HP OpenView Select Identity

Connector Developer Guide

Software Version: 3.3.1

July 2005

Legal Notices

Warranty

Hewlett-Packard makes no warranty of any kind with regard to this document, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

A copy of the specific warranty terms applicable to your Hewlett-Packard product can be obtained from your local Sales and Service Office.

Restricted Rights Legend

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013.

Hewlett-Packard Company
United States of America

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19(c)(1,2).

Copyright Notices

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

Select Identity uses software from the Apache Jakarta Project including:

- Commons-beanutils.
- Commons-collections.
- Commons-logging.
- Commons-digester.
- Commons-httpclient.

- Element Construction Set (ecs).
- Jakarta-poi.
- Jakarta-regexp.
- Logging Services (log4j).

Additional third party software used by Select Identity includes:

- JasperReports developed by SourceForge.
- iText (for JasperReports) developed by SourceForge.
- BeanShell.
- Xalan from the Apache XML Project.
- Xerces from the Apache XML Project.
- Java API for XML Processing from the Apache XML Project.
- SOAP developed by the Apache Software Foundation.
- JavaMail from SUN Reference Implementation.
- Java Secure Socket Extension (JSSE) from SUN Reference Implementation.
- Java Cryptography Extension (JCE) from SUN Reference Implementation.
- JavaBeans Activation Framework (JAF) from SUN Reference Implementation.
- OpenSPML Toolkit from OpenSPML.org.
- JGraph developed by JGraph.
- Hibernate from Hibernate.org.
- BouncyCastle engine for keystore management, bouncycastle.org.

Trademark Notices

HP OpenView Select Identity is a trademark of Hewlett-Packard Development Company, L.P.

Microsoft, Windows, the Windows logo, and SQL Server are trademarks or registered trademarks of Microsoft Corporation.

Sun™ workstation, Solaris Operating Environment™ software, SPARCstation™ 20 system, Java technology, and Sun RPC are registered trademarks or trademarks of Sun Microsystems, Inc. JavaScript is a trademark of Sun Microsystems, Inc., used under license for technology invented and implemented by Netscape.

This product includes the Sun Java Runtime. This product includes code licensed from RSA Security, Inc. Some portions licensed from IBM are available at http://oss.software.ibm.com/icu4j/.

IBM, DB2 Universal Database, DB2, WebSphere, and the IBM logo are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group.

This product includes software provided by the World Wide Web Consortium. This software includes xml-apis. Copyright © 1994-2000 World Wide Web Consortium, (Massachusetts Institute of Technology, Institute National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/

Intel and Pentium are trademarks or registered trademarks of Intel Corporation in the United States, other countries, or both.

AMD and the AMD logo are trademarks of Advanced Micro Devices, Inc.

BEA and WebLogic are registered trademarks of BEA Systems, Inc.

VeriSign is a registered trademark of VeriSign, Inc. Copyright © 2001 VeriSign, Inc. All rights reserved.

All other product names are the property of their respective trademark or service mark holders and are hereby acknowledged.

Support

Please visit the HP OpenView web site at:

http://www.managementsoftware.hp.com/

This web site provides contact information and details about the products, services, and support that HP OpenView offers.

You can also go directly to the support web site at:

http://support.openview.hp.com/

HP OpenView online software support provides customer self-solve capabilities. It provides a fast and efficient way to access interactive technical support tools needed to manage your business. As a valuable support customer, you can benefit by using the support site to:

- Search for knowledge documents of interest
- Submit and track progress on support cases
- Manage a support contract
- Look up HP support contacts
- Review information about available services
- Enter discussions with other software customers
- Research and register for software training

Most of the support areas require that you register as an HP Passport user and log in. Many also require a support contract.

To find more information about access levels, go to:

http://support.openview.hp.com/access_level.jsp

To register for an HP Passport ID, go to:

https://passport2.hp.com/hpp/newuser.do

contents

1

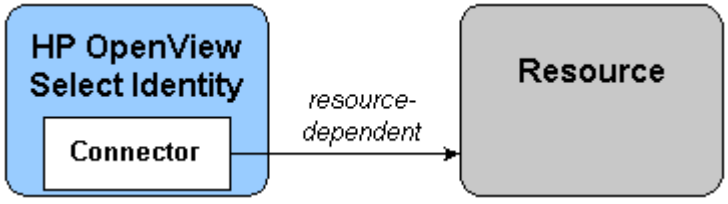
Introduction to Connectors

HP OpenView Select Identity enables you to connect to enterprise applications and resources to configure and manage users, groups, and entitlements in those systems. The component that enables Select Identity to access a resource is called a **connector**.

Overview of Select Identity Connectors

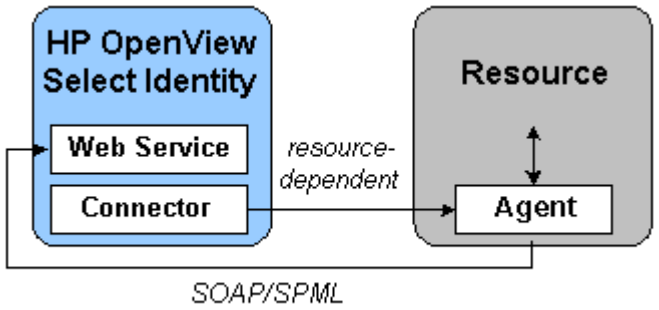
Select Identity supports two types of connectors:

- A one-way connector initiates communication with a resource. If a resource is supported by a one-way connector, provisioning operations initiated in Select Identity are synchronized with the resource through the connector. The following diagram illustrates the flow of data:



The connector resides on the Select Identity server and sends requests to the resource. The resource defines the protocol that must be used by the connector to issue the request. To create a one-way connector, you must create the connector and install it on the Select Identity server.

- A two-way connector comprises the connector that resides on the Select Identity server and an agent that resides on the resource. The connector communicates with the agent and the agent executes the provisioning operations. The agent also listens for changes on the host resource and sends synchronization notifications to Select Identity. Thus, a two-way connector enables data to flow in two directions, as illustrated in the following diagram, and changes to users can occur on either system.



The connector must issue a request according to the resource's specifications. When the agent issues a request to Select Identity's Web Service, it must use the SOAP protocol to send an SPML payload over HTTP or HTTPS.

The advantages of a two-way connector are as follows:

— Improved security — Some resources provide no secure transport and some systems do not encrypt passwords. With a two-way connector, HTTPS can be used to secure communications from the resource to Select Identity.

— Availability — If Select Identity becomes unavailable, you can configure the agent to retry failed requests sent to the server once the server is available again.

— Change detection and reverse synchronization — Many resources do not have a mechanism to easily detect changes and update the resource or a central management system. An agent can detect

changes made on the resource and securely propagate them to the Select Identity server.

J2EE Connector Architecture (JCA)

Creating a Select Identity connector entails building a J2EE Connector architecture (JCA) resource adapter. JCA provides a Java solution to the problem of connectivity between application servers and enterprise information systems (EISs). This architecture is based on technologies that are defined and standardized as part of the Java 2 Platform, Enterprise Edition (J2EE).

For a general overview of JCA, refer to the following page:

http://java.sun.com/j2ee/white/connector.html

You can download the specification from the following page:

http://java.sun.com/j2ee/connector/download.html

Note that Select Identity implements only the Connector Management portion of the JCA specification. You can also refer to the following URL if you are creating a WebLogic connector:

http://e-docs.bea.com/wls/docs81/jconnector/index.html

You must have an understanding of the Java Developer Kit (JDK), version 1.4, and you should be familiar with the JCA, version 1.0. In addition, Select Identity has provided a Connector API that is used in conjunction with JCA to create connectors. For information about the J2EE APIs, including those for connectors, refer to **http://java.sun.com/j2ee/1.4/docs/api/index.html**.

Development Phases

This section outlines the steps that are typically involved in the development of a connector. It is strongly recommended that you take the time to address each phase and plan for the connector's development carefully.

Requirements Phase

Ensure that the resource supports a mechanism for user provisioning by external clients, in a secure and reliable manner. You must have an understanding of the underlying resource, including knowledge of the resource's tools and administration API. You may also need to obtain an administrative account that has privileges to provision.

Collect requirements for development, as follows:

1 Determine the requirements based on the resource system.

- What identity information will be provisioned (users or other objects)?
- What are the entitlements supported by the resource? Typically, resources support groups (groups or users), roles, access control levels (ACLs), privileges, and so on.
- What are the supported attributes of the identity object based on the schema in the resource?
- How is the schema retrieved from the resource?
- How is the identity object addressed on the resource? This could be a DN (for LDAP-type of resources), an SSN,a user ID, hierarchical naming, and so on. This will be used as the primary key to address the identity object.
- How does the resource application support connectivity for external systems to provision identity information? This might mean accessing the system through API calls, RMI, JMS, a Web Service, a CLI such as telnet, ssh, and so on.
- If the resource already supports a connector interface, how can you develop the Select Identity connector leveraging the existing connector?
- Does the resource support an SDK or a development toolkit for administration, which might include JAR files or libraries for making calls to access and provision information?
- Are there security requirements to consider? Is SSL or any proprietary encryption/decryption information required between the connector and the resource?

- What are the performance requirements? How many objects can the resource support? How may entitlements? How many users can the connector create, read, update, or delete in a second, minute, or hour?
- What are the scalability requirements? How many connections does it support? Can the same connector support similar resources through configuration support for transactions?
- Does the resource support synchronous or asynchronous connectivity? It is possible that the resource cannot finish provisioning immediately and might finish the job at a later time. How does the connector know when the resource operation is done and how does it handle the response from the resource?
- Is the connector required to maintain state? If so, what is the required schema?

2 Determine access requirements for the resource.

- What are the addressing parameters such as TCP/IP address, port number, URL,and secure IDs?
- Is there authentication information (user ID and password)?
- Are there secure channel parameters?
- Does the connection pass through a proxy server or a firewall? If so, what are the parameters involved?

3 Determine the requirements for error reporting.

- What errors are supported by the resource?
- What kind of exceptions are reported to Select Identity?
- What kind of errors in the resource are reported to Select Identity?
- What are the recoverable and non-recoverable exceptions?

4 Determine the requirements for reverse synchronization.

- What changes to identity objects on the resource must be synchronized with Select Identity. For example, if a user's password or address changes on the resource, is there a requirement that Select Identity should be notified about this?
- How often do changes occur? Are they done in real time or as a batch job at the end of the day?

- How is information obtained from the resource? The resource might support an audit log of all changes on the resource, or it might support a log of all events that are triggered by someone like an administrator. How is this information retrieved from the resource? Should the connector support a pull model or a push model?

5 Detemine the requirements for child transactions.

- Is an operation invoked on the resource that might trigger child operations within the resource?
- How should the connector notify Select Identity of the status of child operations?
- What status information about child operations should be reported to Select Identity?
- Is the operation is “atomic” or a “best-effort?”
- How does the connector determine when the operation is done?
- Does the resource automatically rollback all previous successful child operations if one child operation fails?

6 Determine requirements for the policies supported by the resource.

- What are the policies for the identity objects? For example, the primary key of the identity object must be obtained from another external system.
- What are the attribute policies? For example, password policy might restrict in the size, content (maximum lenght, minimum length, maximum number of alphabetic characters, minimum number of numeric charactrs, and so on), encryption (one-way or two-way), and so on. What are the limitations on attribute size, masking, and other parameters?

Design Phase

Design the connector you will implement following these guidelines:

1 Provide a high-level design of the approach taken for the provisioning process. Provide the following:

- Mapping of functionality to be supported by the connector to the functionality supported by the resource.

- Mapping of the Select Identity schema to the schema (attribute information) supported by the resource. This is also referred to as the forward mapping.
- The Connector API methods that are supported by the connector implementation.
- Reverse mapping of the attribute information at the time of reverse synchronization.
- How the implementation solves the cyclic update problem. For example, a change in object's information triggers an update on the resource, which might in turn trigger a reverse synchronization with Select Identity for the same object, and vice-versa.
- Use of the JCA framework in the design. Define how the connector makes use of the framework to address some of the requirements.
- Resource product version. Provide any functionality changes between versions of the resource application.

2. Provide information about how to address the various requirements: synchronous versus asynchronous processing, scalability, performance, security, and so on.
   - Can the connector handle a large number of identity objects, such as users?
   - Can it handle large number of entitlements? Is caching, paging, batch loading, or file loading is used by the connector?
   - Can it handle large number of resources?
3. Define whether the connector is agent-based or agent-less.
   - Agent-based requires that an agent is installed on the resource with which the connector implementation interacts. The agent in turn interacts with the resource or the operating system. Reverse synchronization is generally possible with an agent-based solution. On the other hand, an agent-based implementation requires an installation effort and administration on the resource system.
   - An agent-less connector requires complete out-of-box support for all provisioning operations by the resource or through an SDK.
   - Address the advantages and disadvantages for both solutions.

Implementation

Specific information about how to implement the JCA and Connector API methods is provided in Interface, Class, and Method Implementations on page 31. This procedure provides a general overview.

1. Start with a sample application that can provision identity objects and perform entitlement assignment s on the identity objects in the resource.
2. Implement all of the required Select Identity Connector methods to create, read, update, and identity objects, leveraging the sample application.
   - The main interface to implement is TAConnector interface.
   - Implement the connector parameter factory, which creates instances of connection parameter beans.
3. Implement all entitlement association and dissociation methods.
4. Implement all required interfaces in the JCA CCI framework, which enables the connector as a Resource Adapter.
5. If necessary, implement an agent to run on the resource machine.
6. Implement a secure way of communication between the connector and resource, and vice versa. If necessary, use certificates.
7. Implement modules to send SOAP messages containing SPML to the Select Identity Web Service for reverse synchronization (password synchronization and identity object reverse synchronization).
8. If necessary, install the EJB driver for unit testing the connector.
9. If necessary, install the client driver for testing the connector.
10. Use IDEs for the development and ANT for build tools.
11. Use the JDK, J2EE, and third-party libraries for further development.
12. Implement junit test cases.

Integration

Verify the connector's integration with Select Identity as follows:

1. Verify that Select Identity is loading and using the connector as a resource adapter to communicate with the new resource.
2. Create a Service that uses this resource.
3. Provision users in the Service, verifying that they are successfully created in the resource.
4. Associate and disassociate entitlements with users.
5. Verify integration with the Select Identity Web Service for user provisioning through SPML payloads.

Packaging

Detailed information about packaging the connector is described in Coding the Connector on page 30. This provides a general overview:

1. Include all libraries required by the connector in a RAR file.
2. Test the client for unit testing.
3. Determine any schema information (ddl, dml) needed by the connector.
4. Obtain all third-party software licenses and their installation procedures.

