
  </spml:data>
  <capabilityData mustUnderstand="true"
capabilityURI="urn:oasis:names:tc:SPML:2:0:reference">
      <spmlref:reference typeOfReference="dxrGroupLink">
          <spmlref:toPsoID ID="cn=Development Portal,cn=Group
Portals,cn=Accounts and Groups,cn=Intranet
Portal,cn=TargetSystems,cn=My-Company"/>
      </spmlref:reference>
  </capabilityData>

        <capabilityData mustUnderstand="true"
capabilityURI="urn:siemens:dxm:provisioning:permission:matchrule:1:0"
>
       <permMatchrules or="true"
xmlns="urn:siemens:dxm:provisioning:permission:matchrule:1:0">
          <permMatchrule>
              <userAttr>c</userAttr>
              <operator>=</operator>
              <groupConst>Group</groupConst>
              <groupAttr>c</groupAttr>
          </permMatchrule>
      </permMatchrules>
  </capabilityData>

    </spml:addRequest>
```

The request can only contain one capability for a match rule! This capability contains exactly 1 <permMatchrules> element, which in turn can contain 1 or more <permMatchrule> elements.

Modify

The modify operation can be applied both to permissions and to containers. In addition to normal attributes and simple references it can also modify match rules. As with the add request, the match rule capability contains at most 1 <permMatchrules> element.

A "replace" modification replaces the previous content with the one specified in the capability.

The same does the "add" modification, since a permission can contain at most 1 <permMatchrules>. Note that a <permMatchrules> element contains a list of simple <permMatchrules>, which are combined using "and" or "or" according the attribute "or" in <permMatchrules>.

The "delete" modification deletes the old match rule irrespective of the one that might be passed in the request. In other words, it does not check, if the passed match rule matches the existing one.

Here is a sample of a modify request, which replaces the attribute "description" and especially an existing match rule with a new one provided in the capability:

```
    <spml:modifyRequest requestID="permMod-01"
returnData="identifier">
  <spml:psoID
ID="cn=Permission1,cn=wsTestPermissions,cn=Permissions,cn=My-Company"
targetID="permissions"/>
  <spml:modification modificationMode="replace">
      <dsml:modification name="description" operation="replace">
          <dsml:value>description for Permission1
(modified)</dsml:value>
      </dsml:modification>
  </spml:modification>

        <spml:modification modificationMode="replace">
      <capabilityData mustUnderstand="true"
capabilityURI="urn:siemens:dxm:provisioning:permission:matchrule:1:0"
xmlns:mr="urn:siemens:dxm:provisioning:permission:matchrule:1:0">
          <mr:permMatchrules or="false">
              <mr:permMatchrule>
                  <mr:userAttr>c</mr:userAttr>
                  <mr:operator>=</mr:operator>
                  <mr:groupConst>Group</mr:groupConst>
                  <mr:groupAttr>c</mr:groupAttr>
              </mr:permMatchrule>
          </mr:permMatchrules>
      </capabilityData>
  </spml:modification>

    </spml:modifyRequest>
```

Delete

The delete operation can be applied both to permissions and to containers. If a container is not empty, you must supply the attribute "recursive" with a value of "true", or the request is rejected. If auditing is enabled for permissions, the permission entry is not immediately deleted. Instead, its state is set to DELETED and an audit trail is stored in its history attribute. Only when the history attribute is read and cleared will a subsequent consistency

rule physically delete the entry.

The following snippet gives an example of a delete request for a non-empty container:

```
<spml:deleteRequest requestID="permDelCont-03" recursive="true">
  <spml:psoID ID="cn=wsTestPermissions,cn=Permissions,cn=My-Company"
targetID="permissions"/>
</spml:deleteRequest>
```

4.8.3. Permission Match Rules

The content of the match rule capability is controlled by the XML schema with the namespace "urn:siemens:dxm:provisioning:permission:matchrule:1:0". You find the schema in the file "perm-matchrule.xsd" and embedded in the list target response for permissions "listTargetsRspPermissions.xml". This schema makes use of some common definitions for role parameters, which are defined in the file "rpmatchrule.xsd" and again in the list targets response for permissions.

A match rule is represented by the XML element <permMatchrules>. Here is a sample:

```
<permMatchrules or="false"
xmlns="urn:siemens:dxm:provisioning:permission:matchrule:1:0">
   <permMatchrule>
        <userAttr>dxrProject</userAttr>
        <operator>=</operator>
        <groupConst>Group</groupConst>
        <groupAttr>dxrProject</groupAttr>
    </permMatchrule>
    <permMatchrule>
        <userAttr>c</userAttr>
        <operator>contains</operator>
        <groupConst>Group</groupConst>
        <groupAttr>c</groupAttr>
    </permMatchrule>
</permMatchrules>
```

This sample contains a combined match rule that compares the user attributes "dxrProject" and "c" with the according group attributes.The project values must be the same, the country value of the user must contain that of the group.

A <permMatchrules> contains 1 or more single attribute comparisons, which are represented by elements <permMatchrule>.They are combined by "and" or "or" depending on the boolean XML attribute "or" in <permMatchrules>.

The element <userAttr> specifies the user attribute, the element <operator" the compare operation ("=", "contains", "!=", "<=", etc).

The element <groupConst> specifies whether the user attribute must be compared to a group attribute or a constant allowing the values "Group" or "Const".The element <groupAttr> contains either the group attribute or the constant value.

For handling of match rule modifications, see the section on "Add, Modify and Delete".

4.9. Business Objects Management

The term “business objects” refers to object types such as organizations or companies, context objects, location and alike.The Provisioning Web Services for their management have very much in common.Therefore, they are covered in a common section.

Identifiers

The identifiers in all requests are interpreted as the DNs (LDAP distinguished names) of the organizations or containers.

If the add request provides an identifier (element <psoID>), the service tries to create the business object or container with this DN.If the add instead contains a <containerID>, the service takes it as the DN of the parent container and creates an object beneath it with the value of the naming attribute as its RDN (relative DN).

Search

All business objects support the search capability with the namespaceURI “urn:oasis:names:tc:SPML:2.0:search”.

Attributes

Standard attributes are given in the appropriate listTargets response. But in order to simplify extension with further custom attributes, DirX Identity does not check them. Make sure that the attributes are defined in the respective object description of the domain.

References

The business objects references are all simple, that is without attributes. In most cases they refer to other business objects (context, category, location) or users (approver).

Samples

DirX Identity provides samples for requests and responses in the folder:

install_path/**provisioningServices/spmlv2/***type_of_business_object*.

For simplicity all the requests and responses in the samples are children of a <test> root element. This is not supported by the SPML standard.

4.9.1. Organization Management

Organizations or companies can be managed by SPMLv2 requests using the target identifier “organizations”.

Organizations support the following simple references: dxrApprover, dxrContextLink, dxrCategoryLink, dxrLocationLink, dxrOULink, dxrPrivilegeLink.

Attributes

Mandatory attributes are the naming attribute o and the object class. The object class attribute distinguishes organizations and containers in add requests. An object class value of “dxrOrganization” identifies an organization. Otherwise the object is expected to become a container.

4.9.2. Organizational Unit Management

Organization Units can be managed by SPMLv2 requests using the target identifier "organizationalUnit".

Organizational units support the following simple references: dxrApprover, dxrContextLink, dxrCategoryLink, dxrLocationLink, dxrCostUnitLink, dxrPrivilegeLink.

Attributes

Mandatory attributes are the naming attribute "ou" and the object class. The object class attribute distinguishes organizational units and containers in add requests. An object class value of “dxrOrganizationalUnit” identifies an organizational unit. Otherwise the object is expected to become a container. One special container may be an organization identified by object class "dxrOrganization" (with target identifier "organizations").

4.9.3. Context Management

Context objects can be managed by SPMLv2 requests using the target identifier “context”.

They support the following simple references: dxrApprover, dxrContextLink, dxrCategoryLink, dxrPrivilegeLink.

Attributes

Mandatory attributes are the naming attribute cn and the object class. The object class attribute distinguishes context objects and containers in add requests. An object class value of “dxrContext” identifies a context object. Otherwise the object is expected to become a container.

4.9.4. Location Management

Location objects can be managed by SPMLv2 requests using the target identifier “location”.

They support the following simple references: dxrApprover, dxrContextLink, dxrCategoryLink, dxrLocationLink.

Attributes

Mandatory attributes are the naming attribute l and the object class. The object class attribute distinguishes locations and containers in add requests. An object class value of “dxrLocation” identifies a location and “dxrCountry” a country container. Otherwise the object is expected to become a container.

4.9.5. Cost Unit Management

Cost Units can be managed by SPMLv2 requests using the target identifier “costUnit”.

Cost units support the following simple references: dxrApprover, dxrContextLink, dxrCategoryLink.

Attributes

Mandatory attributes are the naming attribute "cn" and the object class.The object class attribute distinguishes cost units and containers in add requests.An object class value of “dxrCostUnit” identifies a cost unit.Otherwise the object is expected to become a container.

4.10. Creating and Deleting Target Systems

SPMLv2 requests can create, modify, and delete target systems and target system containers.The corresponding requests need to set the target identifier “targetSystems”.In addition to the mandatory operations add, modify, delete, and lookup the provisioning service supports the following capabilities:

- Search, namespaceURI = “urn:oasis:names:tc:SPML:2.0:search”.The SPMLv2 search capability applies both to target systems and target system containers.
- Create Options: Allows defining if accounts and groups must be stored in the same container or in separate ones.With the child element <templateTS> the DN of the template Target System can be selected.
- Connection Options: This capability allows setting attributes that are used as connection parameters in realtime provisioning workflows.They typically contain the address of the target system, the DN of the bind account, and options and flags such as SSL and keystore.They are stored at the target system and only evaluated by special types of workflows (“cluster enabled workflows”).
- Environment Properties: This capability allows to set attributes that are used as environment properties in realtime provisioning workflows.They typically contain parameters such as account or group root in target system.They are stored at the target system and only evaluated by special types of workflows (“cluster enabled workflows”).
- Options: This capability allows setting the target system options that are typically used for automatic account and group creation.They are stored in tag/value format in the LDAP attribute dxrOptions.Attributes typically used in this context are "Account Root in TS" and "Group Root in TS".

The product delivers samples for requests and responses in the folder:

install_path/**provisioningServices/spmlv2/ts**

> You must provide appropriate DirX Identity access policies for the requesting user.

4.10.1. Identifiers

The identifiers in all requests are interpreted as the DNs (LDAP distinguished names) of the objects. An add request may, but need not contain an identifier.

If the add request provides an identifier (element <psoID>) the service tries to create the target system or container with this DN. If the add instead contains a <containerID> the service takes it as the DN of the parent node and creates an object beneath it. The RDN (relative DN) must either be passed as an attribute in the request or the object description has to calculate a default value. If neither a <psoID> nor a <containerID> is provided the service creates the new object direct beneath the target systems root (cn=*targetSystems*,cn=*your_domain*).

4.10.2. Attributes

Mandatory attributes are the naming attribute cn and the object class. The values of the object class or the dxrType attribute distinguish target systems, cluster, and (cluster-) containers in add requests. Specify the following values:

- For Object class:
- **dxrTargetSystem** - to indicate a target system
- **dxrContainer** - to indicate a target system container
- For the dxrType attribute:
- **dxrCluster** - to indicate a cluster
- **dxrClusterContainer** - to indicate a cluster container

Other attributes are optional and must be specified in the list provided in the listTargets response. By default the DirX Identity provides the file “listTargetsRspTSMgmt.xml”.

4.10.3. Connection Options

The namespace URI “urn:siemens:dxm:provisioning:ts:connectionOptions:1:0” identifies the capability for custom connection options. It is evaluated for target systems (not for containers) in add, modify, lookup and search operations.

The options are stored at the target system and are only evaluated by a special type of workflows: cluster-enabled realtime workflows. Note that the target system already supports default connection parameters as normal attributes: dxmAddress, dxmDataPort, dxmSecurePort (if SSL enabled), dxmSSL, dxmBindProfile-DN (link to bind account), dxmKeyStore, dxmKeyStoreAlias, dxmTrustStore, dxmTrustStoreAlias, dxmAuthenticationMode.

With the “connectionOptions” capability the client can pass an arbitrary list of additional

connection parameters. Before an account or group of this target system is provisioned, the workflow sets these and the default connection options in the connection configuration of the target system connector. Use the following default option names as far as possible: filename, driverDBType, url.

The parameters are passed as DSMLv2 <attr> elements within the capability. See the following XML snippet for a sample:

```
<capabilityData xmlns:dsml="urn:oasis:names:tc:DSML:2:0:core"
capabilityURI="urn:siemens:dxm:provisioning:ts:connectionOptions:1:0"
>

    <dsml:attr name="driverDBType">

<dsml:value>siemens.dxm.connector.jdbc.AccessOverJdbcOdbcDriver</dsml
:value>
    </dsml:attr>

    <dsml:attr name="url">
        <dsml:value>jdbc:odbc:personal</dsml:value>
    </dsml:attr>