Documentation

For future maintenance and distribution, compile the following information about the connector:

- Detailed documentation on the requirements and design
- User guides
- Configuration guides
- Functionality mapping document
- Schema (or attribute) mapping document
- Installation guides, for agent-less and agent-based solutions

- Javadoc
- Documentation of encryption/decryption used, port numbers of agent, size of agent foot print, and so on
- Requirements on the system administrator to install the agent on the resource
- Administration documents

Product Documentation

The Select Identity product documentation includes the following:

- Release notes are provided in the top-level directory of the HP OpenView Select Identity CD. This document provides important information about new features included in this release, known defects and limitations, and special usage information that you should be familiar with before using the product.
- For installation and configuration information, refer to the *HP OpenView Select Identity Installation and Configuration Guide*. All installation prerequisites, system requirements, and procedures are explained in detail in this guide. Specific product configuration and logging settings are included. This guide also includes uninstall and troubleshooting information.
- An *HP OpenView Connector Installation and Configuration Guide* is provided for each resource connector. These are located on the Select Identity Connector CD.
- The *HP OpenView Select Identity Attribute Mapping Utility User's Guide* describes how to access the Attribute Mapping Utility, provides an overview to the utility's user interface, and describes how to define user and entitlements mappings. This guide is provided on the Select Identity Connector CD and is for use with the SQL and SQL Admin connectors only.
- Detailed procedures for deployment and system management are documented in the *HP OpenView Select Identity Administrator Guide* and Select Identity online help system. This guide provides detailed concepts

and procedures for deploying and configuring the Select Identity system. In the online help system, tasks are grouped by the administrative functions that govern them.

- The *HP OpenView Select Identity Workflow Studio Guide* provides detailed information about using Workflow Studio to create workflow templates. It also describes how to create reports that enable managers and approvers to check the status of account activities.
- The *HP OpenView Select Identity External Call Developer Guide* provides detailed information about creating calls to third-party applications. These calls can then be deployed in Select Identity to constrain attribute values or facilitate workflow processes. In addition, JavaDoc is provided for this API. To view this help, extract the `javadoc.jar` file in the `docs/api_help/external_calls/Javadoc` directory on the HP OpenView Select Identity CD.
- If you need to develop connectors, which enable you to connect to external systems for provisioning, refer to the *HP OpenView Select Identity Connector Developer Guide*. This document provides an overview of the Connector API and the steps required to build a connector. The audience of this guide is developers familiar with Java.

  JavaDoc is also provided for the Connector API. To view this help, extract the `javadoc.jar` file in the `docs/api_help/connectors/Javadoc` directory on the HP OpenView Select Identity CD.
- The *HP OpenView Select Identity Web Service Developer Guide* describes the Web Service, which enables you to programmatically provision users in Select Identity. This guide provides an overview of the operations you can perform through use of the Web Service, including SPML examples for each operation.

  An independent, web-based help system is available for this API. To view this help, double-click the `index.htm` file in the `docs/api_help/web_service/help` directory on the HP OpenView Select Identity CD.

2

Implementing a Connector

To implement a connector, follow this general procedure. Details are provided in subsequent sections of this chapter. You may also find it useful to refer to LDAP Connector Example on page 80 or Dummy Connector Example on page 87 for an overview of those connectors.

To support a connector on a non-US resource, an internationalized resource provider interface must be available. Thus, support for internationalization is provided by the resource, not the connector.

1. If you have not done so, review Development Phases on page 10 in preparation for this procedure. Also, refer to Requirements on page 22 for a list of tools and information necessary to implement a connector, and refer to Overview of the Select Identity Connector API on page 24 to understand the packages provided by Select Identity.
2. Gather the parameters needed to code the connector and create the properties filesbefore building it. See Gathering Connector Parameters on page 27 for more information.
3. Code the Java classes and implement the interfaces that will comprise the connector. See Coding the Connector on page 30 for details.
4. Create a mapping file that maps each attribute on the physical resource to an attribute on the connector. See Understanding the Resource Schema on page 26 and Creating a Mapping File on page 72 before creating the file.

5 Build the connector. Select Identity provides all of the base classes you need to build a connector in a file named `clientintf.jar`, which resides on the Select Identity CD. Use the contents of this file to build your connector. See Description of the Build Files on page 85 for a listing of build files created to build the LDAP connector.

Also, the Dummy Connector example provides build files: `build.xml` and `build_rar.xml`. It also provides a properties file called `build.properties`. You can edit these files to match your build environment, which requires you to find out where all the required JAR files are located. The outcome of the build are these files:

— RAR file — The resource adapter archive that is packaged in a file named *MyResource*`Connector.rar`. It contains the complete code of the connector along with the connection parameter information.

— Schema JAR file — Includes the schema XML mapping file.

6 *Two-way connector only*
Code the file(s) that will comprise the agent. The resource type will dictate the programming language and APIs you use. Typically, if the agent is made up of multiple files, the files are bundled in a file called *connector*`Agent.zip` for Windows and *connector*`Agent.tar.gz` for UNIX. If an installer is also provided (for Windows), the files are bundled in a file called *connector*`Setup.zip`.

When the agent sends data to the Select Identity server, it must send a Service Provisioning Markup Language (SPML) compliant message. It must also send the data using SOAP. Refer to the Web Service help and guide for more information about the SPML-compliant message.

7 Define the deployment descriptor by creating an XML file called `ra.xml`. This file contains deployment specific information; you must specify the interface class names and implementation class names of the connector here. Here is an example of the "resourceadapter" section of the descriptor taken from LDAP connector:

```
<resourceadapter>
  <managedconnectionfactory-class>com.trulogica.truaccess.
   connector.ldap.ldapv3.LDAPManagedConnectionFactory
  </managedconnectionfactory-class>
  <connectionfactory-interface>com.trulogica.truaccess.
   connector.TAConnectorFactory
  </connectionfactory-interface>
  <connectionfactory-impl-class>com.trulogica.truaccess.
   connector.ldap.ldapv3.LDAPConnectorFactory
  </connectionfactory-impl-class>
```

```
  <connection-interface>com.trulogica.truaccess.connector.
   TAConnector
  </connection-interface>
  <connection-impl-class>com.trulogica.truaccess.connector.ldap.
   ldapv3.LDAPConnector
  </connection-impl-class>
  <transaction-support>NoTransaction</transaction-support>
  <reauthentication-support>false</reauthentication-support>
</resourceadapter>
```

Create this XML file according to the JCA specification.

8 If deploying the connector on WebLogic, create the WebLogic-specific deployment descriptor by creating a file called `weblogic-ra.xml`. You must register the JNDI name for the connector (`eis/`*`connector`*) here. The following is an example:

```
<weblogic-connection-factory-dd>
  <connection-factory-name>
    LDAPConnectorFactory
  </connection-factory-name>
  <jndi-name>eis/LDAPv3</jndi-name>
  <pool-params>
    <initial-capacity>0</initial-capacity>
  </pool-params>
</weblogic-connection-factory-dd>
```

9 If deploying the connector on WebSphere, create the `Manifest.mf` file that includes details on the classpath, which references all of the JAR files used by the connector. The following is an example script that creates the `Manifest.mf` file:

```
<jar basedir="${class.dir}" destfile="${class.dir}/
_connectorModule.jar" includes="META-INF" >
   <metainf dir="${basedir}/META-INF">
      <include name="*.xml"/>
   </metainf>
   <manifest >
      <attribute name="Built-by" value="${user.name}" />
      <attribute name="Class-Path" value="${depend.jars}" />
      <section name="${connector.name}">
         <attribute name="Implementation-Version"
          value="${connector.version} ${env.DATE}"/>
      </section>
   </manifest>
</jar>
```

10 If deploying on WebSphere, create a file called *_connector*Module.jar that contains the ra.xml and Manifest.mf files.

11 Bundle all other files in a RAR file called *connector*.rar ,as follows:

— If deploying on WebLogic:
Bundle all connector class files, library JAR files, the ra.xml file, and the weblogic-ra.xml file in the RAR file.

— If deploying on WebSphere:
Bundle all connector class files, library JAR files, and the *_connector*Module.jar file in the RAR file.

After completing these steps, typically the following files will be available for deployment:

- *connector*.rar — Contains the resource adapter (connector binary files)
- *connector*schema.jar — Contains the mapping file(s)
- *connector*Agent.zip or *connector*Setup.zip — Contains the agent files, if an agent is created

You can then install the connector on the Select Identity server. See Installing a Connector on page 75 for general steps; specific details will depend on how the connector was implemented. Then, deploy the connector in Select Identity, using the Select Identity console, as described in Configuring a Connector in Select Identity on page 77. Finally, to verify that the connector is installed correctly and connect to the resource, see Testing a Connector on page 78.

Requirements

You must have an understanding of the Java Developer Kit (JDK), version 1.4, and be familiar with the JCA, version 1.0. In addition, Select Identity provides a Connector API that is used with JCA to create connectors. You can download the JCA specification from the following page:

http://java.sun.com/j2ee/connector/download.html

Also, refer to **http://e-docs.bea.com/wls/docs81/jconnector/index.html** if you are creating a WebLogic connector:

For information about the J2EE APIs, including those for connectors, refer to **http://java.sun.com/j2ee/1.4/docs/api/index.html**.

When implementing a connector using the J2EE Connector APIs and the APIs described here, it is expected that the operations on the connector instances are called within transactions and from multiple threads. Also, the connectors must implement adequate synchronization to prevent data corruption.

For the development environment, the following tools are necessary:

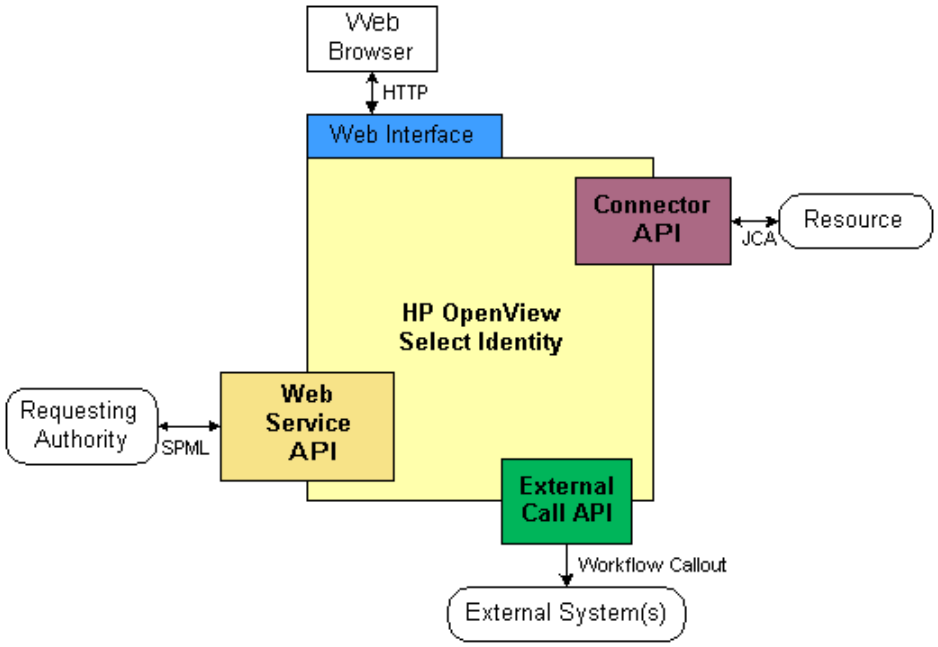
- Java Integrated Development Environment (IDE) — Any Java IDE supporting JDK 1.4.1 or later, such as Eclipse 3.0, is required.
- Build tool — It is recommended that you use Apache Ant 1.6 or later.

The following is a list of the required library JAR files (packages) that are used in the development process:

- Provided by Select Identity:
  - `connector.jar` — The Select Identity connector interface
  - `genConnectorImpl.jar` — Generic implementation of the JCA part of the connector and the **TAConnector** interface
  - `utils-log.jar` — Utility classes
  - `connectorimpl.jar` — Select Identity connector user model classes
- External packages:
  - `jakarta-regexp-1.2.jar`
  - `commons-beanutils.jar`
  - `commons-collections.jar`
  - `commons-logging.jar`

Overview of the Select Identity Connector API

The following diagram illustrates the Select Identity API architecture, showing the relationship of the Connector API to Select Identity and the other APIs:



The following classes and interfaces are provided by the Connector API. Online help (Javadoc) is provided for this API on the Select Identity CD, in the `docs/api_help/connectors/Javadoc` directory:

- **EntitySupport**

  Defines the actions that can be performed on an entity, which is an object that is managed by Select Identity, such as a user, group, role, or stage.

- **GroupModel**

  Represents an entitlement on a resource.

- **RelationSupport**

  Specifies an association between identity object types, such as between a user and entitlement and vice versa.

- **TAConnector** or **SIConnectorInterface**

  TAConnector is the top-level interface that maps identity information to a resource type. This is a generic connector interface that extends the JCA CCI Connection interface.

  SIConnectorInterface also maps identity information to a resource type, but it also helps you focus on the efforts involved in provisioning to the resource while avoiding the details of JCA and the Select Identity user model.

- **TAConnectorFactory**

  Creates instances of connections handles for resources. The connection handle is an implementation of TAConnector.

- **TAConnectorParamBean**

  Describes a configuration parameter needed by the connector. Examples of such parameters include URLs or configuration parameters like wait time. Select Identity retrieves a list of these beans to create a user interface to obtain values from the user.

- **TAConnectorParameterFactory**

  Obtains connection-specific beans that contain connection parameter values.

- **TAConnectorParamValueBean**

  An abstract class that represents the connection parameter values needed to establish a connection to a resource. It also contains all parameters needed to access the resource for user provisioning.

- **TAFilter**

  Enables you to issue search requests.

- **UserEntitySupport**

  Shows the level of support for user objects in the repository. In addition to supporting create, read, update, and delete tasks, UserEntitySupport specifies whether the password can be reset or changed in the resource.

- **UserModel**

  The interface of a class that contains information about the user that is being provisioned in or retrieved from a resource. All connectors must create a class that implements this interface.

Understanding the Resource Schema

The basic assumption is that the target resource for which you are building the connector supports users (or accounts). It may also support entitlements, which are the privileges or roles that can be assigned or unassigned to and from users. A “schema” refers to the definition of the users and their attributes and entitlements on the resource.

Before building the connector, you must determine what user attributes and entitlements are supported on on the resource. A user is defined by a set of attributes. For example, a user may be assigned an ID, an email address, a password, a physical (home or office) address, a social security number, an employee number, and so on. The resource may support and store attributes in one of many ways. Here are some examples:

- Physical attributes — The resource may support physical attributes that can be set with values. Resources that support physical attributes include LDAP servers and SQL databases. In this case, connector can directly assign the Select Identity attribute value to the resource attribute value.
- An abstraction of attributes — Some resources do not support physical attributes, such as UNIX and Windows systems. For these resources, the connector can define an intermediate attribute that is used to store the values defined by Select Identity.
- API — The resource may support an API to perform provisioning operations. Such resources include IBM Tivoli Access Manager and Netegrity SiteMinder. In this case, the connector must call the appropriate API method and pass the attribute value to the method.

“Entitlements” are defined as anything that can be assigned or unassigned to or from a user, and the result of this process authorizes or unauthorizes the user to perform certain operations on the resource. For example, if the Administrators entitlement is assigned to a user, the user has all the power on the system. You must determine if the resource supports entitlements.

Gathering Connector Parameters

Gather the following connector parameters before coding the connector. In particular, you will need some of the values if you implement the SIConnectorInterface interface. You must also create several properties files before building the connector. Collect the following parameters:

- Connection parameters — These parameters are required to establish a connection and perform all provisioning operations on the resource. Store these parameters in the *MyResource*`ParamResources.properties` file along with the following details about each connection parameter:
  - Name — Name of the parameter
  - displayName — Display name of the connection parameter
  - defaultValue — Default value of the parameter
  - helpString — Help text for the parameter that is displayed on the Select Identity console
  - minLength — Minimim length of the value of this parameter
  - maxLength — Maximum length of the value
  - pattern — Regular expression of the value pattern
  - required — Whether this parameter is required; specify true or false
  - tipString — Not used currently
  - type — The parameter type (typically java.lang.String)
  - encryptValue — Whether the value is encrypted; this is typically set to true for password parameters and false for all others

  Here is an example of the file:

```
hostName-displayName=Host
hostName-defaultValue=16.73.17.100
hostName-helpString=Host name or IP address of the server
hostName-minLength=1
hostName-maxLength=255
hostName-pattern=[.]+
hostName-required=true
hostName-tipString=Host name/IP of the Server
hostName-type=java.lang.String
hostName-encryptValue=false
```

- Resource adapter parameters — These parameters define the properties of the JCA resource adapter. This information is stored in a file called `ra.xml` that is packaged as part of the RAR, which is the deployment descriptor of the connector being developed. Here are some of the parameters:

  Edit the `META-INF/ra.xml` file provided in the ZIP file in the `docs/api_help/connectors/dummyConnector` directory on the Select Identity CD and change the following parameters. Let the others remain the same.

  — display-name — Display name of the connector

  — vendor-name — Name of the developer of this connector

  — spec-version — Version of the JCA specification (should be 1.0)

  — eis-type — Type of the resource

  — version — Resource version

  — Configuration properties — Some of the properties are fixed, as in the sample `ra.xml` file. The following are the configuration properties that must be changed:

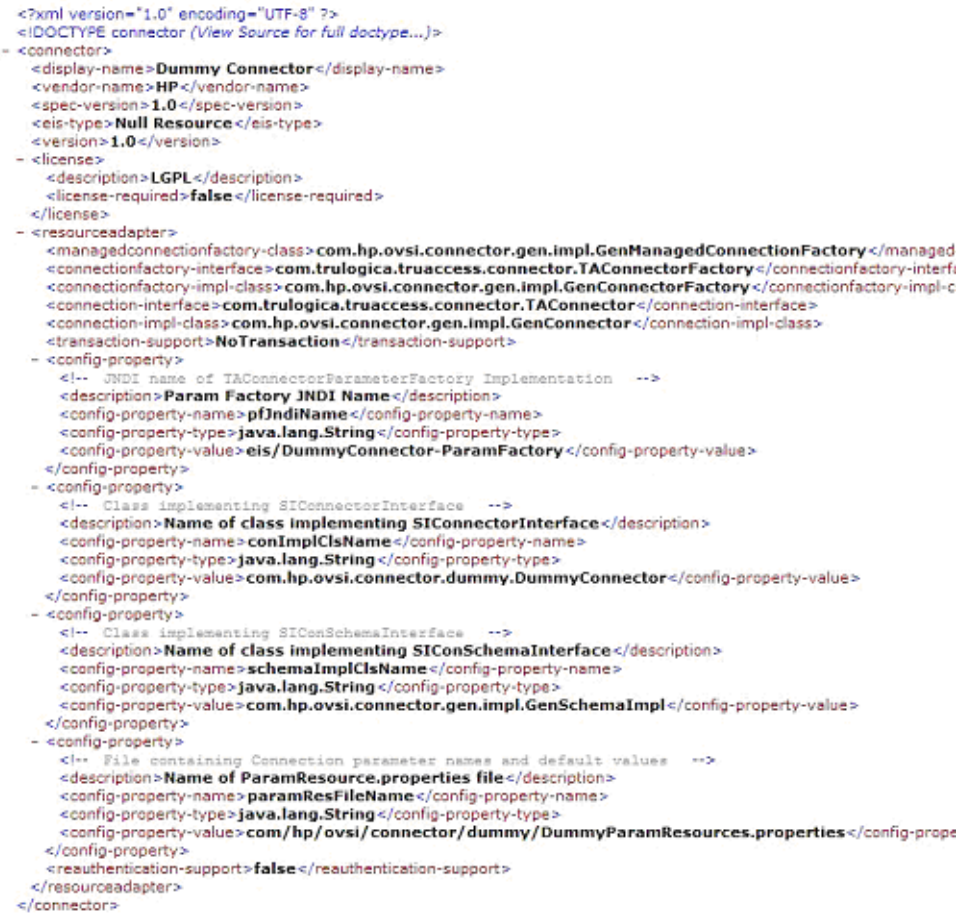
    – pfJndiName — The JNDI name of the connector parameter factory, which is usually derived from the name of the connector, prefixed with **eis/**, and suffixed with **-ParamFactory**, as in this example:
    eis/MyResourceConnector-ParamFactory

    – conImplClsName — The name of the class that implements **SIConnectorInterface**, such as x.y.z.MyResourceConnector (this implies there is a file called `MyResourceConnector.java` implementing the interface); also, see `DummyConnector.java` that is provided in the Dummy Connector code example

    – paramResFileName — The name of the file that stores the connection parameters, such as `x.y.z.MyResourceParamResources.properties`; for an example, see `com/hp/ovsi/connector/dummy/DummyParamResources.properties` provided with the Dummy Connector example

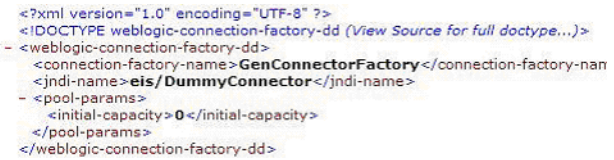
    – jndi-name — The JNDI name of the connector factory, such as eis/MyResourceConnector; make sure this is the same as the first part of the value of the pfJndiName property

Here is a snapshot of an example `ra.xml` file:



```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE connector (View Source for full doctype...)>
- <connector>
    <display-name>Dummy Connector</display-name>
    <vendor-name>HP</vendor-name>
    <spec-version>1.0</spec-version>
    <eis-type>Null Resource</eis-type>
    <version>1.0</version>
  - <license>
      <description>LGPL</description>
      <license-required>false</license-required>
    </license>
  - <resourceadapter>
      <managedconnectionfactory-class>com.hp.ovsi.connector.gen.impl.GenManagedConnectionFactory</managed
      <connectionfactory-interface>com.trulogica.truaccess.connector.TAConnectorFactory</connectionfactory-interf
      <connectionfactory-impl-class>com.hp.ovsi.connector.gen.impl.GenConnectorFactory</connectionfactory-impl-c
      <connection-interface>com.trulogica.truaccess.connector.TAConnector</connection-interface>
      <connection-impl-class>com.hp.ovsi.connector.gen.impl.GenConnector</connection-impl-class>
      <transaction-support>NoTransaction</transaction-support>
    - <config-property>
        <!-- JNDI name of TAConnectorParameterFactory Implementation -->
        <description>Param Factory JNDI Name</description>
        <config-property-name>pfJndiName</config-property-name>
        <config-property-type>java.lang.String</config-property-type>
        <config-property-value>eis/DummyConnector-ParamFactory</config-property-value>
      </config-property>
    - <config-property>
        <!-- Class implementing SIConnectorInterface -->
        <description>Name of class implementing SIConnectorInterface</description>
        <config-property-name>conImplClsName</config-property-name>
        <config-property-type>java.lang.String</config-property-type>
        <config-property-value>com.hp.ovsi.connector.dummy.DummyConnector</config-property-value>
      </config-property>
    - <config-property>
        <!-- Class implementing SIConSchemaInterface -->
        <description>Name of class implementing SIConSchemaInterface</description>
        <config-property-name>schemaImplClsName</config-property-name>
        <config-property-type>java.lang.String</config-property-type>
        <config-property-value>com.hp.ovsi.connector.gen.impl.GenSchemaImpl</config-property-value>
      </config-property>
    - <config-property>
        <!-- File containing Connection parameter names and default values -->
        <description>Name of ParamResource.properties file</description>
        <config-property-name>paramResFileName</config-property-name>
        <config-property-type>java.lang.String</config-property-type>
        <config-property-value>com/hp/ovsi/connector/dummy/DummyParamResources.properties</config-prope
      </config-property>
      <reauthentication-support>false</reauthentication-support>
    </resourceadapter>
</connector>
```

You can also refer to the `weblogic-ra.xml` file that contains information about the Dummy Connector if it is deployed in BEA WebLogic application server. Here is a snapshot of this file:



```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE weblogic-connection-factory-dd (View Source for full doctype...)>
- <weblogic-connection-factory-dd>
    <connection-factory-name>GenConnectorFactory</connection-factory-nan
    <jndi-name>eis/DummyConnector</jndi-name>
  - <pool-params>
      <initial-capacity>0</initial-capacity>
    </pool-params>
</weblogic-connection-factory-dd>
```

Coding the Connector

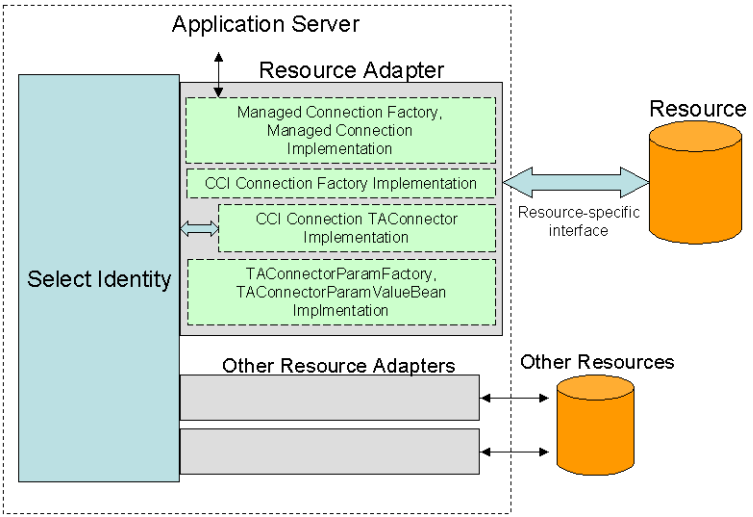
Code the Java classes and implement the interfaces that will comprise the connector. This entails implementing JCA and Select Identity Connector APIs. You must also register the parameter factory implementation with JNDI. See Interface, Class, and Method Implementations on page 31 for details.

If you are implementing a one-way connector (no agent), it may be possible to code a change detection utility that will track changes made on the resource, create a “differences” file that contains the changed data, and send the file to the Select Identity server. The ability to create this utility is dependant on the tools available on the resource. A code example is provided in the ZIP file in the `docs/api_help/connectors/LDAPv3_v3` directory on the Select Identity CD. When unzipped, the source resides in the `com/trulogica/truaccess/agent` directory. This change detection utility is provided for the iPlanet LDAP connector.

If you are implementing a two-way connector, see Implementation of Reverse Synchronization on page 44 for information about implementing reverse synchronization.

Interface, Class, and Method Implementations

The following illustrates the connector architecture and the relationship between connector classes, Select Identity, and the resource:



The following provides guidelines that you must follow when coding the connector.

JCA Interfaces

Implement the JCA interfaces; refer to the J2EE specification for details. Connectors must support both local and distributed transaction protocols. If a resource does not support transactions, the adapter must record transactions so that compensating transactions can be applied on rollback. Regarding security, JAAS is used to authenticate inbound communication.

Here are the interfaces you must implement:

- java.resource.cci.Connector extended by TAConnector
- javax.resource.spi.ManagedConnection
- javax.resource.spi.ManagedConnectionFactory (implement only one instance of this interface)
- javax.resource.spi.ManagedConnectionMetaData

Select Identity Connector API Interfaces and Classes

Implement the following Select Identity Connector API interfaces and classes. Refer to the API online help (on the Select Identity CD, in the `docs/api_help/connectors/Javadoc` directory) for more information.

- TAConnector interface — the main interface to implement. Select Identity calls on the methods in this interface to perform provisioning operations. In particular, you *must* implement these methods:

Method	Comment
add(UserModel)	**To implement:** Build attributes by taking the values from UserModel and creating a user on the resource. Consider that the user may exist on the resource, such as if the user exists in another Service using the same resource or if the user was created by an application other than Select Identity. Thus, if the user exists, modify the user (for example, using update(UserModel)) and return the user ID to Select Identity, as you would for a successful add request. After a succesfully adding the user, set the key field value by calling UserModel.setUserId(). Also, you can call UserModel.set() to return user attributes to Select Identity after the connector sets them on the resource. **Usage:** add(UserModel) is called to provision a new user on the resource.
changePassword(UserModel)	**To implement:** Use the new password in UserModel to update the password on the resource. **Usage:** This method is called to change a user's password.
expirePassword(UserModel, Boolean)	**Usage:** This method expires a password for the specified user.

Method	Comment
get(UserModel)	**To implement:** The connector should construct the key field value of the user on the resource and check the existence of user in the resource. If the user does not exist in the resource, the connector should throw ObjectNotFoundException. **Usage:** This method is called to verify the existence of user in the resource.
getGroupAttributes()	**To implement:** The implementation can return one attribute to represent the ID of the entitlement. **Usage:** This method must return the schema supported by the resource for group and entitlement provisioning.
getGroups()	**To implement:** Return all entitlements in the resource without using the filter (same as getGroups(TAFilter)).
getGroups(TAFilter)	**To implement:** Build and return a set of Strings that identify the entitlement. Using TAFilter, you can filter out the list returned. **Usage:** This is called by Select Identity to get a list of all entitlements on the resource. TAFilter is used to filter out the entitlements retrieved from the resouce and is passed in from the filter.

Method	Comment
getUserAttributes()	**To implement:** Return a list of TAConnectorParamBean instances that contain details about each attribute supported by the connector. These attributes are the Select Identity resource attributes, not the attributes on the physical resource. The mapping between the Select Identity resource attributes and the physical resource attributes is done by the connector. For example, the LDAP connector uses an XML mapping file to map attributes. In the mapping file, "tafield" is the Select Identity resource attribute and "resfield" is the physical resource attribute. **Usage:** Select Identity calls this method to get the schema supported by the connector for the user object.
isPasswordValid(password)	**To implement:** Check the validity of the password on the resource. **Usage:** This method is called before adding a new user or resetting the password of an existing user.
isSupported(entity1)	**To implement:** Return the support for UserModel and GroupModel. GroupModel is a generic container that represents any type of entitlement on the resource. Examples include user groups, access control levels, privileges, roles, and so on. **Usage:** This method is called on the connector to get the level of support for the object.
isSupported(entity1, entity2)	**Usage:** This method is called to get the level of support for the association of entitlements to users.

Method	Comment
link(UserModel, GroupModel)	**To implement:** Select Identity first makes a call to getGroups(TAFilter) to get a list of all entitlements on the resource. Then, GroupModel passed in references to the entitlements returned by getGroups(). Call getGroupId() to identify the entitlement and use it as the key to associate a user with the entitlement on the resource. You can also call UserModel.set() to return attributes to Select Identity after the connector sets them on the resource. When adding entitlements to a user on a resource, consider that the user may have the entitlement you are attempting to add, such as if another Service added the entitlement. Thus, if the entitlement exists, do not throw an exception. **Usage:** This method is used to assign entitlements to an existing user. Select Identity calls this method once for each entitlement to be assigned.
remove(UserModel)	**To implement:** Delete the user account on the resource. When removing a user from a resource, consider that the user have been removed by another Service or application. Thus, if the user has been removed, do not throw an exception. **Usage:** Call this method to remove a user from a resource.
resetPassword(UserModel)	**To implement:** Use the new password in UserModel to update the password on the resource. **Usage:** Call this to reset a user password.