    <dsml:attr name="myConnectionOption">
        <dsml:value>someValue</dsml:value>
    </dsml:attr>

</capabilityData>
```

4.10.4. Environment Properties

The namespace URI “urn:siemens:dxm:provisioning:ts:environmentProperties:1:0” identifies the capability for environment properties. It is evaluated for target systems (not for containers) in add, modify, lookup and search operations.

The properties are stored at the target system and are only evaluated by a special type of workflows: cluster-enabled realtime workflows.

With the “environmentProperties” capability the client can pass an arbitrary list of environment parameters. The provisioning workflow sets them before an account or group of this target system is provisioned.

The parameters are passed as DSMLv2 <attr> elements within the capability. See the following XML snippet for a sample:

```
<capabilityData mustUnderstand="true"
```

```
xmlns:dsml="urn:oasis:names:tc:DSML:2:0:core"
capabilityURI="urn:siemens:dxm:provisioning:ts:environmentProperties:
1:0">

    <dsml:attr name="accountrootints">
        <dsml:value>cn=Accounts</dsml:value>
    </dsml:attr>

    <dsml:attr name="grouprootints">
        <dsml:value>cn=Groups</dsml:value>
    </dsml:attr>

</capabilityData>
```

4.10.5. Options

The namespace URI “urn:siemens:dxm:provisioning:ts:options:1:0” identifies the capability for target system options. It is evaluated for target systems (not for containers) in add, modify, lookup and search operations.

Like the environment or connection options the attributes are passed as DSMLv2 <attr> elements.

4.10.6. Add

The add request supports the following capabilities: connection options, environment properties and create options.

The create options contain two optional child elements:

<accountsAndGroupsSeparate>: This flag specifies whether the accounts and groups are stored in the same or in separate folders. This decision cannot be changed later on in a modify request!

<templateTS>: This element contains the DN of the template target system. If it exists, it overrules the “type/cluster” approach to select the template system (see below).

The XML format is conveyed by the schema “urn:siemens:dxm:provisioning:TScreateOptions:1:0“, which is delivered in the file “tsCreateOptions.xsd”. Here a sample snippet for the capability:

```
<capabilityData
capabilityURI="urn:siemens:dxm:provisioning:TScreateOptions:1:0">
    <accountsAndGroupsSeparate
xmlns="urn:siemens:dxm:provisioning:TScreateOptions:1:0">false</accou
ntsAndGroupsSeparate>
```

```
<templateTS
xmlns="urn:siemens:dxm:provisioning:TScreateOptions:1:0">cn=LDAP,cn=T
argetSystems,cn=DirXmetaRole-SystemDomain</templateTS>
</capabilityData>
```

How to select the target system template?

The web service searches a template target system in the system domain (cn=TargetSystems,cn=DirXmetaRole-SystemDomain) in the following sequence:

If the create options capability contains an element <templateTS>, it uses this DN to read the target system.

Then it searches a target system matching the attributes “dxrType” and “dxrCluster” contained in the request.

If this search does not yield exactly one target system, it searches a target system matching the given “dxrType” alone.

Note that you can import appropriate target system sub-trees into the system domain for each cluster you need. This allows you to select the correct template simply by providing the “dxrCluster” and “dxrType” attributes.

By default, the target system configurations (which are the object descriptions, Java scripts and obligations) are stored directly beneath the target system entry. If you want to deploy many target systems into a common cluster, you should reduce the number of object descriptions by providing only one set of object descriptions that is shared between all the target systems of that cluster. In this case, you need to specify the destination folder for the configuration sub-tree in the normal attribute “dxmTSConfigurationLink”. Set it to the cluster folder to which the new target system belongs. If this attribute is set, the service stores the configuration folder beneath the designated entry. If the configuration folder already exists, the service does not change it.

Please note that the attribute “dxmTSconfigurationLink” is exclusively used by the Web Services when configuring target systems. It’s not used or evaluated by any other component (such as DirX Identity Manager, Services, and so on). You should not create the object descriptions, Java scripts, and so on in any other folder, because other components (such as DirX Identity Manager, Services, etc.) will not evaluate the attribute “dxmTSconfigurationLink”; these components expect the objects descriptions, Java scripts and so on to reside below the target system folder or below the cluster folder in parallel with the target systems of which the cluster consists, but nowhere else.

4.10.7. Modify

A modify request supports the capabilities “connectionOptions” and “environmentProperties”, but no others. This means especially that you cannot switch how to store accounts and groups: in the same or in separate folders.

4.10.8. Delete

The delete request allows deleting a target system or a target system container.

You must set the XML attribute recursive=”true” if you want to delete a non-empty target system container.You do not need to set the flag if you want to delete a target system.

The provisioning service supports a “tombstone” feature.It is realized as a user hook and activated in the default configuration.For more details see the section on user hooks.

4.10.9. Search

Search requests return containers or target system entries, never accounts, groups or configuration objects.

Pass the respective <includeDataForCapability> element if you want only part of the capabilities to be returned.Note that create or delete options are not part of the returned entry.

4.11. Accounts Management

Accounts in target systems can be managed by SPMLv2 requests using the target identifier “accounts”.In addition to the mandatory operations add, modify, delete and lookup, the SPML Provisioning Web Services support the following capabilities for accounts:

- Search, namespaceURI = “urn:oasis:names:tc:SPML:2.0:search”.The SPMLv2 search capability applies both to accounts and account containers.
- Reference capability.The accounts service supports the following reference types:
- dxrUserlink: a simple reference to the associated user.
- dxrUsedBy: simple references from functional accounts to users.
- dxrPwdPolicyLink: a simple reference to a password policy.
- dxrGroupMember: state-full references to groups of which the account is a member.
- Password capability.This capability allows setting and validating passwords.
- Suspend capability. This capability allows activating accounts.
- Async capability. The services support only the status request from this capability.

For the support of password reset scenarios, the services provide operations around challenge/response and for reading the password policy related to an account.

DirX identity provides samples for requests and responses in the folder:

install_path/**provisioningServices/spmlv2/accounts**.

You must provide appropriate DirX Identity access policies for the requesting user.

4.11.1. Attributes

Mandatory attributes are the naming attribute cn and the object class. The object class attribute distinguishes accounts and containers in add requests. An object class value of “dxrTargetSystemAccount” identifies an account. Otherwise the object is expected to become a container.

Other attributes are optional. Standard attributes are given in the listTargets response. Each target system will have its own set of proprietary attributes that DirX Identity does not check. Make sure that the attributes are defined in the respective object description of the domain.

Important standard attributes are dxrState and dxrTSState. The dxrTSState reflects the state of the account in the remote target system, i.e. as requested by the Web Service client.

The dxrState reflects the state of the account as requested by DirX Identity. Thus the resulting state may be different from the requested, since it considers also the state of the associated user and user-group assignments. Note that the account state is also controlled by the suspend capability.

4.11.2. Reference dxrUserlink

This reference is simply mapped to the account attribute “dxrUserlink” and contains the DN of the associated user.

4.11.3. Reference dxrUsedBy

This reference is simply mapped to the account attribute “dxrUsedBy”. It is intended for functional accounts and denotes the users that are allowed to work with this account.

4.11.4. Reference dxrPwdPolicyLink

This reference is simply mapped to the account attribute “dxrPwdPolicyLink”. It references a password policy and is only applicable for functional accounts, for which a separate password is managed different from that of the associated user.

4.11.5. Reference dxrGroupMember

This reference controls the group memberships of the account. The account-group memberships are associated with their state and stored in the attributes “dxrGroupMember*”. Valid states are: **OK**, **ADD**, **DELETED**, **IMPORTED**, **IGNORE**, **REMOTE**. (See the respective documentation on account states for details.)

The resulting member state might be different from the requested state. It is set by DirX Identity considering the previous state, the states of the account and the associated user, and the user-group assignments.

The XML format of the membership is conveyed by the schema file "accountGroupMembership.xsd" with the namespace

"urn:siemens:dxm:provisioning:account:groupmember:1:0" (see also the file listTargetsRspAccounts.xml). See the following request snippet for a sample:

```
<capabilityData
capabilityURI="urn:oasis:names:tc:SPML:2:0:reference">
<spmlref:reference xmlns="urn:oasis:names:tc:SPML:2:0:reference"
xmlns:ag="urn:siemens:dxm:provisioning:account:groupmember:1:0"
typeOfReference="dxrGroupMember">
    <toPsoID  ID="cn=Software
Tests,cn=Groups,cn=ts01,cn=testSystemsWebService,cn=TargetSystems,cn=
My-Company"/>
    <referenceData>
        <ag:accountGroupMembership state="IMPORTED"/>
    </referenceData>
</spmlref:reference>
</capabilityData>
```

A missing state is interpreted as "OK".

4.11.6. Add, Modify and Delete

Besides the normal list of attributes or modifications, add and modify requests may contain the reference capabilities.

To simplify deployment, attribute names are not cross-checked with the definitions made in the respective listTargets response. Hence - not in line with the OASIS SPMLv2 specification - you don’t have to change the listTargets response file, once you extend the LDAP schema and add new attributes or object classes.

For the management of completely new entry types not supported by the DirX Identity domain, see the section "Custom Objects".

You may want to define additional custom object descriptions that extend existing default object descriptions. A sample might be to distinguish between several types of users. In this case keep in mind that the Provisioning Service uses the default object description for creating a new object. For users, this is the "dxrUser" object description. This means that you can create your custom extended user entry, if you provide the appropriate object classes and attributes. Rules for calculating default attribute values are only taken from this default object description, not from your custom one.

When the Service processes a modify or delete, it obtains the appropriate object description the same way as Web Center or Identity Manager by applying the matching rules defined in the object description. This means especially, it associates the proper object description and its rules for attribute values.

4.11.7. Lookup and Search

If you want to see reference data in the lookup or search response you must pass the element <includeDataForCapability> for the reference capability.

The search request supports the suspend capability and its <isActive> query clause. Here a sample of a query element asking for active accounts being an imported member in the group “Firmware Tests”:

```
<spmlsearch:query scope="subTree" targetID="accounts">
 <spmlsearch:basePsoID
ID="cn=test,cn=Accounts,cn=ts01,cn=testSystemsWebService,cn=TargetSys
tems,cn=My-Company"/>
 <spmlsearch:and>
     <dsml:filter>
         <dsml:equalityMatch name="objectclass">
             <dsml:value>dxrTargetSystemAccount</dsml:value>
         </dsml:equalityMatch>
     </dsml:filter>
     <spmlsuspend:isActive/>
     <spmlref:hasReference typeOfReference="dxrGroupMember">
         <spmlref:toPsoID ID="cn=Firmware
Tests,cn=Groups,cn=ts01,cn=testSystemsWebService,cn=TargetSystems,cn=
My-Company" targetID="groups"/>
         <spmlref:referenceData
xmlns:ag="urn:siemens:dxm:provisioning:account:groupmember:1:0">
             <ag:accountGroupMembership state="IMPORTED"/>
         </spmlref:referenceData>
     </spmlref:hasReference>
 </spmlsearch:and>
 <dsml:attributes> … </dsml:attributes>
</spmlsearch:query>
```

4.11.8. Suspend Capability

The suspend capability allows enabling and disabling an account.

The capability controls the attribute dxrTSState and partly dxrState. It sets dxrState only if the account has no associated user. An account is considered active if its state is “ENABLED” or if it is IMPORTED with dxrTSState “ENABLED”.

In requests, the attribute effectiveDate is ignored. The state is always set immediately. The reason is that such a feature does not make sense for DirX Identity accounts: their state in Identity is mainly controlled by the associated user.

4.11.9. Password Capability

The password capability implementation only supports the requests "validatePassword" and "setPassword". Reset and ExpirePassword requests are answered with an error response.

For normal accounts, the password is not stored in the Identity domain. The Web Service sends an event to change the password of the account. The applicable workflow(s) perform the change at the connected system. Only if the account is privileged, the workflow stores the password also at the account in the Identity domain in the attributes userPassword and encrypted in the dxmPassword. For details, see the section "Password Capability" in "User Management".

For reset password scenarios, the Web Service supports identifying the account by using identifying attributes of the account itself, the target system and optionally the associated user. For details, see the section "Identifying Attributes" in "Common SPML Aspects". Especially in this case, normal basic authentication will not be possible. It can be replaced by either signing the request with the user's certificate or by challenge/response authentication.

The Web Service delivers the password policy associated with the identified account either in the response to an explicit getPasswordPolicy request or implicitly in the response for checking the challenge responses. For details, see the section "Authentication by Challenge/Response".

4.11.9.1. Authentication by User Certificate

The client application needs to read attributes from the certificate that identify the user and then insert them into the <identifyingAttributes> element of the request. Next, it needs to produce an XML signature of the SPML request and then put it in the request's SOAP header.

When configured properly, the Web Service verifies the signature and compares it to the identifying attributes. For details on how to configure the Web Service for this use case, see the section "XML Signature Verification Configuration".

To identify the account and optionally the associated user in a setPassword request, the service accepts identifying attributes instead of the normal DN identifier. These attributes must be sub-elements of the PSO ID, as shown in the following snippet:

```
<pass:psoID targetID="accounts">
    <idattr:identifyingAttributes>
         <dsml:attr name="ts.dxrTSDomainName">
              <dsml:value>WinEurope</dsml:value></dsml:attr>
         <dsml:attr name="dxrName">
              <dsml:value>agerbe66</dsml:value>
         </dsml:attr>
         <dsml:attr name="user.mail">
```

```
                <dsml:value>Alexander.Gerber@My-
Company.com</dsml:value>
          </dsml:attr>
     </idattr:identifyingAttributes>
</pass:psoID>
```

The element <identifyingAttributes> is part of the XML namespace “urn:com:dirxcloud:dxi:provisioning:identAttrs:1:0”. See the XML schema file **identifyingAttrs.xsd** for the exact definition.

The attribute names can contain prefixes followed by a dot (.):

- **ts** - the attribute identifies the target system.
- **user** - the attribute identifies the user associated to the account. If the account has no user (that is, attribute dxrUserLink is empty), the Web Service expects the attribute at the account.

An attribute name without a prefix is expected to identify the account itself.

When no DN is given but identifying attributes, the Web Service searches the account using the attributes identifying the account (that is, attributes without any prefix). It then checks whether the account belongs to the target system identified by the attributes with the prefix **ts.**. Finally, it verifies the given user attributes with prefix **user**: if it finds an associated user, it checks them with the user, otherwise with the account. The given identifying values only need to be contained in the corresponding user or account values; an exact match is not necessary

The attribute names provided in the request can be mapped to LDAP names. This method supports scenarios where the client does not know about the correct LDAP attributes. It configured in the Spring bean definition of the SetPassword handler for accounts in the file **accountsContext.xml** in the folder WEB-INF of the servlet deployment:

```
<bean id="SetPasswordAccountsHandler"
class="com.siemens.idm.service.provisioning.spmlv2.accounts.SetPasswo
rdHandler"
scope="prototype" parent="BasicAccountsHandler">

     <property name="attributeNameMapping">
        <map>
            <entry key="serialNr" value="uid"/>
            <entry key="domain" value="dxrTSDomainName"/>
            <entry key="logonname" value="dxrName"/>
        </map>
   </property>
```

```
</bean>
```

The map attributeNameMapping contains the attributes as sent by the clients as keys and the corresponding LDAP attribute names as values. Note that the keys must be lowercase and do not contain the prefix. As a result, if the client sends an identifying attribute **ts.domain**:

```
<dsml:attr name="ts.domain">
    <dsml:value>WinEurope</dsml:value></dsml:attr>
<dsml:attr name="dxrName">
```

The service will map it to ts.dxrTSDomainName.

4.11.9.2. Authentication by Challenge/Response

Without a certificate, users must authenticate by answering the challenges they have previously set up. If there is no certificate, the user must enter some data to identify the account – typically the Active Directory domain name and the account name. With these items wrapped in the psoID section, the client application requests the challenges for the associated user from the Web Service. See the section "Identifying Attributes" for details.

After the user has answered the challenges, the client can send them immediately to the Web Service for authentication, or the user can enter the new password and the client then sends this data together – password and challenge responses – to the Web Service. In either case, the setPassword request needs to contain the challenge responses for authentication even when they have been previously checked separately. In addition, all requests need to contain the attributes identifying the account. For details on these operations, see the section "Challenge/Response and Password Policy".

4.11.9.3. Authentication by One-Time Password

A user who has forgotten his password can request a one-time password via the sendOtp request. He can then authenticate to the Web Service with the one-time password in order to reset his password. See section One-Time Password for details.

4.11.10. Async Capability - Status Request

The SPML Provisioning Web Services implement only a subset of the Async capability defined in SPMLv2: the status request, but not the cancel request. The status request serves only one purpose in the context of password reset:

The setPassword request indirectly triggers a workflow to change the password in the related connected system. The Web Service indicates this action by sending the status "pending" in the response. This response always contains the attribute "requestID". The client can use this ID to ask for the status of the password change. It must send a status request and pass the ID in the attribute "asyncRequestID".

The Web Service checks an account attribute for obtaining the result of the setPassword

workflow. If the workflow is finished, it updates its state in the account. In this case, the Web Service returns the result (success or failure). Otherwise, it returns the state "pending".

4.11.11. Challenge/Response and Password Policy

If end users have forgotten their passwords, they can set new ones after authenticating themselves by answering their challenges. The Web Service provides the following operations to support this scenario:

- getChallenges
- checkChallengeResponses
- getPasswordPolicy

The next sections provide more detail about these operations.

4.11.11.1. getChallenges Operation

The getChallenges operation must contain a psoID with the identifying attributes for the end user's account. Typically, these items are the login name and the name of the Active Directory domain.

The response contains a subset of the user's challenges. The number of returned challenges is configured in the property **numberOfChallenges** of the **GetChallengesAccountsHandler** bean (see the file **accountsContext.xml**):

```
<bean id="GetChallengesAccountsHandler"
class="com.siemens.idm.service.provisioning.spmlv2.accounts.GetChalle
ngesHandler"
scope="prototype" parent="BasicAccountsHandler">
             <property name="attributeNameMapping"
ref="AttributeNameMapping"/>
             <property name="numberOfChallenges" value="3"/>
            <property name="includeChallengeKeys" value="false"/>
</bean>
```

The boolean property **includeChallengeKeys** defines whether the response includes the challenge keys in addition to the challenge questions. If requested, each challenge is returned as key/question pair where key and question are separated by a semicolon. Keys are assigned to challenges in the challenge proposal list. If no key is assigned to a challenge, the question is taken instead.

```
<getChallengesResponse requestID="chall-01" status="success">
   <challenge>Mandatory-PIN;My PIN</challenge>
   <challenge>My-Birthday;My birthday<challenge>
   <challenge>My favorite response;My favorite response</challenge>
```

```
<getChallengesResponse>
```

Keys starting with "Mandatory-" denote questions that must always be answered when authenticating via challenge/response. Note that questions for challenges with keys might vary with the requestor's language.

4.11.11.2. checkChallengeResponses Operation

The checkChallengeResponses operation contains the psoID and a list of challenges along with their responses:

```
<chall:checkChallengeResponsesRequest requestID="chall-01">
    <chall:psoID targetID="accounts">
         <idattr:identifyingAttributes>
              <dsml:attr
name="ts.domain"><dsml:value>WinEurope</dsml:value></dsml:attr>
             <dsml:attr
name="logonname"><dsml:value>agerbe66</dsml:value></dsml:attr>
         </idattr:identifyingAttributes>
    </chall:psoID>
    <chall:challengeResponse>
         <chall:challenge>My favorite film star</chall:challenge>
         <chall:response>George Clooney</chall:response>
    </chall:challengeResponse>
…
</chall:checkChallengeResponsesRequest>
```

The psoID is the same item as described in the section "getChallenges Operation".

If the Web Service successfully verifies the responses, it returns the associated password policy in the corresponding response:

```
<chall:checkChallengeResponsesResponse status="success"
requestID="chall-01">
    <passwordPolicy>
         <policy name="dxrPwdExpireWarning">0</policy>
…
    </passwordPolicy>
</chall:checkChallengeResponsesResponse>
```

The element <passwordPolicy> contains a list of <policy> elements. Each <policy> reflects one of the LDAP attributes in the password policy associated with the account. Name and

value of the policy are the name and value of the LDAP attribute correspondingly.

The algorithm for finding the associated password policy of an account operates as follows:

- If the account has a password policy link: take this one.
- Otherwise take the policy that is referenced by the target system.
- For (personal) accounts with a user: take the policy linked with the user.
- If no policy can be found this way: take the default for the domain.
- If no password policy exists in the domain: error.

The minimum number of responses is configured in the bean **CheckChallengeResponsesAccountsHandler**:

```
<bean id="CheckChallengeResponsesAccountsHandler"
class="com.siemens.idm.service.provisioning.spmlv2.accounts.CheckChal
lengeResponsesHandler"
scope="prototype" parent="BasicAccountsHandler">
         <property name="attributeNameMapping"
ref="AttributeNameMapping"/>
         <property name="minimumNumberOfResponses" value="3"/>
         <property name="includePasswordPolicyInResponse"
value="true"/>
</bean>
```

If you set the property **includePasswordPolicyInResponse** to **false**, the Web Service does not put the password policy into the response.

4.11.11.3. getPasswordPolicy Operation

The getPasswordPolicy operation allows a client to obtain the password policy associated with an account.This action is unnecessary in challenge/response scenarios because the policy is automatically contained in the checkChallengeResponses response.However, it can be helpful in scenarios where the client can authenticate itself via a certificate.In this case, the client needs to ask for the password policy explicitly.

The request needs the psoID with the attributes identifying the end user.

If the service successfully finds the account, it returns the list of attributes of the associated password policy.

4.12. One-Time Password

If an end user has forgotten his password, he can request a one-time password that is sent to his mobile phone via a Short Message Service (SMS) message.He can then authenticate against the Web Service with the one-time password in order to reset his password.

The Web Service provides the following operations to support this scenario:

- sendOTP
- setPassword

These operations support customized error messages.The next sections provide more detail about these operations and error messages.

4.12.1. sendOtp Operation

The sendOtp operation generates a one-time password and sends it to the requestor's mobile phone via an SMS message.

The sendOtp request must contain a psoID with the identifying attributes for the end user's account. Typically, these items are the login name and the name of the Active Directory domain.

```
<chall:sendOtpRequest requestID="otp-01">
   <chall:psoID targetID="accounts">
        <idattr:identifyingAttributes>
            <dsml:attr name="ts.domain">
                <dsml:value>WinEurope</dsml:value>
            </dsml:attr>
            <dsml:attr name="logonname">
                <dsml:value>agerbe66</dsml:value>
            </dsml:attr>
        </idattr:identifyingAttributes>
   </chall:psoID>
</chall:sendOtpRequest>
```

The status attribute in the sendOtp response indicates whether the operation succeeded or failed. On success, the response includes:

- The time until which the one-time password is valid.
- The destination type.
- The mobile phone number, partially scrambled.
- The user's password policy.

> The one-time password is not part of the response.

```
<chall:sendOtpResponse status="success" requestID="otp-01">
   <otpTtlEnd>20151008122615Z</otpTtlEnd>
   <otpDestinationType>mobile</otpDestinationType>
```

```
   <otpDestination>0198 56XXXX</otpDestination>
   <passwordPolicy>
        <policy name="dxrPwdExpireWarning">0</policy>
        …
   </passwordPolicy>
</chall:sendOtpResponse>
```

The format and validity period of the generated one-time password, the SMS message format and the name of the account or user attribute with the mobile phone number are configured in properties of the SendOtpAccountsHandler bean (see the file **accountsContext.xml**):

```
<bean id="SendOtpAccountsHandler"
class="com.siemens.idm.service.provisioning.spmlv2.accounts.SendOtpAc
countsHandler"
scope="prototype" parent="BasicAccountsHandler">
        <property name="attributeNameMapping"
ref="AttributeNameMapping"/>
        <property name="otpTimeToLive" value="15"/>
        <property name="otpDestinationAttributeName" value="mobile"/>
        <property name="otpDestinationType" value="mobile"/>
        <property name="otpPasswordLength" value="8"/>
        <property name="otpPasswordPolicy" value="0123456789"/>
        <property name="checkUserIdentifiers" value="false"/>
        <property name="otpMessageBody" value="#{'#{Common
Text.OTPSMSText}'}"/>
</bean>
```

4.12.2. setPassword Operation

The setPassword request must contain the psoID for the end user's account as in the sendOtp request. For authentication, the request must include the one-time password the user received via SMS. And finally, the request includes the new account password.

```
<pass:setPasswordRequest requestID="set-otp-01">
   <pass:psoID targetID="accounts">
        <idattr:identifyingAttributes>
            <dsml:attr name="ts.domain">
                <dsml:value>WinEurope</dsml:value>
            </dsml:attr>
            <dsml:attr name="logonname">
```

```
                <dsml:value>agerbe66</dsml:value>
            </dsml:attr>
        </idattr:identifyingAttributes>
    </pass:psoID>
    <chall:otpPassword><password></chall:otpPassword>
    <pass:password>dirX!123</pass:password>
</pass:setPasswordRequest>
```

The status attribute in the setPassword response indicates whether the operation succeeded or failed. On error, the error attribute and the errorMessage element give more details.