Method	Comment
setStatus(UserModel, int)	**To implement:** Depending on the resource support that is appropriate, the implementation could disable or enable the user on the resource. For example, on a resource that does not support this feature, you can set an attribute to reflect that the user is disabled or enabled. Also, you can call UserModel.set() to return user attributes to Select Identity after the connector sets them on the resource. If the connector implementation does not support this operation, throw NotImplementedException. When the connector throws this exception, Select Identity will call back on the connector to add or delete all of the entitlements on the user (using the link or unlink methods). This feature is provided to support those resources where the meaning of disable or enable is to remove or add entitlements of the user, respectively. **Usage:** This method is called when user is being disabled or enabled for all Services.

Method	Comment
test()	**To implement:** This method tests the connectivity of the physical resource using the connection parameter values in TAConnectorParamValueBean. This method can also implement the logic that validates the connection parameters. **Usage:** Select Identity calls this method on the connector when a new resource is deployed or an existing resource is modified. Select Identityexpects the connector to verify connectivity with the resource and validate the connection parameters.
unlink(UserModel, GroupModel)	**To implement:** Disassociate the user from the entitlement referred to by GroupModel.getGroupId(). You can also call UserModel.set() to return user attributes to Select Identity after the connector sets them on the resource. When removing entitlements from a user on a resource, consider that the entitlement may have been removed by another Service or application. Thus, if the entitlement does not exist, do not throw an exception. **Usage:** This is called when a user's Service membership is modified or when user is disabled for a given Service.
update(UserModel)	**To implement:** Update the attributes of the user on the resource. You can also call UserModel.set() to return user attributes to Select Identity after the connector sets them on the resource. **Usage:** This method is called to update the attributes of a user. Select Identity sends all attributes of the user.

- SIConnectorInterface interface — Instead of implementing TAConnector, you can implement SIConnectorInterface. If you implement this interface, Select Identity calls on the methods in this interface to perform provisioning operations. In particular, you *must* implement these method.

Method	**Comment**
setBean(TAConnectorParam ValueBean)	Stores all the connection parameters and their values. This method must save this bean for use during the other methods.
openSession()	Keeps a session open with the resource, which can then be put in this instance.
closeSession()	Closes the session, if opened earlier.
setSchema(TASchema)	Copies of the XML mapping schema.
doTest()	Uses the connection parameter information in TAConnectorParamValueBean to check if the resource is up and accessible. It also validates the connection parameters. This is called when the resource is deployed in Select Identity and the **Test and Submit** button is clicked. It is also called when the resource is modified.

Method	Comment
createUser (keyField, keyValue, attrList)	Creates a new user in Select Identity. keyField refers to the concero:resfield value in the mapping file that has the concero:isKey set to "true". keyValue is the value of the above keyField. attrList contains a name/value pair attribute information to be pushed when a user is created. Only mappings in the mapping file that are marked with concero:init="true" will be part of attrList. Use the keyValue and the attributes to create a new user in resource. If the user exists, catch the appropriate exception and perform the modify operation by calling the updateUser() method. The identifier of the user in the resource may be derived from one or more of the attribute values passed or the resource may return the actual ID of the user. This must be returned at the end of this method. Successive calls on the connector will have this as the keyValue.
isUserExists (keyField, keyValue, keyExistsFlg)	Verifies if the user exists. This is called when Select Identity wants to check if a user exists in the resource. Use the keyValue to see if the user is exists. keyExistsFlg will be false if this is called for the first time on a new user. If this is not the first time, this flag will be true.
findUser (keyField, keyValue, attrList, keyExistsFlg)	Gets the details of the user from the resource and populates all attributes in the attrList map.

Method	Comment
updateUser (keyField, keyValue, attrList)	Updates the user's attributes in the resource with the new ones given in the attrList map. Select Identity passes all user attributes, even if they are not changed. Therefore, make sure you handle any errors thrown by the resource if replacing with the same value.
deleteUser (keyField, keyValue)	Deletes a user.
setUserStatus (keyField, keyValue, attrList, status)	Sets the user's status. This is called when all Services are disabled or enabled for a user in Select Identity. Use the keyField and keyValue to disable or enable the user. This might mean different things on different resources; it might mean blocking the user from access, temporarily terminating his account, and so on.
isPasswordValid (passwd)	Validates a password.
expirePassword(keyField, keyValue, flg)	Expires the user's password if flg is true and unexpire on false. Expiring a password might mean that the user will not be able to use his existing password to access the resource.
resetPassword(keyField, keyValue, passwd)	Changes the password of the user to the new one specified in passwd.
getAllUsers(keyField, idList)	Gets the list of IDs of all users and populates the idList collection.
getUsers(userKeyField, groupKeyField, groupKeyValue, idList)	Retrieves a list of users.

Method	Comment
getAllEntitlements(keyField, TAFilter, idList)	Retrieves the IDs of all entitlements present in the resource. Select Identity will use this to link or unlink to or from the user to assign or unassign. The entitlement filter contains criteria to match the result entitlements. If null, it returns all entitlements. If not null, and filter.getName() is populated, this might mean a Search Connector operation is defined on one of the user attributes. In this case, it returns all possible values for this attribute in the resource. If filter.getValue() is not null, use this as the matching criteria along with filter.getOperation(), which could be EQUALITY, BEGINS_WITH, ENDS_WITH, or CONTAINS. For example, if filter.getValue() is "AA" and filter.getOperation() is TAFilter.BEGINS_WITH, this returns all values that start with the string "AA".
getEntitlements(keyField, keyValue, idList)	Retrieves a list of entitlements.

Method	Comment
link(userKeyField, userKeyValue, groupKeyField, groupKeyValue)	Assigns a user to one entitlement. You can perform a two-way or one-way assignment depending on resource support. For example, you could add a user to a members field of an entitlement or add an entitlement as a memberOf field of the user. The groupKeyField parameter is the keyField as defined in the XML mapping file for the objectClassDefinition of the group. The groupKeyValue parameters is the ID of the entitlement as returned by the connector in getAllEntitlements() method.
unlink(userKeyField, userKeyValue, groupKeyField, groupKeyValue)	Unassigns a user from one entitlement.

- TAConnectorFactory interface — creates instances of connection handles for the connector. The connection handle is an implementation of the TAConnector interface. For the TAConnectorFactory interface, you must implement the following method:

Method	Comment
getConnection(connParam)	**Usage:** This is called to return the implementation of the TAConnector interface.

- TAConnectorParameterFactory interface — obtains connector-specific beans that hold connection parameter values. In particular, you must implement these methods:

Method	Comment
createParamValueBean()	**Usage:** This is called to create a bean to pass to the parameter values.
getParamBeans()	**Usage:** This is called to return a collection of TAConnectorParamBean classes if the connector needs configuration values from the user.

- TAConnectorParamValueBean class — an abstract class that represetns the connection parameter values needed to establish a connection with the resource. It also holds all parameters needed to access the resource for user provisioning. In particular, you must implement these methods:

Method	Comment
getTAInstallDirectory()	**Usage:** This is called to return the path of the Select Identity installation directory.
setTAInstallDirectory(path)	**Usage:** This is called to set the path of the Select Identity installation directory.
getParamNames	**To implement:** Return all connection parameter names. **Usage:** This method returns the connection parameter namess that are used to establish a connection with the resource.

Method	Comment
get	**To implement:** Return the value of the connector parameter. **Usage:** This gets the value of the connection parameter.
set	**To implement:** Save the value of the connector parameter. **Usage:** The set method sets the connection parameter value. Select Identity will pass the values provided when the resource was deployed using this method. The bean implementation should store the value for later use.

You can also implement the following Select Identity Connector API interfaces and classes (this is a subset of all interfaces and classes provided by the API), as needed:

- UserModel
- GroupModel
- EntitySupport
- RelationSupport

Finally, implement an authentication mechanism for the connection. For example, you may implement simple username/password authentication using an administrative account that has the necessary authority.

Implementation of Reverse Synchronization

The Select Identity Web Service listens for reverse synchronization requests from resources. These requests are sent as SPML SOAP messages. The messages propagate user changes that were made on the resource to Select

Identity. The following events are captured on the resource and a corresponding SPML request must be sent to Select Identity:

- Adding a user
  A new user is added on the resource. To propogate this change back to Select Identity, an SPML <addRequest> request must be sent that includes all of the user's attributes.
- Changing user attributes
  User attributes are modified on the resource. An SPML <modifyRequest> request must be sent to the Select Identity server to synchronize these changes.
- Adding entitlement to a user or removing entitlements from a user
  Entitlements are associated or disassociated with an existing user on the resource. An SPML <modifyRequest> request must be sent with the new entitlements added or removed.
- Changing a user's password
  A user's password is changed or reset on the resource. An SPML <extendedRequest> request must be sent containing the new password.
- Deleting a user
  A user is deleted from the resource. An SPML <deleteRequest> request must be sent for the deleted user.
- Enabling or disabling a user
  A user is enabled or disabled on the resource. A SPML <modifyRequest> request containing all of the user attributes must be sent to propogate the change(s) to Select Identity.

How the changes are captured and how the SPML request is generated are resource specific. Each generated SPML request is parsed by Select Identity using an XSL file that corresponds to the XML mapping file that enables Select Identity to push data to the resource.

The SPML request that is generated for reverse synchronization includes the following information:

- Operational attributes — Relate to the properties of the Select Identity instance to which the reverse synchronization request is being sent.
- Resource attributes — Define user attributes on the resource.

The following is an example of the operational attributes section of an SPML request:

```
<operationalAttributes>
  <attr name='urn:trulogica:concero:2.0#reverseSync'>
    <value>true</value>
  </attr>
  <attr name='urn:trulogica:concero:2.0#resourceId'>
    <value>AD</value>
  </attr>
  <attr name='urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName'>
    <value>sisa</value>
  </attr>
  <attr name='urn:trulogica:concero:2.0#resourceType'>
    <value>activedirectory</value>
  </attr>
  <attr name='urn:trulogica:concero:2.0#password'>
    <value>abc123</value>
  </attr>
</operationalAttributes>
```

The <attr> elements in this block are as follows:

urn:trulogica:concero:2.0#reverseSync

Whether this request is a reverse synchronization request. The value is a boolean set to **`true`** if the request is a reverse synchronization request.

urn:trulogica:concero:2.0#resourceId

The name of the resource (in Select Identity) to which this request is sent.

urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName

The username of an administrative user in Select Identity.

urn:trulogica:concero:2.0#password

The password of the administrative user.

urn:trulogica:concero:2.0#resourceType

The name of the XSL file (without the .xsl extension) that is associated with the resource and that parses the reverse synchronization request.

SPML Request Examples

The following are SPML examples that were generated for each type of user change on the resource:

- Adding a new user (no entitlements)

```
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/
  envelope/'>
<soap:Body>
  <addRequest xmlns='urn:oasis:names:tc:SPML:1:0'
   requestID='1'
   execution='urn:oasis:names:tc:SPML:1:0#asynchronous'>
    <operationalAttributes>
      <attr name='urn:trulogica:concero:2.0#reverseSync'>
        <value>true</value>
      </attr>
      <attr name='urn:trulogica:concero:2.0#resourceId'>
        <value>AD</value>
      </attr>
      <attr
      name='urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName'>
        <value>sisa</value>
      </attr>
      <attr name='urn:trulogica:concero:2.0#resourceType'>
        <value>activedirectory</value>
      </attr>
      <attr name='urn:trulogica:concero:2.0#password'>
        <value>abc123</value>
      </attr>
    </operationalAttributes>
    <attributes>
      <attr name='UserName'>
        <value>rvcs2002</value>
      </attr>
      <attr name='FirstName'>
        <value>Test User</value>
      </attr>
      <attr name='LastName'>
        <value>Last name</value>
      </attr>
      <attr name='mail' >
        <value>asdf@adf.com</value>
      </attr>
    </attributes>
```

```
  </addRequest>
</soap:Body>
</soap:Envelope>
```

- Adding a new user (with multiple entitlements)

```
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/
  envelope/'>
<soap:Body>
  <addRequest xmlns='urn:oasis:names:tc:SPML:1:0' requestID='1'
   execution='urn:oasis:names:tc:SPML:1:0#asynchronous'>
    <operationalAttributes>
      <attr name='urn:trulogica:concero:2.0#reverseSync'>
        <value>true</value>
      </attr>
      <attr name='urn:trulogica:concero:2.0#resourceId'>
        <value>AD</value>
      </attr>
      <attr
       name='urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName'>
        <value>sisa</value>
      </attr>
      <attr name='urn:trulogica:concero:2.0#resourceType'>
        <value>activedirectory</value>
      </attr>
      <attr name='urn:trulogica:concero:2.0#password'>
        <value>abc123</value>
      </attr>
    </operationalAttributes>
    <attributes>
      <attr name='UserName'>
        <value>rvcs2002</value>
      </attr>
      <attr name='FirstName'>
        <value>Test User</value>
      </attr>
      <attr name='LastName'>
        <value>Last name</value>
      </attr>
      <attr name='mail' >
        <value>asdf@adf.com</value>
      </attr>
      <attr name='urn:trulogica:concero:2.0#groups'
       operation='add'>
        <value>Administrator</value>
        <value>Guest</value>
```

```
      <value>Backup Operator</value>
     </attr>
    </attributes>
  </addRequest>
</soap:Body>
</soap:Envelope>
```

- Modifying a user by adding new entitlements

```
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/
  envelope/'>
<soap:Body>
  <modifyRequest xmlns:dsml="urn:oasis:names:tc:DSML:2:0:core"
   xmlns:spml="urn:oasis:names:tc:SPML:1:0"
   xmlns="urn:oasis:names:tc:SPML:1:0" requestID="1"
   execution="urn:oasis:names:tc:SPML:1:0#asynchronous">
    <operationalAttributes xmlns="">
      <attr name="urn:trulogica:concero:2.0#reverseSync">
        <value>true</value>
      </attr>
      <attr name="urn:trulogica:concero:2.0#resourceId">
        <value>AD</value>
      </attr>
      <attr
       name="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
        <value>sisa</value>
      </attr>
      <attr name="urn:trulogica:concero:2.0#resourceType">
        <value>activedirectory</value>
      </attr>
      <attr name="urn:trulogica:concero:2.0#password">
        <value>abc123</value>
      </attr>
      <attr name="urn:trulogica:concero:2.0#keyFields">
        <value>UserId</value>
      </attr>
    </operationalAttributes>
    <identifier xmlns=""
     type="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
      <id>rvcs2002</id>
    </identifier>
    <modifications xmlns="">
      <modification name="urn:trulogica:concero:2.0#groups"
       operation="add">
        <value>Replicator</value>
        <value>Guest</value>
```

```
      </modification>
    </modifications>
  </modifyRequest>
</soap:Body>
</soap:Envelope>
```

- Modifying a user by deleting entitlements

```
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/
  envelope/'>
<soap:Body>
  <modifyRequest xmlns:dsml="urn:oasis:names:tc:DSML:2:0:core"
   xmlns:spml="urn:oasis:names:tc:SPML:1:0"
   xmlns="urn:oasis:names:tc:SPML:1:0" requestID="1"
   execution="urn:oasis:names:tc:SPML:1:0#asynchronous">
    <operationalAttributes xmlns="">
      <attr name="urn:trulogica:concero:2.0#reverseSync">
        <value>true</value>
      </attr>
      <attr name="urn:trulogica:concero:2.0#resourceId">
        <value>AD</value>
      </attr>
      <attr
       name="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
        <value>sisa</value>
      </attr>
      <attr name="urn:trulogica:concero:2.0#resourceType">
        <value>activedirectory</value>
      </attr>
      <attr name="urn:trulogica:concero:2.0#password">
        <value>abc123</value>
      </attr>
      <attr name="urn:trulogica:concero:2.0#keyFields">
        <value>UserId</value>
      </attr>
    </operationalAttributes>
    <identifier xmlns=""
     type="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
      <id>rvcs2002</id>
    </identifier>
    <modifications xmlns="">
      <modification name="urn:trulogica:concero:2.0#groups"
       operation="delete">
        <value>Replicator</value>
        <value>Guest</value>
      </modification>
```

```
    </modifications>
  </modifyRequest>
</soap:Body>
</soap:Envelope>
```

- Modifying a user by adding and deleting entitlements

```
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/
  envelope/'>
<soap:Body>
  <modifyRequest xmlns:dsml="urn:oasis:names:tc:DSML:2:0:core"
   xmlns:spml="urn:oasis:names:tc:SPML:1:0"
   xmlns="urn:oasis:names:tc:SPML:1:0" requestID="1"
   execution="urn:oasis:names:tc:SPML:1:0#asynchronous">
    <operationalAttributes xmlns="">
      <attr name="urn:trulogica:concero:2.0#reverseSync">
        <value>true</value>
      </attr>
      <attr name="urn:trulogica:concero:2.0#resourceId">
        <value>AD</value>
      </attr>
      <attr
       name="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
        <value>sisa</value>
      </attr>
      <attr name="urn:trulogica:concero:2.0#resourceType">
        <value>activedirectory</value>
      </attr>
      <attr name="urn:trulogica:concero:2.0#password">
        <value>abc123</value>
      </attr>
      <attr name="urn:trulogica:concero:2.0#keyFields">
        <value>UserId</value>
      </attr>
    </operationalAttributes>
    <identifier xmlns=""
     type="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
      <id>rvcs2002</id>
    </identifier>
    <modifications xmlns="">
      <modification name="urn:trulogica:concero:2.0#groups"
       operation="add">
        <value>Replicator</value>
        <value>Guest</value>
      </modification>
      <modification name="urn:trulogica:concero:2.0#groups"
```

```
      operation="delete">
       <value>Administrator</value>
       <value>Backup Operator</value>
     </modification>
   </modifications>
  </modifyRequest>
</soap:Body>
</soap:Envelope>
```

- Deleting a user

```
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/
  envelope/'>
<soap:Body>
  <deleteRequest requestID='12345'
   execution='urn:oasis:names:tc:SPML:1:0#asynchronous'>
    <operationalAttributes>
      <attr
       name='urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName'>
        <value>sisa</value>
      </attr>
      <attr name='urn:trulogica:concero:2.0#password'>
        <value>abc123</value>
      </attr>
      <attr name='urn:trulogica:concero:2.0#resourceId'>
        <value>AD</value>
      </attr>
      <attr name='urn:trulogica:concero:2.0#reverseSync'>
        <value>true</value>
      </attr>
      <attr name='urn:trulogica:concero:2.0#resourceType'>
        <value>activedirectory</value>
      </attr>
    </operationalAttributes>
    <identifier
     type='urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName'>
      <id>pk22</id>
    </identifier>
  </deleteRequest>
</soap:Body>
</soap:Envelope>
```

- Changing (resetting) a user password

```
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/
  envelope/'>
<soap:Body>
```

```
<extendedRequest requestID='1769'
 execution='urn:oasis:names:tc:SPML:1:0#synchronous'>
  <operationalAttributes>
    <attr
   name='urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName'>
    <value>sisa</value>
  </attr>
  <attr name='urn:trulogica:concero:2.0#password'>
    <value>abc123</value>
  </attr>
  <!-- Name of deployed resource in SI for this machine -->
  <attr name='urn:trulogica:concero:2.0#resourceId'>
    <value>AD</value>
  </attr>
  <attr name='urn:trulogica:concero:2.0#reverseSync'>
    <value>true</value>
  </attr>
</operationalAttributes>
<identifier
 type='urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName'>
  <id>pk27_7</id>
</identifier>
<providerIdentifier
 providerIDType='urn:oasis:names:tc:SPML:1:0#URN'>
  <providerID>urn:trulogica:concero:2.0</providerID>
</providerIdentifier>
<operationIdentifier
 operationIDType='urn:oasis:names:tc:SPML:1:0#URN'>
  <operationID>urn:trulogica:concero:2.0#changePassword
  </operationID>
</operationIdentifier>
<attributes>
  <attr name='urn:trulogica:concero:2.0#newPassword'>
    <value>Welcome1</value>
  </attr>
</attributes>
</extendedRequest>
</soap:Body>
</soap:Envelope>
```

XSL File for Parsing Reverse Synchronization SPML

The SPML request received by Select Identity contains resource-specific attribute names. These must be converted to the attribute names defined by the resource (XML) mapping file. To do this, the SPML request is parsed

using an XSL file, which must be provided with the connector. Any change to attribute names in the XML mapping file must be propagated to the XSL file for the reverse synchronization to work correctly.

The XSL file given below is an XSL translator template, followed by an explanation. Use this template for generating your XSL files.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!-- Reverse Synchronization for Select Identity
       Translator template for SPML messages received
       from physical resources
  -->

  <!-- ************************************************************
    Reverse Mapper for SPML sync messages from agents on Physical
    resource
    ************************************************************
    NOTE: This file depends on the mappings in the mapping XML file
    used by the connector and if the mapping XML changes, this XSL
    needs to be updated
    ************************************************************
  -->

  <xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:spml='urn:oasis:names:tc:SPML:1:0'
   xmlns:dsml='urn:oasis:names:tc:DSML:2:0:core' >

    <!-- Initial password for new users/if not coming in from
     resource -->
    <xsl:variable name="DEFAULT_PASSWORD" select="'abc123'"/>

    <!-- Physical resource user id and password attribute names.
     Replace USERID with the user id attribute name from
     physical resource as defined in the mapping file -->
    <xsl:variable name="RES_USERID" select="'USERID'"/>

    <!-- Replace PASSWORD with the password attribute name from
     physical resource as defined in the mapping file -->
    <xsl:variable name="RES_PASSWORD" select="'PASSWORD'"/>

    <!-- User ID and Password SI Attribute names
     Replace UserId with the user id attribute name as in SI -->
    <xsl:variable name="SI_USERID" select="'UserId'"/>

    <!-- Replace Password with the password attribute name as
     mapped onto SI -->
    <xsl:variable name="SI_PASSWORD" select="'Password'"/>
```

```
<!-- ****************************************
 Attribute name mappings
 ****************************************
 RES_ATTR : Name of the attribute on the physical resource  -
 resfield in forward xml (lowercase)
 SI_ATTR : Name of SI resource attribute - tafield in forward
 xml
 The AttributeMapper below maps RES_ATTRxx -> SI_ATTRxx
 ****************************************
 -->
<!-- Example mappings ($RES_ATTR0 and $RES_ATTR1). Replace
 xxxxxx with the attribute names
 Add more as needed and add them in the AttributeMapper
 template -->
<xsl:variable name="RES_ATTR0" select="'xxxxxxxxxx'"/>
<xsl:variable name="SI_ATTR0" select="'xxxxxxxxxx'"/>
<xsl:variable name="RES_ATTR1" select="'xxxxxxxxxxx'"/>
<xsl:variable name="SI_ATTR1" select="'xxxxxxxxxxx'"/>
<!-- ************************************************************
 There generally should not be any change in this file below
 this line
 ************************************************************
 -->
<!-- ************************************
 Handler for addRequest elements
 ************************************
 -->
<xsl:template name="AddRequestHandler" match="spml:addRequest">
  <xsl:text>
  </xsl:text>
  <xsl:element name="addRequest"
   namespace="urn:oasis:names:tc:SPML:1:0">
    <!-- Add the two attributes -->
    <xsl:attribute name="requestID">
      <xsl:value-of select="'1'" />
      <!-- <xsl:value-of select="./@requestID" /> has some
       problem -->
    </xsl:attribute>
    <xsl:attribute name="execution">
      <xsl:value-of
       select="'urn:oasis:names:tc:SPML:1:0#asynchronous'" />
      <!-- <xsl:value-of select="./@execution" /> has some
       problem -->
    </xsl:attribute>
```

```
<xsl:text>
</xsl:text>
<xsl:call-template name="OperationalAttrHandler">
  <xsl:with-param name="ADDREQFLAG" select="'true'"/>
</xsl:call-template>
<xsl:element name="attributes">
  <xsl:text>
  </xsl:text>
  <!-- Convert the attribute names -->
  <xsl:for-each select="spml:attributes/spml:attr">
    <xsl:choose>
      <xsl:when
       test="@name='urn:trulogica:concero:2.0#groups'">
        <xsl:element name="attr" >
          <xsl:attribute name="name">
            <xsl:value-of select="@name" />
          </xsl:attribute>
          <xsl:text>
          </xsl:text>
          <xsl:for-each select="spml:value">
            <xsl:element name="value" >
              <xsl:value-of select="." />
            </xsl:element>
            <xsl:text>
            </xsl:text>
          </xsl:for-each>
        </xsl:element>
        <xsl:text>
        </xsl:text>
      </xsl:when>
      <xsl:otherwise>
        <xsl:call-template name="AttributeMapper">
          <xsl:with-param name="DSMLELEMENT"
           select="'attr'"/>
          <xsl:with-param name="ATTRNAME"
           select="translate(@name,
           'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
           'abcdefghijklmnopqrstuvwxyz')"/>
          <xsl:with-param name="ATTRVALUE"
           select="spml:value"/>
        </xsl:call-template>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:for-each>
  <!-- add default password to newly created users -->
  <xsl:variable name="lPasswd"
```

```
      select="spml:attributes/spml:attr[@name =
      $RES_PASSWORD]"/>
     <xsl:choose>
      <xsl:when test="$lPasswd != ''"/>
      <xsl:otherwise>
       <xsl:call-template name="AttributeBuilder">
        <xsl:with-param name="DSMLELEMENT" select="'attr'"/>
        <xsl:with-param name="ATTRNAME"
         select="$SI_PASSWORD"/>
        <xsl:with-param name="ATTRVALUE"
         select="$DEFAULT_PASSWORD"/>
       </xsl:call-template>
      </xsl:otherwise>
     </xsl:choose>
    </xsl:element>  <!-- attributes -->
    <xsl:text>
    </xsl:text>
  </xsl:element>  <!-- addRequest -->
  <xsl:text>
  </xsl:text>
</xsl:template>

<!-- ************************************
 Handler for modifyRequest elements
 ************************************
 -->
<xsl:template name="ModifyRequestHandler"
 match="spml:modifyRequest">
  <xsl:text>
  </xsl:text>
  <xsl:element name="modifyRequest"
   namespace="urn:oasis:names:tc:SPML:1:0">
   <!-- Add the two attributes -->
   <xsl:attribute name="requestID">
    <!-- <xsl:value-of select="./@requestID" /> has some
      problem -->
    <xsl:value-of select="'1'" />
   </xsl:attribute>
   <xsl:attribute name="execution">
    <!-- <xsl:value-of select="./@execution" /> has some
     problem -->
    <xsl:value-of
     select="'urn:oasis:names:tc:SPML:1:0#asynchronous'" />
   </xsl:attribute>
   <xsl:text>
   </xsl:text>
```

```
<xsl:call-template name="OperationalAttrHandler">
  <xsl:with-param name="ADDREQFLAG" select="'false'"/>
</xsl:call-template>
<xsl:element name="identifier" >
  <xsl:attribute name="type">
    <xsl:value-of select="spml:identifier/@type" />
  </xsl:attribute>
  <xsl:call-template name="IdentityHandler">
    <xsl:with-param name="DSMLELEMENT" select="'id'"/>
    <xsl:with-param name="IDVALUE"
     select="spml:identifier/spml:id"/>
  </xsl:call-template>
</xsl:element> <!-- identifier -->
<xsl:text>
</xsl:text>
<xsl:element name="modifications">
  <xsl:text>
  </xsl:text>
  <!-- Convert the attribute names -->
  <xsl:for-each
   select="spml:modifications/spml:modification">
    <xsl:choose>
      <xsl:when
       test="@name='urn:trulogica:concero:2.0#groups'">
        <xsl:element name="modification" >
          <xsl:attribute name="name">
            <xsl:value-of select="@name" />
          </xsl:attribute>
          <xsl:attribute name="operation">
            <xsl:value-of select="@operation" />
          </xsl:attribute>
          <xsl:text>
          </xsl:text>
          <xsl:for-each select="spml:value">
            <xsl:element name="value" >
              <xsl:value-of select="." />
            </xsl:element>
            <xsl:text>
            </xsl:text>
          </xsl:for-each>
        </xsl:element>
        <xsl:text>
        </xsl:text>
      </xsl:when>
      <xsl:otherwise>
        <xsl:call-template name="AttributeMapper">
```

```
            <xsl:with-param name="DSMLELEMENT"
             select="'modification'"/>
            <xsl:with-param name="ATTRNAME"
             select="translate(@name,
             'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
             'abcdefghijklmnopqrstuvwxyz')"/>
            <xsl:with-param name="ATTRVALUE"
             select="spml:value"/>
            <xsl:with-param name="MODIFYFLAG"
             select="spml:operation"/>
          </xsl:call-template>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each>
   </xsl:element> <!-- modifications -->
   <xsl:text>
   </xsl:text>
  </xsl:element> <!-- modifyRequest -->
  <xsl:text>
  </xsl:text>
</xsl:template>

<!-- ************************************
 Handler for deleteRequest elements
 ************************************
 -->
<xsl:template name="DeleteRequestHandler"
 match="spml:deleteRequest">
  <xsl:text>
  </xsl:text>
  <xsl:element name="deleteRequest"
   namespace="urn:oasis:names:tc:SPML:1:0">
   <!-- Add the two attributes -->
   <xsl:attribute name="requestID">
     <!-- <xsl:value-of select="./@requestID" /> has some
      problem
      -->
     <xsl:value-of select="'1'" />
   </xsl:attribute>
   <xsl:attribute name="execution">
     <!-- <xsl:value-of select="./@execution" /> has some
      problem
      -->
     <xsl:value-of
      select="'urn:oasis:names:tc:SPML:1:0#asynchronous'" />
   </xsl:attribute>
```

```
    <xsl:text>
    </xsl:text>
    <xsl:call-template name="OperationalAttrHandler">
      <xsl:with-param name="ADDREQFLAG" select="'false'"/>
    </xsl:call-template>
    <xsl:element name="identifier" >
      <xsl:attribute name="type">
        <xsl:value-of select="spml:identifier/@type" />
      </xsl:attribute>
      <xsl:call-template name="IdentityHandler">
        <xsl:with-param name="DSMLELEMENT" select="'id'"/>
        <xsl:with-param name="IDVALUE"
         select="spml:identifier/spml:id"/>
      </xsl:call-template>
    </xsl:element> <!-- identifier -->
  </xsl:element> <!-- deleteRequest -->
  <xsl:text>
  </xsl:text>
</xsl:template>