```
<pass:setPasswordResponse status="success" requestID="set-otp-01"/>
```

4.12.3. Custom Error Messages

On failure, the sendOtp and setPassword request may return custom errors to indicate the cause of the failure; for example:

```
<pass:setPasswordResponse status="failure" error="customError"
requestID="set-otp-01">
   <errorMessage>otpMismatch</errorMessage>
</pass:setPasswordResponse>
```

The list of custom error messages includes:

otpGenerationFailed - generating the one-time password failed.

otpNoAddress - neither the account nor the user have a mobile phone number.

otpSendFailed - sending the one-time password via SMS to the end user failed.

otpMismatch - the one-time password presented in a setPassword request doesn't match.

otpTtlReached - the one-time presented in a setPassword request has expired.

otpLocked -the account is locked from authenticating via one-time password.

pwdPolicyMismatch - the new password presented in a setPassword request doesn't comply with the password policies.

4.13. Groups Management

Groups in target systems can be managed by SPMLv2 requests using the target identifier

“groups”.In addition to the mandatory operations add, modify, delete, and lookup, the provisioning service supports the following capabilities for groups:

- Search, namespaceURI = “urn:oasis:names:tc:SPML:2.0:search”.The SPMLv2 search capability applies both to groups and group containers.
- Reference capability.The groups service supports the following reference types:
  - dxrFunctionalAccount: simple references to privileged accounts.
  - dxrRoleParamLink: simple references to role parameters.
  - dxrObligationLink: simple references to obligations.

DirX Identity provides samples for requests and responses in the folder:

install_path/**provisioningServices/spmlv2/groups**.

You must provide appropriate DirX Identity access policies for the requesting user.

4.13.1. Identifiers

The identifiers in all requests are interpreted as the DNs (LDAP distinguished names) of the groups or containers.

If the add request provides an identifier (element <psoID>) the service tries to create the group or container with this DN. If the add instead contains a <containerID>, the service takes it as the DN of the parent container and creates an object beneath it. The RDN (relative DN) must either be passed as an attribute in the request or the object description has to calculate a default value.

4.13.2. Attributes

Mandatory attributes are the naming attribute cn and the object class. The object class attribute distinguishes groups and containers in add requests. An object class value of “dxrTargetSystemGroup” identifies a group. Otherwise the object is expected to become a container.

Other attributes are optional. Standard attributes are given in the listTargets response. Each target system will have its own set of proprietary attributes that DirX Identity does not check. Make sure that the attributes are defined in the respective object description of the domain.

4.13.3. References

Account-group memberships are only managed via the accounts service!

The reference values are simply mapped to the respective attributes with the same name.

The following references are supported:

dxrFunctionalAccount

dxrRoleParamLink

dxrObligationLink

dxrProject

dxrWorkflowLink

dxrDelWorkflowLink

dxrModWorkflowLink

dxrPrivAssDelWorkflowLink

dxrPrivAssWorkflowLink

dxrReappWorkflowLink

dxrSODWorkflowLink

The capability "urn:siemens:dxm:provisioning:group:RPValues:1:0" supports the management of role parameter values, which are modelled as specific properties.

4.13.4. Add, Modify and Delete

Besides the normal list of attributes or modifications add and modify requests may contain the reference capabilities.

The delete request sets the attribute dxrTSState to **DELETED**.It only deletes the group physically if its attribute dxrState has the value **DELETED**.

4.13.5. Lookup and Search

If you want to see reference data in the lookup or search response you must pass the element <includeDataForCapability> for the reference capability.

4.14. Custom Objects

Sometimes DirX Identity customers not only need to extend standard DirX Identity domain objects like the user with custom attributes but to work with completely new objects.This task can be accomplished by creating new object descriptions.To manage these objects with the SPML Provisioning Web Services, they need to extend the Web Services configuration with custom target identifiers.

The Web Services provide generic classes to manage custom object types.These classes allow you to create, modify, delete, lookup and search such objects.In addition, standard capabilities - especially the reference capability - are supported out of the box.They must be enabled and configured analogous to the standard object types such as the user.

The following files must be adapted:

applicationContext.xml:

This Spring bean configuration file contains the dispatcher configurations for the SPML operations: add, modify, etc.For each custom object the appropriate handler must be added to the dispatcher.

customContext.xml:

This Spring bean configuration file is imported into the applicationContext.xml and should contain the custom bean configurations. They should be kept separate from the product configuration files in order to ease version upgrades.

listTargetsRspCustom.xml:

For each entity, type one listTargets response file must be supplied and referenced from the applicationContext. It describes the schema and capabilities of entities associated with a target identifier.

The "Custom Objects Operations" section describes how the custom handler processes the SPMLv2 requests. For sample requests and responses, see the folder *install_path* /**provisioningServices/spmlv2/custom**. You'll also find templates for the configuration files in this location.

4.14.1. Custom Objects Custom Context

The custom bean configurations for one entity type should be configured in a separate Spring bean configuration file: one file per entity type. This file is to be imported into **applicationContext.xml**. These configuration items should be kept separate from the standard product configuration files in order to minimize migration effort for version upgrades.

Copy the file **customContext.xml** delivered with the product in the **spmlv2/custom** folder to the WEB-INF folder of your Web Services application. Optionally rename it to *myEntity**Context.xml*. In this file, define your custom handlers and adjust their properties. If you need to support more than one custom entity type, you must rename the sample bean names (not the Java class names).

You need to configure a handler for each operation and each capability supported by your custom entity types. The following section explains your changes for the handlers needed for one custom entity type.

4.14.1.1. CustomReferenceHandler Bean

Enter the references you want to support. Let's assume you have only the contextLink as a reference. You need to enter this as an element of the refType2HandlerMap property. This property is a map with the reference type as the key and the reference handler bean as its value.

Standard reference handlers are already configured in the application context and you can simply use them here. Only if you have a new attribute (that is, a reference type) you need to define the handler bean analogous to the existing ones.

Here is the sample:

```
<property name="refType2HandlerMap">
        <map>
            <!--  key: reference type,  value: IReferenceHandler -->
```

```
            <entry key="dxrContextLink"><ref
bean="ContextLink"/></entry>
        </map>
    </property>
```

4.14.1.2. Object Type Meta Data Beans

You need to define the meta data of your custom entity type and of the associated containers. For each type, you need to configure the following meta data: object class, entity type, object description name, naming attribute. You do this by configuring one bean of class ObjectTypeMetaData. Here is an example for dxrContext:

```
<bean id="objectTypeContext"
class="com.siemens.idm.service.provisioning.spmlv2.custom.ObjectTypeM
etaData">
        <property name="objectClass" value="dxrContext"/>
        <property name="entityType" value="dxrContext"/>
        <property name="odName" value="dxrContext"/>
        <property name="namingAttribute" value="cn"/>
</bean>
```

Make sure the object description with the given name exists and that it is associated with your custom entity when it is read from LDAP (the <mapping> section in the object description).

4.14.1.3. BasicCustomHandler Bean

The basic handler bean defines a bean of class BasicCustomHandler and contains maps with references to the before configured reference handlers and meta data. The object description name for your custom entity type (not for any containers) must be configured in an extra property:

```
<property name="objectDescriptionName" value="dxrContext"/>
```

The map objectTypeMap contains the references to object type meta data beans with the object description name as the key:

```
<property name="objectTypeMap">
     <map>
          <!--  key: OD name;  value: ObjectTypeMetaData -->
           <entry key="dxrContext" value-ref="objectTypeContext"/>
           <entry key="dxrContainer" value-
```

```
ref="objectTypeContainer"/>
           <entry key="dxrCustomContainer" value-
ref="objectTypeCustomContainer"/>
      </map>
</property>
```

The map capability2HandlerMap contains the capability handlers with the entity type as the key. Typically you will only have the reference handler as in the following sample:

```
<property name="capability2HandlerMap">
      <map>
           <!--  key: capability URI,  value: handler class -->
                   <entry
key="urn:oasis:names:tc:SPML:2:0:reference">
                <ref bean="CustomReferenceHandler"/>
           </entry>
      </map>
</property>
```

4.14.1.4. Operation Handler Beans

For each of the operations you want to support, you must configure the appropriate bean. As the operations add, modify, delete and lookup are mandatory and you most probably want to support a search, you need to define at a minimum the following beans:

```
<bean id="AddMyEntity"
class="com.siemens.idm.service.provisioning.spmlv2.custom.AddCustom"
scope="prototype" parent="BasicCustomHandler"></bean>

<bean id="DeleteMyEntity"
class="com.siemens.idm.service.provisioning.spmlv2.custom.DeleteCusto
m" scope="prototype" parent="BasicCustomHandler"></bean>

<bean id="LookupMyEntity"
class="com.siemens.idm.service.provisioning.spmlv2.custom.LookupCusto
m" scope="prototype" parent="BasicCustomHandler"></bean>

<bean id="ModifyMyEntity"
class="com.siemens.idm.service.provisioning.spmlv2.custom.ModifyCusto
m" scope="prototype" parent="BasicCustomHandler"></bean>
```

```
<bean id="SearchMyEntity"
class="com.siemens.idm.service.provisioning.spmlv2.custom.SearchCusto
m" scope="prototype" parent="BasicCustomHandler"></bean>
```

If you have only one custom entity, you can leave the bean names "*Custom" of the sample files. Do not change the Java class names and make sure the parent reference matches your basic handler bean name!

4.14.2. Custom Objects ListTargets

You only need to provide a listTargets response file to comply with the SPMLv2 specification. It requires the implementation of a listTargets request, which returns a definition of all supported target identifiers. The Web Service creates the listTargets response by composing the content of all configured listTargets response files into one single response document.

The custom handler does not evaluate the entity and schema definitions so you don't need to ensure the correct attribute names and entity schema definitions in the response file.

For each entity type, one listTargets response file must be supplied and referenced from the applicationContext. It describes the schema and capabilities of entities associated with a target identifier.

Copy the file listTargetsRspCustom.xml and optionally rename it according your entity type; for example, listTargetsRspMyEntity.xml.

In the element <**target**> replace the value of the attribute 'targetID' with your entity type, for example:

```
<spml:target ... targetID="myEntity">
```

As the schema information is not evaluated by the Web Service, you can leave the attribute and object class definitions as they are. Just for clarity, change the **objectClassDefinition** name from "dxrContext" to a name that reflects your entity; for example, 'myEntity'.

```
<spmldsml:objectClassDefinition name="myEntity">
```

Next, adapt the **supportedSchemaEntity** accordingly:

```
<spml:supportedSchemaEntity entityName="myEntity"
isContainer="false"/>
```

Make sure that the **capabilities** are correct: Typically references to other objects are modeled as reference capabilities. The sample in listTargetsRspCustom.xml has 2

references: dxrContextLink and dxrCategoryLink. Adapt these parts as you need. Typically the reference names equal the LDAP attribute names. The attribute 'entityName' in <appliesTo> and <schemaEntity> has to refer to your entity, e.g. entityName="myEntity".

4.14.3. Application Context

The SPMLv2 Web Services are configured using Spring beans. The root configuration file needs to be the **applicationContext.xml** file in the WEB-INF folder of the deployed Web Service. This file mainly contains the configuration of the dispatcher beans for the SPML operations (add, modify, and so on) and imports entity type-specific configurations from corresponding files.

To enable the management of custom entity types, the handlers for this entity type must be added to the central dispatchers. This task must be performed in the one-and-only **applicationContext.xml**. For a template on how to do this, see the file **customApplicationContext.xml** delivered in the **spmlv2/custom** folder of the product installation.

This Spring bean configuration file contains the dispatcher configurations for the SPML operations: add, modify, and so on. The appropriate handler for each custom object must be added to the dispatcher.

Make sure to import your handler bean configurations from your custom context file (see the section “Custom Objects Custom Context” for a description of how to generate this):

```
<import resource="myEntityContext.xml"/>
```

There is one dispatcher for each operation. Mandatory operations are: add, modify, delete, lookup and most often you will want to search. In these cases, you need to extend the handler map of the following dispatchers.

4.14.3.1. ListTargetsHandler

Enter your basic handler bean and your listTargets response file into the respective maps:

```
<property name="target2HandlerMap">
    <map>
        <!--  key: target id,  value: ITargetHandler bean ref -->
        <entry key=...</entry>
        <entry key="myEntity"><ref
bean="BasicMyEntityHandler"/></entry>
    </map>
</property>
 <property name="target2ResponseMap">
    <map>
        <!--  key: target id,  value: name of file with list targets
```

```
response -->
        <entry key=.../>
        <entry key="myEntity" value="WEB-
INF/listTargetsRspMyEntity.xml"/>
    </map>
</property>
```

4.14.3.2. AddDispatcher