<!-- ************************************
 Handler to convert the identifier value
 ************************************
 -->
<xsl:template name="IdentityHandler">
  <xsl:param name="DSMLELEMENT" />
  <xsl:param name="IDVALUE" />
  <xsl:text>
  </xsl:text>
  <xsl:element name="{$DSMLELEMENT}" >
    <xsl:value-of select="$IDVALUE" />
  </xsl:element>
  <xsl:text>
  </xsl:text>
</xsl:template>

<!-- ************************************
 Handler to output attr name/value pairs
 ************************************
 -->
<xsl:template name="AttributeBuilder">
  <xsl:param name="DSMLELEMENT" />
  <xsl:param name="ATTRNAME" />
  <xsl:param name="ATTRVALUE" />
  <xsl:param name="MODIFYFLAG" />
  <xsl:variable name="ATTRVAL"
   select="normalize-space($ATTRVALUE)" />
```

```
  <xsl:if test="$ATTRVAL != ''">
    <xsl:element name="{$DSMLELEMENT}" >
      <xsl:attribute name="name">
        <xsl:value-of select="$ATTRNAME" />
      </xsl:attribute>
      <xsl:if test="$MODIFYFLAG != ''" >
        <xsl:attribute name="operation">
          <xsl:value-of select="$MODIFYFLAG" />
        </xsl:attribute>
      </xsl:if>
      <xsl:element name="value">
        <xsl:value-of select="$ATTRVAL" />
      </xsl:element>
    </xsl:element>
    <xsl:text>
    </xsl:text>
  </xsl:if>
</xsl:template>

<!-- Handler to generate operationalAttributes section -->
<xsl:template name="OperationalAttrHandler">
  <xsl:param name="ADDREQFLAG" />
  <xsl:element name="operationalAttributes">
    <xsl:text>
    </xsl:text>
    <xsl:for-each select="spml:operationalAttributes/spml:attr">
      <xsl:call-template name="AttributeBuilder">
        <xsl:with-param name="DSMLELEMENT" select="'attr'"/>
        <xsl:with-param name="ATTRNAME" select="@name"/>
        <xsl:with-param name="ATTRVALUE" select="spml:value"/>
      </xsl:call-template>
    </xsl:for-each>
    <!-- keyFields: Name of the key field from the resource -->
    <xsl:call-template name="AttributeBuilder">
      <xsl:with-param name="DSMLELEMENT" select="'attr'"/>
      <xsl:with-param name="ATTRNAME"
       select="'urn:trulogica:concero:2.0#keyFields'"/>
      <xsl:with-param name="ATTRVALUE" select="$SI_USERID"/>
    </xsl:call-template>
    <xsl:if test="$ADDREQFLAG='true'">
      <!-- taUserName: Value of the key field from the resource
       -->
      <xsl:call-template name="AttributeBuilder">
        <xsl:with-param name="DSMLELEMENT" select="'attr'"/>
        <xsl:with-param name="ATTRNAME"
         select="'urn:trulogica:concero:2.0#taUserName'"/>
```

```
        <xsl:with-param name="ATTRVALUE"
         select="spml:attributes/spml:attr[@name = $RES_USERID]"
         />
       </xsl:call-template>
       <!-- taResourceKey: Value of the key field from the
        resource -->
       <xsl:call-template name="AttributeBuilder">
        <xsl:with-param name="DSMLELEMENT" select="'attr'"/>
        <xsl:with-param name="ATTRNAME"
         select="'urn:trulogica:concero:2.0#taResourceKey'"/>
        <xsl:with-param name="ATTRVALUE"
         select="spml:attributes/spml:attr[@name = $RES_USERID]"
         />
       </xsl:call-template>
     </xsl:if>
   </xsl:element> <!-- operationalAttributes -->
   <xsl:text>
   </xsl:text>
</xsl:template>  <!-- OperationalAttrHandler -->

<!-- ************************************************************
 Handler to convert the attribute names
 Use the file aduser.properties and reverse map the attributes
 onto the names on the left side
 NOTE: The incoming attribute names are converted to lowercase
 by the caller
 ************************************************************
 -->
<xsl:template name="AttributeMapper">
  <xsl:param name="DSMLELEMENT" />
  <xsl:param name="ATTRNAME" />
  <xsl:param name="ATTRVALUE" />
  <xsl:param name="MODIFYFLAG" />
  <xsl:if test="$ATTRNAME != ''" >
    <xsl:choose>
      <!--
       Block for mapping attribute defined $RES_ATTRxx defined
       earlier. Add similar blocks as required for any new
       attributes. Example here shows the mapping for $RES_ATTR0
       and $RES_ATTR1 defined earlier.
       -->
      <xsl:when test="$ATTRNAME = $RES_ATTR0">
        <xsl:call-template name="AttributeBuilder">
          <xsl:with-param name="DSMLELEMENT"
           select="$DSMLELEMENT"/>
          <xsl:with-param name="ATTRNAME" select="$SI_ATTR0"/>
```

```
            <xsl:with-param name="ATTRVALUE" select="$ATTRVALUE"/>
            <xsl:with-param name="MODIFYFLAG" select="$MODIFYFLAG"
             />
          </xsl:call-template>
        </xsl:when>
        <xsl:when test="$ATTRNAME = $RES_ATTR1">
          <xsl:call-template name="AttributeBuilder">
            <xsl:with-param name="DSMLELEMENT"
             select="$DSMLELEMENT"/>
            <xsl:with-param name="ATTRNAME" select="$SI_ATTR1"/>
            <xsl:with-param name="ATTRVALUE" select="$ATTRVALUE"/>
            <xsl:with-param name="MODIFYFLAG" select="$MODIFYFLAG"
             />
          </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
          <!-- ignore the attribute -->
        </xsl:otherwise>
      </xsl:choose>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

Many of the XML and resource fields are configurable variables. The code snippets below explain how to configure this template for any mapping file:

- RES_USERID is the resource attribute that represents the user ID of the user on the resource. The RES_PASSWORD is the resource attribute that represents the password on the resource. The following shows the use of these variables in the XSL template:

  ```
  <!-- Physical resource user id and password attribute names -->

  <!-- Replace USERID with the user id attribute name from
   physical resource as defined in the mapping file -->
  <xsl:variable name="RES_USERID" select="'USERID'"/>

  <!-- Replace PASSWORD with the password attribute name from
   physical resource as defined in the mapping file -->
  <xsl:variable name="RES_PASSWORD" select="'PASSWORD'"/>
  ```

  The following is an example from the XSL file for the SQL Server connector:

  ```
  <!-- Physical resource user id and password attribute names -->

  <xsl:variable name="RES_USERID"
  select="'schema=dbo,table=USERINFO_1,column=USERID'"/>
  ```

```
<xsl:variable name="RES_PASSWORD"
select="'schema=dbo,table=USERINFO_1,column=PASSWORD'"/>
```

- SI_USERID is the Select Identity attribute that represents the user ID, and SI_PASSWORD is the Select Identity attribute that represents the password. The following shows the use of these variables in the XSL template:

```
<!-- User ID and Password SI Attribute names -->

<!-- Replace UserId with the user id attribute name as mapped
 onto SI -->
<xsl:variable name="SI_USERID" select="'UserId'"/>

<!-- Replace Password with the password attribute name as mapped
 onto SI -->
<xsl:variable name="SI_PASSWORD" select="'Password'"/>
```

  The following is an example from the XSL file for the SQL Server connector:

```
<!-- User ID and Password SI Attribute names -->

<xsl:variable name="SI_USERID" select="'UserId'"/>

<xsl:variable name="SI_PASSWORD" select="'Password'"/>
```

- For each of the resource attributes, the following blocks of code are defined:

```
<!-- Example mappings ($RES_ATTR0 and $RES_ATTR1). Replace
 xxxxxx with the attribute names. Add more as needed and add
 them in the AttributeMapper template -->

<xsl:variable name="RES_ATTR0" select="'xxxxxxxxxxx'"/>
<xsl:variable name="SI_ATTR0" select="'xxxxxxxxxxx'"/>
```

  and

```
<!-- Block for mapping attribute defined $RES_ATTRxx defined
 earlier. Add similar blocks as required for any new attributes
 Example here shows the mapping for $RES_ATTR0 and $RES_ATTR1
 defined earlier.
 -->

<xsl:when test="$ATTRNAME = $RES_ATTR0">
  <xsl:call-template name="AttributeBuilder">
    <xsl:with-param name="DSMLELEMENT" select="$DSMLELEMENT"/>
    <xsl:with-param name="ATTRNAME" select="$SI_ATTR0"/>
    <xsl:with-param name="ATTRVALUE" select="$ATTRVALUE"/>
```

```
    <xsl:with-param name="MODIFYFLAG" select="$MODIFYFLAG"/>
  </xsl:call-template>
</xsl:when>
```

Resource attribute names are definted by RES_ATTR0, RESATTR1, RES_ATTR2, and so on. For each RES_ATTR*x* variable, there is a corresponding SI_ATTR*x* variable, which defines the Select Identity attribute to which the resource attribute is mapped. An example from the XSL file for the SQL Server connector is provided here:

```
<xsl:variable name="RES_ATTR0"
 select="'schema=dbo,table=userinfo_1,column=userid'"/>
<xsl:variable name="SI_ATTR0" select="'UserId'"/>

<xsl:variable name="RES_ATTR1"
 select="'schema=dbo,table=userinfo_1,column=address'"/>
<xsl:variable name="SI_ATTR1" select="'Address'"/>
```

and

```
<xsl:when test="$ATTRNAME = $RES_ATTR0">
  <xsl:call-template name="AttributeBuilder">
    <xsl:with-param name="DSMLELEMENT" select="$DSMLELEMENT"/>
    <xsl:with-param name="ATTRNAME" select="$SI_ATTR0"/>
    <xsl:with-param name="ATTRVALUE" select="$ATTRVALUE"/>
    <xsl:with-param name="MODIFYFLAG" select="$MODIFYFLAG"/>
  </xsl:call-template>
</xsl:when>

<xsl:when test="$ATTRNAME = $RES_ATTR1">
  <xsl:call-template name="AttributeBuilder">
    <xsl:with-param name="DSMLELEMENT" select="$DSMLELEMENT"/>
    <xsl:with-param name="ATTRNAME" select="$SI_ATTR1"/>
    <xsl:with-param name="ATTRVALUE" select="$ATTRVALUE"/>
    <xsl:with-param name="MODIFYFLAG" select="$MODIFYFLAG"/>
  </xsl:call-template>
</xsl:when>
```

These blocks are defined for every RES_ATTR*x* variable that is defined.

JNDI Registration of the Parameter Factory Implementation

You must register the parameter factory implementation with JNDI. Select Identity will look up the parameter factory when creating instances of TAConnectorParamValueBeans.

The following sample code illustrates how you could register the parameter factory implementation with the JNDI on the application server. Select Identity will reference this factory and use it to create instances of ParamValueBean in which it passes the connection information.

```
private void registerParamFactory(String connectorJndiName)
throws Exception
{
  String lFuncName = "registerParamFactory()";
  LDAPParamFactory paramFactory = new LDAPParamFactory();
  InitialDirContext initCtx = new InitialDirContext();

  // Initialize the factory
  paramFactory.initialize();

  try {
    initCtx.lookup("eis");
  } catch (NameNotFoundException e) {
    initCtx.createSubcontext("eis");
  }

  // Register param factory with JNDI
  // Example: eis/LDAPv3-ParamFactory
  String lPfJndiName = connectorJndiName +
    TAConnectorParameterFactory.JNDI_PARAMFACTORY_SUFFIX;
  try {
    initCtx.lookup(lPfJndiName);
    catch (NameNotFoundException e) {
    initCtx.bind(lPfJndiName, paramFactory);
  } finally {
      initCtx.rebind(lPfJndiName, paramFactory);
    }
  }
```

Mapping Select Identity Attributes to the Resource Schema

As described in Coding the Connector on page 30, you must create a file that maps the Select Identity fields defined for a user to the fields used by the resource. The connector will reference this mapping file to understand the target fields on the resource for each user value. This section provides an overview of the mapping file.

The LDAP connector provides three mapping files: one for an Active Directory server (`ActiveDir.xml`), one for an iPlanet server (`iPlanet.xml`), and one for ETrust (`CAEtrust.xml`). The files are created in XML, according to SPML standards, and are bundled in a JAR file called `schema.jar`. In general, all connectors that provide XML mapping files must provide the following content.

This mapping file is always stored in the `com/trulogica/truaccess/connector/schema/spml` directory and the parent folder is packaged in the schema JAR file.

General Attribute Information

The following operations can be performed in the mapping file:

- Add a new attribute mapping
- Delete an existing attribute mapping
- Modify attribute mappings

Here is an explanation of the elements in the XML mapping files provided by the LDAP connectors:

- **<Schema>**, **<providerID>**, and **<schemaID>**

  Provides standard elements for header information.

- **<objectClassDefinition>**

  Defines the actions that can be performed on the specified object as defined by that name attribute (in the <properties> element block) and the Select Identity-to-resource field mappings for the object (in the <memberAttributes> block). In general, the XML mapping file supports two types of entities: users and groups . These entities are defined in the mapping file by an <objectClassDefinition> block.

  - **<properties>**
    Defines the operations that are supported on the object. This can be used to control the operations that are performed through Select Identity. The following operations can be controlled:

    — Create (CREATE)

    — Read (READ)

    — Update (UPDATE)

— Delete (DELETE)

— Enable (ENABLE)

— Disable (DISABLE)

— Reset password (RESET_PASSWORD)

— Change password (CHANGE_PASSWORD)

— Assign entitlements (LINK)

— Unassign entitlements (UNLINK)

— Retrieve entitlements (GETALL)

The operation is assigned as the name of the <attr> element and access to the operation is assigned to a corresponding <value> element. You can set the values as follows:

— true — the operation is supported by the connector

— false — the operation is not supported by the connector and will throw a permission exception

— bypass — the operation is not supported by the connector but will not throw an exception; the operation is simply bypassed

Here is an example:

```
<objectClassDefinition name="User" description="Oracle ERP
User">
   <properties>
    <attr name="GETCHILDREN">
      <value>true</value>
    </attr>
    <attr name="DELETE">
      <value>true</value>
    </attr>
    <attr name="EXPIREPASSWORD">
      <value>false</value>
    </attr>
    <attr name="GETALL">
      <value>true</value>
    </attr>
...
```

- **<memberAttributes>**
  Defines the attribute mappings. This element contains <attributeDefinitionReference> elements that describe the mapping

for each attribute. Each <attributeDefinitionReference> can be followed by an <attributeDefinition> element that specifies details such as minimum length, maximum length, and so on.

Each <attributeDefinitionReference> element contains the following attributes:

- Name — the name of the attribute definition reference. Make sure this fis ollowed by an <attributeDefinition> block whose name attribute matches this name.
- Required— whether this attribute is required in the provisioning process (set to true or false).
- Concero:tafield — the name of the attribute in Select Identity. In general, the attribute assigned to tafield should be the same as the physical resource attribute, or at least the connector attribute. For example, it is recommended to have the following:

  ```
  <attributeDefinitionReference name="FirstName"
  required="false" concero:tafield="[givenname]"
  concero:resfield="givenname" concero:init="true"
  concero:isMulti="true"/>
  ```

  instead of this:

  ```
  <attributeDefinitionReference name="FirstName"
  required="false" concero:tafield="[FirstName]"
  concero:resfield="givenname" concero:init="true"
  concero:isMulti="true"/>
  ```

- Concero:resfield — the name of the attribute from the resource schema. If the resource does not support physical attributes, this can be a tag field that indicates a resource attribute mapping.

  Also, the attribute name may be case-sensitive; for example, if the attribute is defined in all uppercase letters on the resource, be sure to specify it in all uppercase letters here.

- Concero:isKey — An optional attribute that, when set to true, specifies that this is the key field to identify the object on the resource. Only one <attributeDefinitionReference> can be specified where isKey="true". This key field does not need to be the same as the key field of the identity object in Select Identity.

Note that for a key field mapping where isKey=”true” and tafield is not assigned the UserName attribute, UserName should not be used in any other mapping. That is, UserName can be assigned to tafield only in cases where it is mapped to the key field in the resource. Example:

```
<attributeDefinitionReference name="UserName"
required="true" concero:tafield="[UserName]"
concero:resfield="uid" concero:isKey="true"
concero:init="true"/>
```

— Concero:init — Set this to true if this attribute needs to be passed as part of the creation of the user. You can use this parameter to control which attributes must be specified during creation and which must be specified when a user is modified.

Here is an example:

```
<memberAttributes>
  <attributeDefinitionReference name="ATTR_UserName"
  required="true" concero:tafield="UserName"
  concero:resfield="[x_user_name][USER_NAME][][VARCHAR]"
  concero:isKey="true" concero:init="true"/>
...
```

The interpretation of the mapping between the connector field (as specified by the Concero:tafield attribute) and the resource field (as specified by the Concero:resfield attribute) is determined by the connector.

- **<attributeDefinition>**

  Defines the properties of each object’s attribute. For example, the attribute definition for the Directory attribute defines that it must be between one and 50 characters in length and can contain the following letters, numbers, and characters: a-z, A-Z, 0-9, @, +, and a space.

  Here is an example:

```
<attributeDefinition name="ATTR_ResponsibilityKey"
  description="Responsibility Key" type="xsd:string" >
    <properties>
      <attr name="minLength">
        <value>1</value>
      </attr>
      <attr name="maxLength">
        <value>128</value>
      </attr>
```

```
      <attr name="pattern">
        <value><![CDATA[[a-zA-Z0-9@]+]]> </value>
      </attr>

    </properties>
</attributeDefinition>
```

- **<concero:entitlementMappingDefinition>**

  Defines how entitlements are mapped to users. Defining this element for each entitlement enables you to control the entitlements from the XML mapping file, instead of the requiring that the connector retrieve a list of entitlements from the resource. Using this element may not be appropriate in all cases, but this is one way to do it:

```
<concero:entitlementMappingDefinition>
  <concero:entitlementMap name="Administrators" />
</concero:entitlementMappingDefinition>
<concero:entitlementMappingDefinition>
  <concero:entitlementMap name="Backup Operators" />
</concero:entitlementMappingDefinition>
<concero:entitlementMappingDefinition>
  <concero:entitlementMap name="Guests" />
</concero:entitlementMappingDefinition>
<concero:entitlementMappingDefinition>
  <concero:entitlementMap name="Network Config Operators" />
</concero:entitlementMappingDefinition>
<concero:entitlementMappingDefinition>
  <concero:entitlementMap name="Power Users" />
</concero:entitlementMappingDefinition>
<concero:entitlementMappingDefinition>
  <concero:entitlementMap name="Remote Desktop Users" />
</concero:entitlementMappingDefinition>
<concero:entitlementMappingDefinition>
  <concero:entitlementMap name="Replicator" />
</concero:entitlementMappingDefinition>
<concero:entitlementMappingDefinition>
  <concero:entitlementMap name="Users" />
</concero:entitlementMappingDefinition>

<concero:entitlementMappingDefinition>
  <concero:entitlementMap name="Debugger Users" />
</concero:entitlementMappingDefinition>
<concero:entitlementMappingDefinition>
  <concero:entitlementMap name="HelpServicesGroup" />
</concero:entitlementMappingDefinition>
```

- **<concero:objectStatus>**

  Defines how to assign status to a user.

- **<concero:relationshipDefinition>**

  Defines how to create relationships between users and groups (entitlements). Here is an example:

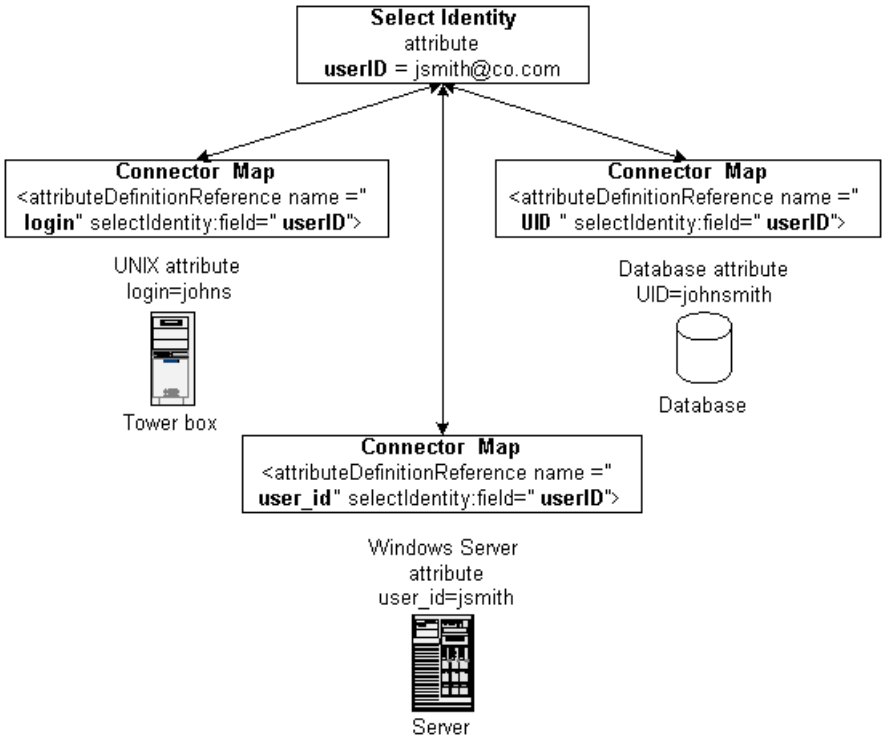
```
<concero:relationshipDefinition>
  <properties>
    <attr name="CREATE">
      <value>true</value>
    </attr>
    <attr name="NAVIGATE">
      <value>true</value>
    </attr>
    <attr name="DELETE">
      <value>true</value>
    </attr>
  </properties>
  <concero:party concero:entity="User"
   concero:cardinality="ZERO_OR_MORE" concero:start="false" />
  <concero:party concero:entity="Group"
   concero:cardinality="ZERO_OR_MORE" concero:start="false" />
</concero:relationshipDefinition>
```

  This example defines that it is allowed to create a user-to-group link, the connector and resource support this operation, the user-to-group link may be deleted, and a user can be unassigned from an entitlement.

Creating a Mapping File

Create a mapping file that maps each attribute on the physical resource to an attribute on the connector. (To complete this mapping, attributes must be created using the Select Identity client to map a name on the server to this name on the connector.) For example, the connector may store the user ID in a field called **userID** and the resource may store the ID in a field called **user_id**. The connector will reference the mapping file to understand the target field on the resource for each user value.

The following illustrates the relationship between the fields in Select Identity, the connector, and the resource:



Instances of UserModel and GroupModel are populated and provided by Select Identity when it calls the TAConnector methods. Obtain user and group attributes from here and map them to the resource using map file.

You determine the format of the mapping file. The connector may require only a simple mapping stored in a text file. Here is a simple text file example where the Select Identity field is specified first and a pipe (|) separates the fields:

```
User Name|UserId
Password|Password
User Name|cn
First Name|givenName
Last Name|sn
[First Name] [Last Name]|displayName
Title|Title
```

```
Directory|homeDirectory
Email|Mail
Address 1|streetAddress
```

Or, the connector may require a format that supports robust mapping, such as an XML file. XML mapping files are used by all connectors built and provided by HP. Here is an excerpt from the `iPlanet.xml` file, which is provided with the LDAP connector. Refer to Mapping Select Identity Attributes to the Resource Schema on page 66 for a full description of the file.

```
<objectClassDefinition name="User" description="LDAP User">
  <properties>
    <attr name="CREATE">
      <value>true</value>
    </attr>
    <attr name="READ">
      <value>true</value>
    </attr>
    <attr name="UPDATE">
      <value>true</value>
    </attr>
    <attr name="DELETE">
      <value>true</value>
    </attr>
    <attr name="ENABLE">
      <value>true</value>
    </attr>
    <attr name="DISABLE">
      <value>true</value>
    </attr>
    <attr name="RESET_PASSWORD">
      <value>true</value>
    </attr>
    <attr name="EXPIRE_PASSWORD">
      <value>false</value>
    </attr>
    <attr name="CHANGE_PASSWORD">
      <value>true</value>
    </attr>
  </properties>
  <memberAttributes>
    <!-- For iPlanet -->
    <attributeDefinitionReference name="UserName" required="true"
     concero:tafield="[UserName]" concero:resfield="uid"
     concero:isKey="true" concero:init="true"/>
    <attributeDefinitionReference name="Password" required="false"
     concero:tafield="[Password]" concero:resfield="userpassword"
     concero:init="true" />
```

Installing a Connector

To deploy the connector on the Select Identity server, you must copy the connector files to the target locations and configure the application server. The following procedures provide general guidelines for installing a connector on the supported application servers; the details will depend on how the connector was implemented and the type of application server.

On WebLogic

Complete the following steps to install the connector on the WebLogic Server:

1. Create a subdirectory in the Select Identity home directory where the connector's RAR file will reside.
2. Copy the RAR file to the connector subdirectory.
3. Create a schema subdirectory in the Select Identity home directory where the connector's mapping file(s) will reside.
4. Extract the contents of the JAR file to the schema subdirectory.
5. Ensure that the CLASSPATH environment variable in the WebLogic server startup script references the schema subdirectory.
6. Modify the mapping file to reflect the attribute names in Select Identity and on the resource, if necessary.
7. Start the application server if it is not currently running.
8. Log on to the WebLogic Server Console.
9. Navigate to ***My_domain*** →**Deployments** →**Connector Modules**.
10. Click **Deploy a New Connector Module**.
11. Locate and select the RAR file from the list. It is stored in the connector subdirectory.
12. Click **Target Module**.
13. Select the **My Server** (your server instance) check box.
14. Click **Continue**. Review your settings.
15. Keep all default settings and click **Deploy**. The Status of Last Action column should display Success.

If the connector is a two-way connector and uses an agent, install and configure the agent on the resource with which the connector communicates to provision users. The agent may also be used to synchronize changes to the identity objects, pushing the changes from the resource to Select Identity.

On WebSphere

Complete the following steps to install the connector on WebSphere Application Server:

1. Stop the application server.
2. Create a subdirectory in the Select Identity home directory on the application server.
3. Copy the RAR file to the subdirectory in the Select Identity home directory.
4. Extract the contents of the JAR file to the `WebSphere\AppServer\lib\ext` directory.
5. Modify the mapping file to reflect the attribute names in Select Identity and on the resource, if necessary.
6. Start the application server.
7. Log on to the WebSphere Application Server Console.
8. Navigate to **Resources** →**Resource Adapters**.
9. Click **Install RAR**.
10. In the Server path field, enter the path to the RAR file.
11. Click **Next**.
12. In the Name field, enter a name for the connector.
13. Click **OK**.
14. Click the **Save** link (at the top of the page).
15. On the Save to Master Configuraton dialog, click the **Save** button.
16. Click **Resources** →**Resource Adapters**.
17. Click the new connector.
18. Click **J2C Connection Factories** in the Additional Properties table.

19 Click **New**.

20 In the Name field, enter the name of the factory (enter **`eis/ess_name`**) for the connector.

21 Click **OK**.

22 Click the **Save** link.

23 On the Save to Master Configuraton dialog, click the **Save** button.

24 Restart WebSphere.

Configuring a Connector in Select Identity

After you create a connector, you can configure it for use by Select Identity using the Select Identity client (interface). The following provides an overview of the procedures you must complete in order to deploy your connector:

1 After you build and install the connector, you must register it with Select Identity. Do so on the home page of the Connectors tab by clicking the **Deploy New Connector** button. Complete this procedure, referencing your connector files, as described in the "Connectors" chapter of the *HP OpenView Select Identity Administrator Guide*.

2 You must deploy the resource that uses the newly created connector. On the home page of the Resources tab, click the **Deploy New Resource** button. Complete the steps in this procedure, referencing the new connector created in step 1, as described in the "Resources" chapter of the *HP OpenView Select Identity Administrator Guide*.

3 Create attributes that link Select Identity to the connector. For each mapping in the connector's mapping file, create an attribute using the Attributes capability on the Select Identity client. Refer to the "Attributes" chapter in the *HP OpenView Select Identity Administrator Guide* for more information.

4 Create a Service that will use the newly created resource. To do so, click the **Deploy New Service** button on the home page of the Services tab. Complete this procedure as described in "Services" of the *HP OpenView Select Identity Administrator Guide*. You will reference your new resource created in step 2 while creating this Service.

Testing a Connector

To test a connector, verify that you can perform user provisioning tasks. Perform each of the following tasks to thoroughly test the connector.

1 Verify provisioning operations using the Select Identity client. Go to the Users home page and perform the following tasks, if applicable. Refer to the *HP OpenView Select Identity Administrator Guide* for detailed information.

- Add a user
- Modify the user attributes
- Delete an existing user from the resource
- Retrieve the details of user from the resource
- Disable the user on the resource
- Enable the user on the resource
- Change the user's password
- Retrieve all entitlements present in the resource
- Associate entitlements with an existing user on the resource
- Remove entitlements from the user
- Synchronize passwords, which involves changing a user's password on the resource; the resource should then propagate to the existing user in Select Identity
- With an agent-based connector, an SPML **<extendedRequest>** request should be sent to the Select Identity Web Service with the password information
- Reverse synchronization, which involves synchronizing Select Identity with changes to identity information on the resource.

2 Perform the following operations directly on the resource using its interface. These tests verify the reconciliation in Select Identity. With an agent-based connector, SPML requests should be sent back to the Select Identity Web Service with the changes made on the resource.

- Add a new user on the resource. This should result in an SPML **<addRequest>** request including all the attributes of the user.
- Modify the user attributes on the resource. This should result in an SPML modifyRequest with the modified attribute information
- Delete an existing user from the resource. This should result in an SPML deleteRequest with the id of the user
- Disable the user on the resource. This should result in an SPML extendedRequest with all the attributes of the user
- Enable the user on the resource. This should result in an SPML extendedRequest with all the attributes of the user
- Associate entitlements to an existing user on the resource. This should result in an SPML modifyRequest with the new entitlements added.
- Dissociate entitlements from the user. This should result in an SPML modifyRequest with the removal of entitlements
- Associate some and dissociate some entitlements on the user on the resource. This should result in an SPML modifyRequest addition/ deletion of entitlements.

3 Verify changes made on the ID object in the Select Identity repository. You can view user attribute or service membership information in the repository.