```
<property name="handlerMap">
    <map>
        <!--  key: target id,  value: handler bean name -->
        <entry key=.../>
        <entry key="myEntity" value="AddMyEntity"/>
    </map>
</property>
```

Extend the other dispatchers in the same way: ModifyDispatcher, DeleteDispatcher, LookupDispatcher, SearchDispatcher.

4.14.4. Custom Objects Operations

This chapter gives information on how the operations (add, modify, etc) are implemented in the generic handlers.

4.14.4.1. Add

The AddCustom handler class supports creation of object types including containers. It expects it is responsible for one object type (such as a context object) and a number of container objects that can contain that object type.

The addRequest needs to contain an attribute "objectClass" whose values allow for determining the appropriate object description. The algorithm is based on the configuration of the Object Meta Data beans. See the chapter on custom context for more details. The handler uses the values of the objectClass as a key to get an object Meta Data object. This object contains the name of the corresponding object description.

The addRequest also needs to contain either the DN of the new object or its parent container.

If an approval workflow has been started rather than creating the object in the LDAP domain, the handler returns the state "PENDING".

4.14.4.2. Modify

The modifyCustom handler performs the requested modifications using the Service class configured in the object description.

4.14.4.3. Delete

In case the object to be deleted is a container, the deleteCustom handler first deletes its child objects, if the recursive flag has been set in the request. The object is considered a container, if its object description name is not the one of the entity as configured in the property "objectDescriptionName" of the BasicCustomHandler. See the configuration of the custom context for more details on this.

4.14.4.4. Lookup and Search

The lookup- and searchCustom handlers use the normal read and search functionality to retrieve the matching objects.Search is done using LDAP paging in the same way as for the other entity types.

4.14.4.5. Attributes

The handler processes the attributes transparently.In particular, it does not evaluate the listTargets response configured in the appropriate file (see the section on listTargets response).You only need to make sure that the properties are configured in the corresponding object descriptions.

4.14.4.6. Capabilities

The custom handler supports simple references; that is, DN links from the entity to other ones without any reference attributes.Of course, they can also be processed as normal attributes rather than SPMLv2 references.

4.15. User Hooks

The SPML Provisioning Web Services support custom code before and after a request is processed.Note that this feature is only available for SPMLv2 based requests.

4.15.1. User Hook Interface

When receiving a SPMLv2 request the request processor checks if a user hook is configured.If a user hook is configured the processor performs the user hook before and after processing the request.

A user hook must implement the interface “ISpmlv2Userhook” and it can optionally also implement “ISpmlv2UserhookSetContext”.For a Java documentation see the folder **Documentation/DirXIdentity/ProvisioningWebServices** of your installation media.

The interface “**ISpmlv2Userhook**” defines two methods:

beforeRequest(...)

DirX Identity performs this method before it processes the request itself. It passes the request and the authenticated LDAP connection that the service uses itself. The user hook may perform any operation including reading and writing in the Identity domain. It may change the request and return it back to the request processor. In case of errors it should throw an exception. In this case DirX Identity returns an error response to the client.

afterRequest(...)

DirX Identity performs this method after it has processed the request itself. The user hook receives the request, the response produced by DirX Identity, and the LDAP connection. This allows the user hook to perform any operations after the request was processed. It may even react on failed requests.

The method must return a SPMLv2 response that is then returned to the client.

The interface "**ISpmlv2UserhookSetContext**" defines one method:

setContext (...)

With this method DirX Identity passes a request context. This context implements the interface "ISpmlv2UserhookContext". It especially provides a method to read the DN of the authenticated user.

DirX Identity returns the response from the user hook to the web service client.

For a sample implementation see the folder **Additions/ProvisioningWebServices/samples** on your installation media. It contains the class "TombstoneUserhook". This class considers only delete requests for target systems. In its method "beforeRequest" it copies the target system subtree to the subtree denoted by the configured root DN.

4.15.2. User Hook Configuration

User hooks are configured using Spring dependency injection. In the file

servlet-home/**WEB-INF/applicationContext.xml**

a bean "DispatcherWithUserhooks" is defined with a map of references to user hook beans. The target identifiers are used as keys for the user hooks. Thus, you can define a different user hook for each target identifier.

The following sample shows how to configure the user hook "TombstoneUserhook" for the target identifier "targetSystems":

```
<bean id="TombstoneUserhook"
class="com.siemens.idm.service.provisioning.spmlv2.ts.TombstoneUserho
ok" scope="prototype">
  <property name="tombstoneDN" value="l=tombstone,cn=My-Company"/>
</bean>
```

```
<bean id="DispatcherWithUserhooks"
class="com.siemens.idm.service.provisioning.spmlv2.BasicDispatcher"
scope="prototype">
 <property name="userhookMap">
     <map>
         <!--  key: target id,  value: ISpmlv2Userhook implementation
bean ref -->
         <entry key="targetSystems">
             <ref bean="TombstoneUserhook"/>
         </entry>
     </map>
 </property>
</bean>
```

The relevant key for user hooks in Iterate operations is calculated from the initial searchRequest related to the Iterate operation.

Configuring the user hook as a Spring bean allows a very flexible configuration.In the sample above, the user hook class has got a public field named “tombstoneDN” that is set with the value “l=tombstone,cn=My-Company”.

4.16. Using the Sample Client

The library **com.siemens.dxm.provisioning.jar** contains a Java-based SOAP client and the classes that represent all the requests and responses.You can use this client as the basis for a quick client implementation.It is configured the same way as the other agents based on the Java Connector Integration Framework.

This section describes how to:

- Get started with the sample client
- Test the sample client’s functionality

An additional convenience class is provided as a sample and simplifies instantiating and configuring the client.See the section "Using the Sample SOAP Connector" for more details.

4.16.1. Getting Started with the Test Client

A test client is available in *install_path***/provisioningServices/spmlv2**. It is based on the DirX Identity Connector Integration Framework and can be used for initiating sample requests and obtaining sample responses. Perform these steps to get started:

1. Use a command prompt or shell and navigate into *install_path***/provisioningServices/spmlv2**.
2. If you intend to use client-SSL for the connection between the Web Services and the Java-based Server, perform the steps described in the *DirX Identity Connectivity*

Administration Guide, chapter "Server SSL Connections" for Java-based Server, section "HTTP/SOAP Connection to DirX Identity SOAP Clients".

3. Verify and, if necessary, customize the file **conf.xml** with respect to the parameter **url**, the authentication parameters, and the SSL parameters of the SOAP connector configuration section. We recommend creating a backup copy before you start to customize. The following cases must be considered:

   a. If **Non-SSL-connections** to the Web Services have been deployed into the Java-based Server. The file **conf.xml** as shipped with the product is suitable for the demo domain only and for Tomcat/IdS-J port 40000 and non-SSL connections to the Java-based Server. Here is sample of the related connector configuration:

   ```
   <connection type="Soap"
   url="http://localhost:40000/ProvisioningService-My-
   Company/services/Spmlv2RequestService" />
   ```

   Customize the **url** parameter of the connector configuration so that it is correct regarding host, port and the technical domain name in the URL.

   b. If **Non-SSL-connections** to Web Services have been deployed into a native Tomcat. In this case, the port 40000 is inappropriate and must be replaced by the Tomcat port.

   c. For **SSL connections**, see the subsection "Securing Connections between Web Services Clients and Web Services with SSL" of the section "Establishing Secure Connections with SSL" in the *DirX Identity Connectivity Administration Guide* and ensure that the **url** parameter of above connector configuration starts with **https:**.

   d. If the use of proprietary authentication based on asymmetric key cryptography is required, a key material must be generated and correctly deployed. Use the sample configuration file *install_path*/provisioningServices/spmlv2/conf-proprietary-auth.xml* as a sample for your configuration file. See the section "Proprietary Authentication" for more details.

4. Check the file name for the file reader in the configuration **conf.xml**. The file reader is configured in the <connector> element with name "in". Ensure that the contents of the "filename" attribute is "request.xml". This is typically true for the shipped with the project.

5. Launch the **fireSpmlv2-Request.bat** sample (or the **./fireSpmlv2-Request.bat sample** for UNIX) with an additional parameter that identifies the operation to be tested. The possible values for the sample are case-sensitive and described below. The operations to be executed are specified in **sample-request.xml.** The related response file on execution is **sample-responseOut.xml**. Response and trace of the last launch of a sample are **responseOut.xml** and **trace.txt**, respectively. For example, to run the sample for creating a role container for Windows platforms, enter the command **fireSpmlv2-Request.bat addRoleCont**.

To get started, we recommend running the test client with the samples in following order:

- Value **listTargets** - specifies a **listTargets** operation, using the request **listTargets-**

request.xml and creating a response file **listTargets-responseOut.xml**.

- Value **addRoleCont** - specifies adding a role container, using the request **addRoleCont-request.xml** and creating a response file **addRoleCont-responseOut.xml**.
- Value **addRole** - specifies adding a role, using the request **addRole-request.xml** and creates a response file **addRole-responseOut.xml**.
- Value **lookupRoleCont** - specifies lookup for a role container.
- Value **lookupRole** - specifies lookup for a role.
- Value **searchRole** - specifies searching for roles and role containers. In addition to the found entries, an **iterator** identifier is returned in the response. For the very first search request after restart of Tomcat, its value is **1**. The current identifier can be found by searching for the pattern **iterator** in the response file.
- Value **iterate** - specifies getting the next page of the search result associated with the **iterator** identifier **1**. Note that this identifier is suitable only for the very first search request after restarting the Tomcat. In all other cases, the identifier in the request file must be changed according to the search result for paging.
- Value **closeIterator** - specifies client-side termination of paging through the search result associated with **iterator** identifier **1**. Note that this identifier is suitable only for the very first search request after restarting the Tomcat. In all other cases, the identifier in the request file must be changed according to the search result for paging.
- Value **delRole** - specifies deletion of the role created by running the **addRole** sample.
- Value **delRoleCont** - specifies deletion of the role container created by running the **addRoleCont** sample.

4.16.2. Testing the Sample Client Functionality

Before you can test the sample client functionality, make sure you have performed the steps described the section "Getting Started with the Test Client", including at a minimum the successful execution of the operations **listTargets**, adding role container and deleting role container.

Perform these steps to test other functions using the related samples; for example, the samples delivered for users:

- Use a command prompt or shell and navigate into *install_path***/provisioningServices/spmlv2**.
- Create a backup copy of the file **conf.xml**.
- Change the file name for the file reader in the configuration conf.xml. The file reader is configured in the <connector> element with name **in**. Exchange the contents of the **filename** attribute with another one; for example, **users/sampleRequests.xml**.
- Remove the files **responseOut.xml** and **trace.txt** if they still exist from a previous test client launch.
- Launch the command **fireSpmlv2-Request.bat** (or **./fireSpmlv2-Request.bat** for UNIX).
- View the related response **responseOut.xml** and **trace.txt**.

- Finally, restore the file **conf.xml** from your backup.

4.16.3. Generating a Client from the WSDL

Typically you obtain a client for the Web Service by using some tool, which generates the Java (or .Net or ...) classes from the wsdl and XML schema files. In order to do this you should be aware of the wsdl component model shown in the next diagram:

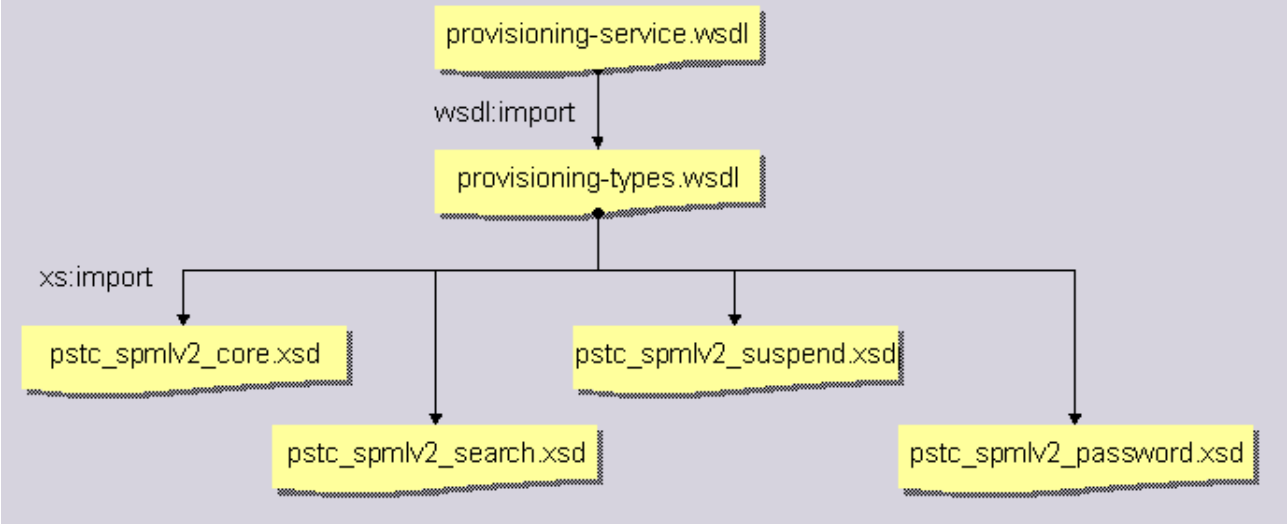

Figure 8. WSDL Component Model

The subfolder **WEB-INF/etc** of a deployment instance contains a file **provisioning-service.wsdl** which in turn contains the wsdl description for the Web Service. It imports the services types in the file **provisioning-types.wsdl**, which in turn imports the necessary SPML schemata. But for implementing a client you need to generate classes from one or more other XML schemata:

- **pstc_spmlv2_reference.xsd**: SPMLv2 definitions for reference data. They are used for practically all object types.
- **DSMLv2_SPMLv1.xsd**: Since the Provisioning Web Services support the SPMLv2-DSMLv2 profile, the client needs the DSMLv2 schema plus some additions required and defined later on for SPMLv2. These are contained in the given file, but could also be obtained via the official OASIS links.

For selected features, the Web Services need some proprietary XML schemata:

- **operAttrs.xsd**: operational attributes that can be passed with some requests; for example, the timeout parameter for creation of a user.
- **paramAsg.xsd**: role parameter values in user-role assignments.
- **rpmatchrule.xsd**: role parameter match rules in roles.
- **perm-matchrule.xsd**: match rules in permissions.
- **tsCreateOptions.xsd**: options useful when creating a target system.
- **tsDeleteOptions.xsd**: options useful when deleting a target system.
- **accountGroupMembership.xsd**: states of account-group memberships managed with

accounts (not groups!).

Some tools have deficiencies with respect to wsdl and xsd imports and with handling <xsd:any> definitions. Therefore, we recommend using the wsdl and xsd files delivered with the product rather than the wsdl returned upon the "<patch>?wsdl" URL. We have had good experience using the JAXWS tools of the JEE development kit (for example, the CXF **wsdl2java** tool).

4.16.4. Using the Sample SOAP Connector

To support easy implementation of a SPMLv2 SOAP client, a convenience class **com.siemens.dxm.provisioning.framework.client.SampleSpmlv2Client** is delivered as part of the Web Services client library. It provides a method for sending SPMLv2 requests via the SOAP connector and accepts configuration options in a variety of ways.

You can simply instantiate the class from your Java application, pass the connector's configuration via one of the **open()** methods, create SPMLv2 requests using the classes of the library and pass them to the client's **processSpmlv2Request()** method. The client sends the appropriate SOAP request to the configured URL of the provisioning web service and hands the response back as a SPML **ResponseType** class.

4.16.4.1. Providing Configuration Data

The client requires some configuration properties in order to send SOAP requests to the web service: URL (or port, host, path and ssl), user, password and optionally key store and trust store. For more details, see the SPML SOAP connector of the Java Integration Framework.

The client provides a number of **open** methods that accept configuration data:

open(DxmConnectorConfig connectorConfig)

The Java interface DxmConnectorConfig represents a Java Bean with getter and setter methods for connector and connection properties. The corresponding classes are generated from a XML schema that describes the configuration of a connector within the Java Integration Framework.

You can either insert the configuration properties using the appropriate setter methods or unmarshall the class from a file or string holding the correct XML document. The following snippet shows how to unmarshall the class from a file:

```
FileReader r = new FileReader(conffilename);
DxmConnectorConfig connectorConfig =
Connector.unmarshalConnector(r);
```

open(String conffilename)

If you have the XML configuration document stored in a file, it is sufficient to pass the file name to the client. It then creates the configuration class as described above.

open(Properties props)

The simplest way is to pass the configuration in a Properties class, i.e. a list of properties. You simply need to provide values for the following properties: url (or alternatively port, host, path and ssl), user, password and optionally keystore, keystorePassword, keystoreAlias, truststore, truststorePassword, timeout, maintainSession.

The JUnit class **test.spmlv2.client.TestSpml2SampleClient** shows how to create and use the sample connector. It also shows how to build requests and evaluate responses. The sources of both classes are provided in the **Additions** folder of your product DVD.

5. Workflow Management Web Services

DirX Identity provides Web services for managing request workflow instances. They allow clients to create a workflow instance, read, change, suspend and resume it.

The workflow services are automatically deployed to the embedded Tomcat Web container of each IdS-J server. They cannot be deployed to another, standalone Web container! The Web Service interface is provided as a Web Service Description Language (WSDL) file together with a number of imported XML schemata in the following folder of each IdS-J server of the DirX Identity installation:

install_path/**ids-j-***domain***-S** *n***/extensions/com.siemens.idm.requestworkflow/webapps/workflowService/WEB-INF\wsdl**.

This chapter describes the workflow operations and some other topics, like authentication and authorization.

5.1. Operations

The next sections describe the Workflow Management Web Services operations, including:

listDefinitions - read the workflow definitions matching filter criteria.

createInstance - create and start a new workflow.

getInstance - read the current state of a workflow instance.

listInstances - read the current state of workflow instances matching filter criteria.

getWorklist - read the workflow activities where the user is a participant.

modifyInstance - modify a workflow instance or its activities.

suspend - suspend a workflow .

resume - re-start a workflow.

abort - cancel a workflow or activity.

activate - activate an activity.

verify - starts a verification workflow and returns its error messages - if any.

waitForIdle - wait until the workflow is idle - finished or in a people activity.

notifyParticipantChanged - notify that a participant in a workflow has been changed.

updateCertification - update the provided resource orders in a delta mode.

bulkApproval - accept or reject a couple of approval tasks in one chunk.

Each operation accepts a request element and returns a response element. For more details see the specific chapters.

On a failure, each request will return a fault response. In addition to the attribute **requestID** taken from the request, it contains the following attributes or elements:

requestType - the request name.

error - an error string taken from an enumeration that categorizes the error type.

errorMessage - an error message that typically contains the appropriate error log.

5.1.1. List Definitions

Clients can use listDefinitions operation to obtain a number of workflow definitions matching the passed-in filter criteria.

The service accepts the common attributes **language**, **requestID** and **domain** (see the section on Common Parameters).

In addition to the common element **returnData** (see the section on Common Parameters), the service accepts an optional element **definitionFilter** of type *DefinitionFilterType*. This type contains a number of optional elements that are compared with the corresponding attributes of the workflow definitions:

operation, **subjectType**, **resourceType** and **initiator** are simple String elements simply compared with the corresponding workflow definitions LDAP attributes.

nameFilter - allows you to specify the matching conditions for the workflow definition name, which can be:

the DN of the workflow definition,

its path - an inverse DN notation without the leading workflow definition base (e.g. Definitions/My-Company/...)

or a number of value assertions with a match operation (equals, startsWith, endsWith, contains).

definition - allows you to pass criteria to specify the workflow definition from where to create the instance.

If the client provides the DN of the workflow definition LDAP entry in the nameFilter/dn element, the service instantiates this one without checking any other criteria. If no DN is given, the service takes the path of the workflow, an inverse DN notation without the leading workflow definition base (e.g. Definitions/My-Company/...). In the other cases, the service tries to find the best suitable definition by matching the other input parameters (operation, initiator, subject, resource) with the "When Applicable" criteria of the workflow definitions.

The following optional elements are matched with the "When Applicable" criteria of the workflow definitions:

initiator - the DN of the initiating user.

subject - the order that holds one or more attributes of the workflow subject.

resources - the order that holds one or more attributes of a workflow resource. Typically, a resource represents a user-privilege assignment.

The response contains the attribute **requestID** and a list of matching elements workflowDefinition with the following elements:

name - the name of the workflow definition.

id - the identifier; that is, the DN of the workflow definition.

requestWorkflowType - the type of the request workflow.

description - a description of the workflow with message placeholders resolved by their nationalized text fragments.

5.1.2. Create Instance

With the operation createInstance, a client can create and start a workflow. The operation requires a createInstanceRequest as input and returns a createInstanceResponse. If the DN of a workflow definition is provided, this one is instantiated and filled with the other parameters: subject, resource, initiator, context. Else the input parameters (operation, subjectType, resourceType, initiator, subject, resource) are used to find the best suitable definition (see listDefinitions) and this is instantiated.

The request accepts the following optional (XML) attributes:

waitForTermination - if true, returns when workflow is finished or timeout reached. Default is false.

waitForIdle - if true, returns when workflow is idle (either finished or in a people activity) or timeout reached. Default is false.

timeout - the maximum time in milliseconds until the workflow service is waiting for the workflow to be finished or idle.

causeRefID - the reference ID in audit trails denoting the ID of the causing audit trail.

For the common attributes **language**, **requestID** and **domain**, see the section on Common Parameters.

The request accepts the following optional elements:

correlationID - a parameter that is stored in the workflow context and may be used from a client to identify the workflow started upon a request. Currently, it is used by the ticket service and contains the request ID produced by the client.

definition - allows you to pass criteria to specify the workflow definition from where to create the instance.

If the client provides the DN of the workflow definition LDAP entry in the nameFilter/dn element, the service instantiates this one without checking any other criteria. If no DN is given, the service takes the path of the workflow, an inverse DN notation without the leading workflow definition base (e.g. Definitions/My-Company/...). In the other cases, the service tries to find the best suitable definition by matching the other input parameters (operation, initiator, subject, resource) with the "When Applicable" criteria of the workflow definitions.

initiator - the DN of the initiating user. If it is missing, the service assumes the authenticated user as the initiator. The access policies for executing a workflow are applied to this user.

subject - an order considered the subject of the workflow. It is one of addOrder (subject

is to be created), modifyOrder (subject is to be changed), deleteOrder (subject is to be deleted) and infoOrder (when a resource is to be assigned to that subject).

resources - none, one or many resource orders. Currently, these are user-privilege assignments and can be one of addOrder (assignment to be created), modifyOrder (assignment to be changed) or deleteOrder (assignment to be revoked).

context - a list of String attributes that go into the context of the created workflow. Note that activity context attributes are not evaluated in the create Instance request, but in modify and list Instances requests!

For the common elements like **returnData**, see the section on Common Parameters.

The response contains the attribute **requestID** and the following elements:

workflowID - the workflow ID, which can be used in other operations, especially in modifyInstance to identify the workflow instance.

workflowPath - the path of the workflow instance, i.e. the inverse DN notation without the workflow instance base; for example, My-Company/Approval/4-Eye-Approval/etc.

workflowInstance - the representation of the workflow instance is only returned, if returnData was set to *'everything'*. It contains the simple attributes of the instance such as uid, dn, name, state, applicationState, startDate and expirationDate and complex elements such as the subject, resources the context attributes and the activity representations including their states. See the definition of the WorkflowInstanceType in the XML schema.

5.1.3. Get Instance

With the operation getInstance, a client can read a workflow instance.

In addition to the common attributes **language**, **requestID** and **domain** (see the section on Common Parameters) the service expects the mandatory attribute **workflowID** with the identifier of the workflow. It is returned in the response of operations that return one or more workflow instances; for example, createWorkflow or listInstances.

The response contains the attribute **requestID** and one of the following elements:

workflowID - the workflow ID, which can be used in other operations, especially in modifyInstance to identify the workflow instance.

workflowInstance - if returnData was set to *'everything',* the complete workflow instance is returned in this element. It contains the simple attributes of the instance such as uid, dn, name, state, applicationState, startDate and expirationDate and complex elements such as the subject, resources the context attributes and the activity representations including their states. If returnData was set to "matchingActivities" only the activities matching the activity filter criteria are returned. See the definition of the WorkflowInstanceType in the XML schema.

5.1.4. List Instances

With the operation listInstances a client can obtain a number of workflow instances

matching the passed-in filter criteria.

The service accepts the common attributes **language**, **requestID** and **domain** (see the section on Common Parameters).

In addition to the common element **returnData** (see the section on Common Parameters) the service accepts the optional elements **workflowInstanceFilter** of type WorkflowAndActivityFilterType and **resultConstraints** of type ResultConstraintsType.

The WorkflowAndActivityFilterType type extends a *WorkflowInstanceFilterType* and contains the optional element **workflowActivityFilter** of type ActivityInstanceFilterType.

The WorkflowInstanceFilterType contains the following optional elements:

correlationID - the string to be used by clients (e.g. ticket issuing systems) to correlate workflow instances to internal ID's.

operation - the operation String (create, modify, etc) that is taken from the associated workflow definition.

definitionFilter - an element of type NameFilterType, which contains one of the elements name, path or dn, which refer to the corresponding attributes of the workflow definition.

initiator - the DN of the user who created the workflow.

subjectType - the object description name of the workflow subject; for example, dxrUser.

subjectID - the DN of the workflow subject.

resourceID - the DN of a workflow resource.

resourceType - the object description name of the workflow resource.

requestWorkflowType - the type of workflow.

state - the state of workflow.

started - an element of type TimestampFilterType that allows you to search for workflows, which were started **after**, **before** or exactly **at** a certain point in time. The time is denoted in Z-format; for example, "20100701000000Z".

expires - an element of type TimestampFilterType that allows you to search for workflows, which expire **after**, **before** or exactly **at** a certain point in time. The time is denoted in Z-format; for example, "20100701000000Z".

error - an element of type MultiValueAssertionType, which allows you to specify match criteria for error messages. You can pass a number of <valueAssertion> elements each containing an element <value> and an attribute "matchOperation" with one of the values "equals", "startsWith", "endsWith" or "contains".

owner - the DN of the owner of the workflow definition.

expertFilter - an element of type ExpertFilterType, where you can pass a LDAP filter string in the element <filter>.

The ActivityInstanceFilterType contains the following optional elements -

nameFilter - an element of type MultiValueAssertionType, which allows you to specify match criteria for the name of an activity. You can pass a number of <valueAssertion> elements each containing an element <value> and an attribute "matchOperation" with one of the values "equals", "startsWith", "endsWith" or "contains".

title - an element of type MultiValueAssertionType, which allows you to specify match criteria for the activity title. You can pass a number of <valueAssertion> elements each containing an element <value> and an attribute "matchOperation" with one of the values "equals", "startsWith", "endsWith" or "contains".

type - the string compared with the attribute dxmType of an activity.

activityType - the string compared with the attribute dxmActivityType of an activity.

activitySubType - a specific attribute that indicates the type of a people activity.

dueDateEnabled - a Boolean flag that indicates the existence of a due date for the order.

description - an element of type MultiValueAssertionType, which allows you to specify match criteria for the attribute description of an activity. You can pass a number of <valueAssertion> elements each containing an element <value> and an attribute "matchOperation" with one of the values "equals", "startsWith", "endsWith" or "contains".

state - the state of an activity.

applicationState - the application state of an activity.

participant - the DN of a participant of the activity.

category - the category of an activity.

started - an element of type TimestampFilterType that allows you to search for activities, which were started **after**, **before** or exactly **at** a certain point in time. The time is denoted in Z-format; for example, "20100701000000Z".

expires - an element of type TimestampFilterType that allows you to search for activities, which expire **after**, **before** or exactly **at** a certain point in time. The time is denoted in Z-format; for example, "20100701000000Z".

ended - an element of type TimestampFilterType that allows you to search for activities, which finished **after**, **before** or exactly **at** a certain point in time. The time is denoted in Z-format; for example, "20100701000000Z".

escalationLevel - an integer denoting the escalation level of an activity.

retryCount - an integer denoting the retry count of an activity.

retryLimit - an integer denoting the configured retry limit of an activity.

lastRetryTime - the timestamp when the activity was last restarted after a timeout.

waitBeforeRetry - an integer denoting the configured waiting time before it is re-started after a temporary error.

signApprove - the activity definition option requesting signature in case of an accepted approval.

signDeny - the activity definition option requesting signature in case an approval is rejected.

contentReadOnly - the activity definition option prohibiting changes in an approval

activity.

approvalResult - the result of an approval activity (ACCEPTED or REJECTED), taken from the application state. Note that this attribute is only set at that activity instance, where the decision was taken. Other depending activities may inherit the application state, but not the approval result. This allows you to decide, which approver has taken the decision.

reason - an element of type MultiValueAssertionType, which allows you to specify match criteria for the reason given with an approval. You can pass a number of <valueAssertion> elements each containing an element <value> and an attribute "matchOperation" with one of the values "equals", "startsWith", "endsWith" or "contains".

expertFilter - an element of type ExpertFilterType, where you can pass a LDAP filter string in the element <filter>. Note that multi-value assertion types are OR-based; that is, the multi assertion is fulfilled when at least one contained assertion value is fulfilled.

The **ResultConstraintsType** contains the following optional elements:

applyTo - an element of type ResultConstraintsApplyToEnum. Defines the entry types to which the result constraints apply: workflows level or activities level.

range - an element of type RangeType that contains two elements: **itemFrom** and **itemTo**. Note that the **itemFrom** element is deprecated and should not be used. The element **itemTo** has a default value of **100** and defines the size limit (number of the returned results).

sortCriteria - an element of type SortCriteriaType; used to generate the sort criteria that will be applied on the returned entries.

The response contains the attribute **requestID** and a number of either one of the following elements:

workflowID - the workflow ID, which can be used in other operations, especially in modifyInstance to identify the workflow instance.

workflowInstance - each matching workflow instance is returned in such an element,but only if **returnData** was set to **everything**. It contains the simple attributes of the instance such as uid, dn, name, state, applicationState, startDate and expirationDate and complex elements such as the subject, resources the context attributes and the activity representations including their states. If **returnData** was set to **matchingActivities**, only the activities matching the activity filter criteria are returned. See the definition of the WorkflowInstanceType in the XML schema.

5.1.5. Get Worklist

With the operation getWorklist, a client can obtain the workflows where he is a participant of an activity.

The service accepts the common attributes **language**, **requestID** and **domain** (see the section on Common Parameters).

The client typically doesn't need to supply any parameters. The service returns all workflows where the authenticated user is a participant. More exactly, each workflow contains only

those activities with the user as participant.

The client can optionally supply the following elements to either reduce the number of workflows or change the participant:

participant - the string with the DN of a user. If this element is supplied, not the authenticated user but the given user DN is taken as participant. Only those activities are returned, where that user is a participant. The authenticated user must have the access right to read the participant's activities.

workflowInstanceFilter - the criteria for narrowing the list of returned workflows and activities. This element is described in more detail in the section on list Instances.

resultConstraints - defines the filter and sorting criteria that will be applied on the list of returned workflows and activities. This element is described in more detail in the section "List Instances".

The response contains the attribute **requestID** and a list of **workflowInstance** elements. For more details on a workflow instance, see the section "List Instances". The response also contains the Boolean flag **sizeLimitExceeded** which indicates that the search returned less entries than requested; see also constraints.range.itemTo.

5.1.6. Modify Instance

With the operation modifyInstance, a client can change the state of a workflow instance and of its activities.

In addition to the common attributes **language**, **requestID** and **domain** (see the section on Common Parameters), the service accepts the following optional attributes:

timeout - the maximum timeout in milliseconds to wait.

causeRefID - the string used in audit trails as a reference to the causing audit trail.

In addition to the common element **returnData** (see the section on Common Parameters) the service expects the mandatory element **workflow**. It represents modifications for a workflow and its activities. If you want to set or replace attributes in the workflow instance, just supply the appropriate values in the corresponding elements of the workflow element. To delete a value, you need to set the corresponding "Delete" flag to true. For example, if you want to delete the attribute "description", set the element "descriptionDelete" to true.

You can set the following attributes of a workflow instance:

uid - the identifier of the workflow. It is used just for identification of the workflow and is not modified itself.

dn - the DN of the workflow instance. It is used just for identification of the workflow and is not modified itself.

correlationID - the identifier for correlation of the workflow to client tickets.

name - the common name of the workflow instance.

description - the workflow description.

state - the state of the workflow.

applicationState - the application state of the workflow.

requestWorkflowType - the type of workflow.

startDate - the date and time when the workflow was started. Denoted in Z-format; for example, "20100701000000Z".

endDate - the date and time when the workflow was finished. Denoted in Z-format; for example, "20100701000000Z".

expirationDate - the date and time when the workflow is considered expired. Denoted in Z-format; for example, "20100701000000Z".

initiator - the DN of the user who created the workflow.

subjectType - the object description name of the workflow subject; for example, dxrUser.

subjectID - the DN of the workflow subject.

subject - the order element representing the workflow subject.

resources - the list of resource order elements representing the workflow resources, typically privilege assignments.

context - the list of String attributes that represent the context of the workflow. Note that activity context attributes are not yet implemented!

> The following attributes cannot be changed, but they are available for reading in responses:

operation - the operation(s) of the workflow instance; cannot be changed!

path - the workflow instance path in inverse LDAP notation omitting the instance base node.

The following list of boolean sub-elements of the workflow allow you to delete workflow attributes:

descriptionDelete, stateDelete, applicationStateDelete, startDateDelete, expirationDateDelete, endDateDelete, subjectTypeDelete, subjectIDDelete, subjectDelete, resourcesDelete, initiatorDelete, contextDelete, requestWorkflowTypeDelete.

The <workflow> element can contain a number of <**activity**> elements that allow you to change certain attributes of an activity. The activity attributes you can change are:

description - a description of the activity. Note that the original description stored in LDAP contains placeholders for nationalization that are replaced in responses. Make sure that you use such placeholders in your modified text.

title - the title of the activity. Note that the original title stored in LDAP contains placeholders for nationalization that are replaced in responses. Make sure that you use such placeholders in your modified text.

category - the category of the activity.

state - the workflow state.

applicationState - the workflow application state.

startDate - the time when activity was started. Denoted in Z-format; for example, "20100701000000Z"

expirationDate - the time when the activity is considered expired. Denoted in Z-format, e.g. "20100701000000Z".

endDate - the date and time when the activity was finished. Denoted in Z-format; for example, "20100701000000Z".

participant - the list of user DNs that are participants of the activity.

escalationLevel - the integer representing the current escalation level.

retryCount - the integer denoting how often a timeout of the activity has occurred within one escalation level.

retryLimit - the integer denoting the maximum number of retries after a timeout until the activity reaches the next escalation level.

reason - the reason for accepting or rejecting of a participant in an approval activity.

activityType - the type of an activity as configured in its definition; related to LDAP attribute *dxmType*.

activitySubType - the type of an activity as configured in its definition; related to LDAP attribute *dxmActivityType*.

approvalResult - the result of an approval activity (ACCEPTED or REJECTED), taken from the application state. Note that this attribute is only set at that activity instance, where the decision was taken. Other depending activities may inherit the application state, but not the approval result. This allows you to decide, which approver has taken the decision.

signApprove - the activity definition option requesting signature in case of an accepted approval.

signDeny - the activity definition option requesting signature in case an approval is rejected.

contentReadOnly - the activity definition option prohibiting changes in an approval activity.

The following **activity** attributes cannot be changed, but they are available for reading in responses:

name - the name of the activity.

dn - the DN of the activity instance.

The following list of boolean sub-elements of the **activity** allow you to delete activity attributes:

descriptionDelete, titleDelete, stateDelete, applicationStateDelete, categoryDelete, startDateDelete, expirationDateDelete, endDateDelete, participantDelete, escalationLevelDelete, retryCountDelete, retryLimitDelete, reasonDelete,

activityTypeDelete, contextDelete, approvalResultDelete.

5.1.7. Suspend

On receipt of the **suspend** request the workflow service puts the workflow instance into suspended state. It can later on be resumed by the **resume** request.

In addition to the common attributes **requestID** and **domain** (see the section on Common Parameters) the service expects the mandatory attribute **workflowID** and an optional **timeout**:

workflowID - the identifier of the workflow to be suspended.

timeout - the maximum timeout in milliseconds to wait until processing of the request is finished.

The response contains only the optional **requestID**.

5.1.8. Resume

On receipt of the **resume** request, the workflow service resumes a previously suspended workflow instance. If also an activity name has been supplied, the workflow service assumes this activity to be in state "waitInError" and resets it, i.e. starts it again.

In addition to the common attributes **requestID** and **domain** (see the section on Common Parameters) the service expects the mandatory attribute **workflowID** and the optional attributes **timeout** and **activityName**:

workflowID - the identifier of the workflow to be suspended.

activityName - the name of an activity that is assumed to be in state "waitInError".

timeout - the maximum timeout in milliseconds to wait until processing of the request is finished.

The response contains only the optional **requestID**.

5.1.9. Abort

On receipt of the **abort** request, the workflow service cancels the workflow instance.

In addition to the common attributes **requestID** and **domain** (see the section on Common Parameters) the service expects the mandatory attribute **workflowID** and the optional attribute **timeout**:

workflowID - the identifier of the workflow to be aborted.

timeout - the maximum timeout in milliseconds to wait until processing of the request is finished.

The response contains only the optional **requestID**.

5.1.10. Activate

On receipt of the **activate** request, the workflow service finishes the designated activity with state SUCCEEDED, if it is running and the activation key stored in the workflow context matches the one that was supplied in the request. See also operation verify.

This operation is typically used in user registration scenarios: The first part of the registration is realized by a verification workflow, where the user enters a number of attributes, especially their email address. The workflow verifies that the user can be created. On success, the client Web application creates an activation workflow. This creates a unique activation key and sends an email to the user containing this key. At the Web application, the user supplies his activation key and the application triggers the activate operation supplying the identifier of the activation workflow. In the activation operation, the workflow service matches the activation key with the one stored in the workflow and finishes the activity with the given name. Then the activation workflow can continue and store the new user in the data store.

In addition to the common attributes **requestID**, **language**, **returnData** and **domain** (see the section on Common Parameters) the service expects the mandatory attributes **workflowID**, **activityName** and **activationKey**:

workflowID - the identifier of the workflow.

activityName - the name of an activity, which is expected to be in state RUNNING.

activationKey - the key to be compared with the activation key stored in the workflow context. If they match, the activity state is set to SUCCEEDED.

The response contains the attribute **requestID** and one of the following elements depending on the input parameter returnData:

workflowID - the workflow ID, which can be used in other operations, especially in modifyInstance to identify the workflow instance.

workflowInstance - if returnData was set to *'everything',* the complete workflow instance is returned in this element. It contains the simple attributes of the instance such as uid, dn, name, state, applicationState, startDate and expirationDate and complex elements such as the subject, resources the context attributes and the activity representations including their states. See the definition of the WorkflowInstanceType in the XML schema.

5.1.11. Verify

On receipt of the **verify** request, the workflow service starts a verification workflow, waits until the workflow finishes and returns error messages stored in the workflow context. The workflow is either identified by its name or if the name is missing an applicable workflow is found by evaluating the operation "verify" and the attributes of the given subject. See also operation activate.

This operation is typically used in user registration scenarios: The first part of the registration is realized by a verification workflow, where the user enters a number of attributes, especially their email address. The workflow verifies that the user can be created.

In case of success the client Web Application creates an activation workflow. This creates a unique activation key and sends an email to the user containing this key. At the Web Application, the user supplies their activation key and the application triggers the activate operation supplying the identifier of the activation workflow. In the activation operation the workflow service matches the activation key with the one stored in the workflow and finishes the activity with the given name. Then the activation workflow can continue and store the new user in the data store.

In addition to the common attributes **requestID** and **domain** (see the section on Common Parameters) the service expects the optional attributes **workflowDefinitionName** and **timeout**:

workflowDefinitionName - the name of a the workflow definition to be created.

timeout - the maximum timeout in milliseconds to wait until processing of the request is finished.

The service evaluates the following elements:

initiator - the DN of the user who is considered the workflow initiator.

subject - an addOrder representing the attributes of the user to be created.

context - a list of attributes (tag/value) that are stored into the context of the created workflow.

The response contains the attribute **requestID** and the following element:

errorMessage - the error messages that the created workflow produced .

5.1.12. Wait for Idle

With the operation **waitForIdle** the workflow service waits until the workflow is in idle state. A workflow is considered in idle state, if it is finished or a people activity is running, i.e. waiting for input of a participant.

In addition to the common attributes **requestID**, **language**, **returnData** and **domain** (see the section on Common Parameters) the service expects the mandatory attribute **workflowID** and an optional **timeout**:

workflowID - the identifier of the workflow.

timeout - the maximum timeout in milliseconds to wait until the service returns even if the workflow is not in the idle state.

The response contains the attribute **requestID** and one of the following elements:

timedout - If this boolean attribute is true, the timeout was exceeded and the client cannot expect that the workflow is in the idle state.

workflowInstance - if returnData was set to *'everything',* the complete workflow instance is returned in this element. It contains the simple attributes of the instance such as uid, dn, name, state, applicationState, startDate and expirationDate and complex elements such as the subject, resources the context attributes and the activity

representations including their states. See the definition of the WorkflowInstanceType in the XML schema.

5.1.13. Notify Participant Changed

With the **notifyParticipantChanged** operation, the client informs the workflow service that the participant of an activity has been changed. The workflow service assumes that the state of the activity is set to RUNNABLE and requests the workflow engine to re-start the activity. If configured accordingly, an email will be sent to the new participant.

In addition to the common attributes **requestID** and **domain** (see the section on Common Parameters) the service expects the mandatory attribute **workflowID** and an optional **timeout**:

workflowID - the identifier of the workflow.

timeout - the maximum timeout in milliseconds to wait until processing of the request is finished.

The response contains the attribute **requestID** and one of the following elements:

timedout - if this boolean attribute is true, the timeout was exceeded and the client cannot expect that the workflow is in the idle state.

workflowInstance - if returnData was set to *'everything',* the complete workflow instance is returned in this element. It contains the simple attributes of the instance such as uid, dn, name, state, applicationState, startDate and expirationDate and complex elements such as the subject, resources the context attributes and the activity representations, including their states. See the definition of the WorkflowInstanceType in the XML schema.

5.1.14. Update Certification

The **updateCertification** operation is a convenience operation for handling approvals in certification or attestation workflows. It allows the client to update a potentially long list of resources of an approval in several steps. Only if the client sets the submit flag to true and all resources are processed, the activity is considered finished, its state set to SUCCEEDED and its application state to ACCEPTED.

In addition to the common attributes **requestID** and **domain** (see the section on Common Parameters), the service expects the following attributes:

workflowID - the identifier of a the workflow instance.

activityName - the name of the approval activity, which is to be finished on submit = true.

submit - if this boolean flag is set to true, the activity is finished: its state is set to SUCCEEDED and its application state to ACCEPTED.

timeout - the maximum timeout in milliseconds to wait until processing of the request is finished.

The service evaluates the following elements:

initiator - the DN of the user who is considered the workflow initiator.

resource - a list of order's considered the resources of the workflow. In case of a deleteOrder, the associated assigned is considered to be revoked.

context - a list of attributes (tag/value) that are stored into the context of the created workflow.

The response contains the attribute **requestID** and the flag **complete**, which indicates whether the activity is finished

5.1.15. Bulk Approval

The bulkApproval operation is a fast operation for handling approvals. It allows the client to approve (accept or reject) multiple workflow tasks with one request. The operation returns the response to the client immediately after the modifications in the LDAP directory are completed. This operation performs direct LDAP modifications and does not use the Identity service layer. It finishes the activity by setting the activity state to SUCCEEDED and the application state to ACCEPTED. It also removes the approver from the list of current approvers in the workflow entry, so that a subsequent request for the approver's work list will not contain this workflow. The Web Service notifies the workflow engine and writes audit records in separate threads so they do not block returning the response.

The bulkApproval operation ignores all common attributes except the requestID.

The service expects the following attributes:

participantDN (optional) - the DN of the user whose tasks are approved. It is only needed, if the authenticated user approves the tasks of another person. In this case an access policy has to allow this.

accept - the list of requests to be approved.

reason (optional) - the reason for accepting a request. It is applied to all accepted requests.

workflowID - a list with workflow ID's to be approved. The service approves all currently running tasks of the approver in this workflow.

reject - the list of requests to be rejected.

reason (optional) - the reason for rejecting a request. It is applied to all rejected requests.

workflowID - a list with workflow ID's to be rejected. The service rejects all currently running tasks of the approver in this workflow.

workflowContext - not currently used by the bulkApproval service.

The response contains the attribute requestID and the failed list, which contains failed workflow ID's. Each entry from the failed list contains:

id - the workflow ID.

name - the workflow or activity name.

errorCode - 1x for error at activity level, 2x for error at workflow level, 3x for error related to participant, 4x for invalid workflow ID.

auditRecord - for internal use only; always empty.

5.2. Implementing a Client

Using the WSDL and the XML schemata, you can generate client stubs in various technologies (for example, Java, C#, PHP).For a Java client we recommend the standard tools of the JAXWS 2 development kit.

> Subject and resource orders of a request workflow follow the schema with the namespace "urn:siemens:dxm:ORDER:1:0".It is defined in the schema 'order.xsd', which in turn imports SPML 1.0, DSML 2.0, XML Signature, SAML 1.0 and some proprietary schemata (SPMLext.xsd, auditRecord-base.xsd).If you generate your client stub based on the wsdl, these schema elements are not produced, because they are only referenced via the <any> syntax: *<any namespace="urn:siemens:dxm:ORDER:1:0" processContents="lax"/>*.Therefore, you have to apply the JAXB source generator explicitly for the XML schema *order.xsd,* which recursively imports the other schema files (SPML, etc) mentioned above.

If your client is developed in Java and running on a JRE >= 6, you can make use of the classes already deployed with DirX Identity: The stub classes were generated with JAXWS 2 and compiled to the file '*com.siemens.idm.workflow.services.api.jar*'.The order elements were generated already in earlier versions using the Open Source toolkit Castor and are in the files *dxmOrder.jar* and *dxmSpml.jar*.

The remainder gives some sample snippets that show how to instantiate a modify request and especially how to create and read orders:

Some constant definitions needed later on:

```
private static final String APPROVAL_ACTIVITY = "Approval by
Privilege Managers";
private static final String U_TASPATCH_NIK = "cn=Taspatch
Nik,ou=Global IT,o=My-Company,cn=Users,cn=My-Company";
private static final String U_WAGNER_RETHA = "cn=Wagner
Retha,ou=Global IT,o=My-Company,cn=Users,cn=My-Company";
private static final String U_ABELE_MARC = "cn=Abele
Marc,ou=Finances,o=My-Company,cn=Users,cn=My-Company";
```

Set the user and password into the HTTP request:

Suppose you obtained the port as WorkflowPortType from the generated stub. Then get the request context from the port and set user, password and URL the following way:

```
BindingProvider bp = (BindingProvider) port;
Map<String, Object> reqContext = bp.getRequestContext();
reqContext.put(BindingProvider.USERNAME_PROPERTY, user);
reqContext.put(BindingProvider.PASSWORD_PROPERTY, pwd);
reqContext.put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY, endpoint);
```

Instantiate a modify request type:

```
ModifyInstanceRequestType req = new ModifyInstanceRequestType();
req.setRequestID("string identifying this request");
req.setTimeout(5000L);
```

Add the modification of the workflow description:

```
WorkflowInstanceModificationType wfMod = new
WorkflowInstanceModificationType();
req.setWorkflow(wfMod);
wfMod.setUid("<uid> obtained from a getInstance response");
wfMod.setDescription("newDescription");
```

Add the modification of a workflow context attribute:

```
AttributesType attrs = new AttributesType();
AttrType t = new AttrType();
t.setName("newName");
t.setValue("newValue");
attrs.getAttr().add(t);
wfMod.setContext(attrs);
```

Modification of the participants of an activity:

```
ActivityInstanceModificationType actMod = new
ActivityInstanceModificationType();
wfMod.setActivities(new ActivityInstancesModificationType());
wfMod.getActivities().getActivityInstance().add(actMod);
actMod.setName(APPROVAL_ACTIVITY);
actMod.getParticipant().add(U_ABELE_MARC);
```

Create an info order as subject:

```
OrderType infoOrder = null;
try {
OrderFactory orderFactory = new OrderFactoryImpl();
OrderRequest infoOrder = (OrderRequest) orderFactory.createInfoOrder(
   U_WAGNER_RETHA, //subject DN
   "info." + Calendar.getInstance().getTimeInMillis(), //request id
   U_TASPATCH_NIK,  // creator DN
   null, //activationDate
  "dxrUser", //directory type
  "subject Uid",  // subject's UID required for auditing
  "dxrUser", // audit object type
  null // map of identifier attributes, required for auditing
);

    infoOrder = orderTypeFromOrder(infoOrder);
wfMod.setSubject(infoOrder);
}catch (Exception e) {
//log.error("Can not create assign order");

}
```

This method helps to transform an order to an order type, which is needed in requests:

```
/**
* @param order order from workflow, subject or resource.
* @return OrderType.
*/
public static OrderType orderTypeFromOrder(Order order) {
       if (order == null)
                 return null;
        try {
                 OrderType ret = new OrderType();
                  Element subjectElem =
AbstractOrderFactoryImpl.toDomElement((OrderRequest) order);
                  ret.setAny(subjectElem);
                  return ret;
        } catch (Exception e) {
                  //log.error(e.getClass().getCanonicalName() + "
while converting " + order.getClass().getSimpleName() + " to
```

```
OrderType: " + e.getMessage());
        } catch (NoClassDefFoundError e) {
                  //log.error(e.getClass().getCanonicalName() + "
while converting " + order.getClass().getSimpleName() + " to
OrderType: " + e.getMessage());
        }
        return null;
}
```

The following method helps transforming an OrderType to an order, which is needed in evaluating responses:

```
/**
 * @param orderType orderType from workflow - subject or resource,
e.g. wfRsp.getSubject().
 * @return order unmarshalled from DOM in order type.
 */
public static Order orderFromOrderType(OrderType orderType) {
       if (orderType == null)
                 return null;
        Element anyElem = (Element) orderType.getAny();
        if (anyElem == null)
                  return null;
        try {
                  return new OrderFactoryImpl().create(anyElem);
        } catch (Exception e) {
                  //log.error(e.getClass().getCanonicalName() + "
while converting OrderType to Order: " + e.getMessage());
                  return null;
        }
}
```

5.3. Authentication

The workflow services support HTTP basic authentication based on SSL/TLS protocol and SAP cookies out of the box.

The default authenticator is configured as a Tomcat valve in the file **context.xml** in the **META-INF** folder of the Web application workflowService: *install_path***/ids-j-***domain***-S***n***/extensions/com.siemens.idm.requestworkflow/webapps/workflowService/META-INF/context.xml**.It handles HTTP basic authentication and proprietary authentication from Web Center applications.

Other authentication form factors, such as Single Sign-On (SSO) authentication via HTTP header can be realized using the standard Tomcat filters and valves.Support of SOAP authentication can be realized in projects by adding handlers into the JAXWS stack of the Web service.They are configured in the file **sun-jaxws.xml**, which is located in the **WEB-INF** folder of the Web application.See the JAXWS documentation for more details.The workflow service reads the DN of the authenticated user from the Web Service context: context.getUserPrincipal().

5.4. Common Parameters

This section describes attributes and elements that are used in all or at least several requests.

Attributes

requestID - the identifier of the request; will be returned in the response.

domain - if provided, will be checked against the common name of the Identity domain with which the IdS-J server is associated.This attribute helps to prevent a request from being issued to the wrong domain.

language - used for nationalizing the response. The default is "en".

Elements

returnData - if supplied, controls the amount of data returned in the response. Accepted values are:

identifier - the response contains only the identifier of the workflow.

matchingActivities - the response contains only the workflow(s) and those activities that match the given activity filter.

everything - returns all data - workflow(s) with all activities.

nothing - the response doesn’t contain any data.

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, Atos Group – 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.