3

LDAP Connector Example

The Active Directory LDAP connector enables HP OpenView Select Identity to manage user data in LDAP. It is a one-way connector and pushes changes made to user data in the Select Identity database to a target LDAP server. This connector is generic and can be used to connect to any LDAP data source. The mapping file controls how Select Identity fields are mapped to LDAP fields.

The mapping file, source files, definition file, and build files are provided in a ZIP file in the `docs/api_help/connectors/LDAPv3_v3` directory on the Select Identity CD. Extract this file to review the LDAP connector's source files.

This chapter provides an explanation of the source code that implements the LDAP connector, the mapping file that Select Identity refers to when pushing data, and the packaging. Use this example to help you build your own connector.

Description of the Connector Source Files

The following provides a description of the files extracted from the `connector_src.jar` file:

- `LDAPConnector.java`

  This is the implementation of TAConnector interface to provision users onto the LDAP data store. This represents a physical connection to the LDAP store.

  The class uses the JNDI API for a directory interface to access and update LDAP. Connection parameters should contain the URL to access the LDAP store and the root directory name and password. This class uses the SPML-based XML mapping file to map Select Identity resource fields to LDAP attributes.

- `LDAPManagedConnectionFactory.java`

  This is the implementation of the javax.resource.spi.ManagedConnectionFactory interface. This class is registered with the application server by specifying the managedconnectionfactory-class in the `ra.xml` deployment descriptor file.

  The application server calls on this implementation to create and return an instance of ManagedConnection, which represents the connection to the resource and matches existing managed connections with the given one. Also, the connector parameter factory implementation is registered with JNDI in this file.

- `LDAPConnectorFactory.java`

  This is the implementation of TAConnectorFactory interface and represents a factory to create managed connections. This class is registered with the application server by specifying the factory under the connectionfactory-impl-class in the `ra.xml` file.

  The getConnection(TAConnectorParamValueBean connParam) method is implemented and it calls on the application server connection manager to allocate and return a new connection.

- `LDAPManagedConnection.java`

  This is the implementation of the javax.resource.spi.ManagedConnection interface and it represents the physical connection to the resource.

The application server calls the getConnection() method in this class to get a connection handle to the resource. The connection parameter value bean is passed in by the application server. A local copy of this bean is created and a new instance of LDAPConnector is created and returned.

A copy of the schema repository is maintained here for reference by LDAPConnector. This repository is built from the mapping file.

- `LDAPParamFactory.java`

  This is the implementation of TAConnectorParameterFactory interface. It is instantiated and registered with the JNDI so that Select Identity can lookup and call on this instance to create instances of beans that contain the connection parameter values.

- `LDAPParamValueBean.java`

  This is the derived class of the TAConnectorParamValueBean abstract class. It contains the names of all of the connection parameters needed to connect to and access the LDAP resource, as follows:

  - accessURL — the URL to access the LDAP store
  - suffix — the suffix of the domain name (DN) for all users and groups
  - rootDN — the root DN to log in to the LDAP store
  - rootPassword — the root password
  - userSuffix — the user suffix, such as ou=Users
  - userObjectClass — the Object class of all users
  - groupSuffix — the group suffix, such as ou=Groups
  - groupObjectClass — the Object class of all group objects
  - mappingFile — the name of the file that contains the attribute mappings

  Each instance of this bean contains one set of information for the connection parameters.

  Also, the following method are implemented:

  - getParamNames() returns all the above-listed connection parameters
  - get(name) returns the value of the connection parameter

- set(name, value) stores the value of the connection parameter. This value is passed from the configuration at the time of resource deployment

- `LDAPManagedConnectionMetaData.java`

  This is the implementation of the javax.resource.spi.ManagedConnectionMetaData interface and is used to return the EIS product name, version, and maximum connections allowed to the application server.

- `LDAPRAMetaData.java`

  This is the implementation of the javax.resource.cci.ResourceAdapterMetaData interface and is used to return the resource adapter-specific information to the application server, such as the adapter name, vendor name, and version.

- `LDAPUtil.java`

  This is a utility class that implements some methods used by other parts of the connector.

- `LDAPParamResources.properties`

  This is a text file containing configuration properties for the connector and has the default values for all of the connection parameters. This file is read in during startup by `LDAPUtil.java` to return the default values of the connection parameters. These are displayed in the Select Identity client, on the Resources home page.

- `ra.xml`

  This is the deployment descriptor for the resource adapter implementing the connector. The interface and implementation class names are registered here.

  As described in Step 7 on page 20, this file contains the name of the connector, the configuration, the interface names of the connector, and the JNDI name for the connector. Refer to the `ra.xml` file provided by the LDAP connector as an example when creating your own. Create this XML file according to the JCA specification. Here is an explanation of the elements in the `ra.xml` file:

  - **<display-name>**, **<vendor-name>**, **<spec-version>**, **<eis-type>**, **<version>**, and **<license>**

    Provides general information about the connector.

- **<managedconnectionfactory-class>**

  Specifies the path to the class implementing the ManagedConnectionFactory interface.

- **<connectionfactory-interface>**

  Specifies the path to the TAConnectorFactory interface.

- **<connectionfactory-impl-class>**

  Specifies the path to the class implementing the TAConnectorFactory interface.

- **<connection-interface>**

  Specifies the path to the TAConnector interface.

- **<connection-impl-class>**

  Specifies the path to the class implementing the TAConnector interface.

- **<transaction-support>**

  Specifies whether the connector supports transactions.

- **<config-property>**

  Defines a configuration property for the connector. For example, the UserName property is defined. It is a string and its value is set to cn=Directory Manager.

  A <config-property> element is defined for each of the connector's configuration properties.

- **<reauthentication-support>**

  Specifies whether the connector supports authentication after the connector has communicated with Select Identity.

- `weblogic-ra.xml`

  This is the WebLogic-specific deployment descriptor for the resource adapter and contains the LDAP connector JNDI name.

- `activedir.xml`

  This is the mapping file for Active Directory. It maps Select Identity resource attributes to Active Directory attributes.

- `caetrust.xml`

  This is the mapping file for CA eTrust. It maps Select Identity resource attributes to eTrust attributes.

- `iplanet.xml`

  This is the mapping file for iPlanet. It maps Select Identity resource attributes to iPlanet attributes.

Description of the Build Files

The following XML and property files are used by Apache Ant to build the LDAP connector. Refer to the LDAP JAR file to view the contents.

- `build_sa.xml`

  This is the main build file for the connector. It references the `build.sa.properties` file and calls the `build_rar_sa.xml` file, which contains information about building the .rar file.

- `build_rar_sa.xml`

  This file contains information about building the .rar file.

- `build.sa.properties`

  This file contains definitions used by the build files. Edit the following entries in the sample file provided on the CD to reflect the name of the folders in your environment:

  ```
  # folder containing common jars needed by all connectors
  # example: apache commons beanutils, collections etc

  si.external.lib.dir=C:/SelectIdentity/external_lib

  # Folder containing SI connector interface jars

  si.connector.lib.dir=C:/SelectIdentity/connector/lib

  # folder to place the built jars and files

  connector.build.dir=C:/tmp/output
  ```

  Make sure that the WL_HOME environment variable is set and points to the WebLogic home directory. Also, ensure that the PATH environment variable references the `ant/bin` folder. After editing the file, use

build_sa.xml to build the connector by entering the following command at the command prompt:

ant -v build_sa.xml

The TALDAPv3.rar and schema.jar files will be placed in the output folder specified by the connector.build.dir entry in the build.sa.properties file.

4

Dummy Connector Example

An example connector called the Dummy Connector is provided in a ZIP file in the `docs/api_help/connectors/dummyConnector` directory on the Select Identity CD. Extract this file to review the source files. This chapter provides an snapshots of the source code that implements the Dummy Connector, the build files used to build the connector, and the schema JAR and RAR files. Use this example to help you build your own connector.

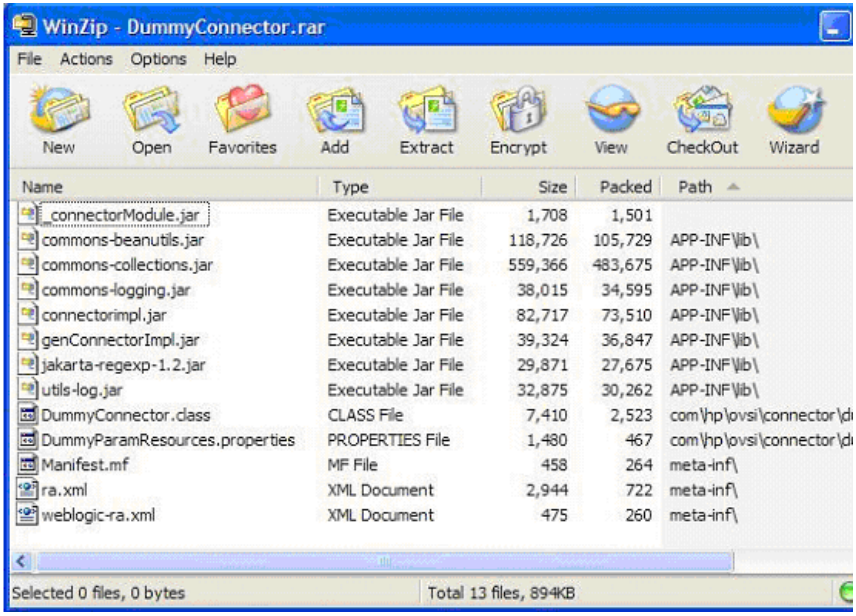
The following snapshot shows the hierarchy of the Dummy Connector source:

Here is an explanation of the folders:

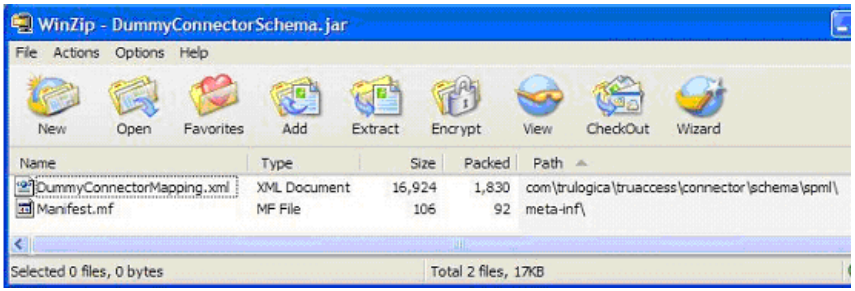
- The connector-related JAR files are in the `connector_lib` folder and the external JAR files are in `external_lib` folder.
- `ra.xml` and `weblogic-ra.xml` are in the `META-INF` folder.
- Source code and the connection parameters properties file are in the `src/com/hp/ovsi/connector/dummy` folder.
- The schema mapping file called `DummyConnectorMapping.xml` is in `src/com/trulogica/truaccess/connector/schema/spml` (the file *must* reside in this location when it is installed).

- All build files are in the main folder:
  - `build.properties` contains all properties needed to build the connector including the connector-specific properties such as the name, package name, RAR name, and so on.
  - `build.xml` is the overall build file that invokes `build_rar.xml`.
  - `build_rar.xml` compiles and builds the connector RAR and the schema JAR containing the mapping XML.

Here are the contents of the RAR file that is built from the Dummy Connector source:



Here are the contents of the schema JAR file, which contains only one mapping file called `DummyConnectorMapping.xml`:



glossary

A

Access Control List (ACL)

An abstraction that organizes entitlements and controls authorization. An ACL is list of entitlements and users that is associated with a secured object, such as a file, an operation, or an application. In an ACL-based security system, protected objects carry their protection settings in the form of an ACL.

Access Management

The process of authentication and authorization.

Action

A task that can be performed within each Select Identity capability.

In Workflow Studio, an action invokes functions provided by the workflow engine or external applications within an activity. For example, you can log information to a file, set a property to be used later in the workflow, call an external process, provision a user in Select Identity, or store data in a database.

See also: Capability

Admin Role

A template that defines the administrative actions that can be performed by a user. An Administrative Service is created to provide access to roles. Users are then given access to the Service. Users with administrative roles can also grant their set of roles to another administrator within their Service context.

Approval Process

The process of approving the association, modification, or revocation of entitlements for an identity. This process is automated of these through workflow templates.

Approver

A Select Identity administrator who has been given approval actions through an Admin Role.

Attribute

An individual field that helps define an identity profile. For each identity, an attribute has a corresponding value. For example, an attribute could be "department" with possible values of "IT," "sales," or "support."

Audit Report

A report that provides regular account interaction information within the Select Identity system.

Authentication

Verification of an identity's credentials.

Authoritative Source

A resource that has been designated as the "authority" for identity information. Select Identity accounts can be reconciled against accounts in an authoritative source.

Authorization

Real-time enforcement of an identity's entitlements. Authentication is a prerequisite for authorization.

Auto Discovery

The process of adding user accounts to the Select Identity system for a specified Service through the use of a data file.

B

Business Relationship

A Select Identity abstraction that defines how a logical grouping of users will access a Select Identity Service. The Select Identity Service is a superset of all the identity management elements of a business service.

Business Service

A product or facility offered by, or a core process used by, a business in support of its day-to-day operations. Example business services could include an online banking service, the customer support process, and IT infrastructure services such as email, calendaring, and network access.

See also: Service

C

Capability

Actions that can be performed within the Select Identity client are grouped by capability, or link, in the interface.

See also: Action

Challenge and Response

A method of supplying alternate authentication credentials, typically used when a password is forgotten. Select Identity challenges the end user with a question and the user must provide a correct response. If the user answers the question correctly, Select Identity resets the password to a random value and sends email to the user. The challenge question can be configured by the administrator. The valid response is stored for each user with the user's profile and can be updated by an authenticated user through the Self Service pages.

Configurations

A capability that enables you to import and export Select Identity settings and configurations. This is useful when moving from a test to a production environment.

Configuration Report

A report that provides current system information for user, administrator, and Service management activities.

Connector

A J2EE connector that communicates with the system resources that contain your identity profile information.

Context

A Select Identity concept that defines a logical grouping of users that can access a Service.

Contextual Identity Management (CIM)

An organizational model that introduces new abstractions that simplify and provide scale to the business processes associated with identity management. These abstractions are modeled after elements that exist in businesses today and include Select Identity Services and Business Relationships.

Credential

A mechanism or device used to verify the authenticity of an identity. For example, a user ID and password, biometrics, and digital certificates are considered credentials.

D

Data File

An SPML file that enables you to define user accounts to be added to Select Identity through Auto Discovery or Reconciliation.

Delegated Administration

The ability to securely assign a subset of administrative roles to one or more users for administrative management and distribution of workload. Select Identity enables role delegation through the Self Service pages from one administrator to another user within the same Service context.

Delegated Registration

Registration performed by an administrator on behalf of an end user.

See also: Self Registration

E

End User

A role associated to every user in the Select Identity system that enables access to the Self Service pages.

Entitlement

An abstraction of the resource privileges granted to an identity. Entitlements are resource-specific and can be resource account IDs, resource role memberships, resource group memberships, and resource access rights and privileges. Entitlements are also considered privileges, permissions, or access rights.

External Call

A programmatic call to a third-party application or system for the purpose of validating accounts or constraining attribute values.

F

Form

An electronic document used to capture information from end users. Forms are used by Select Identity in many business processes for information capture and system operation.

I

Identity

The set of authentication credentials, profile information, and entitlements for a single user or system entity. Identity is often used as a synonym for “user,” although an identity can represent a system and not necessarily a person.

Identity Management

The set of processes and technologies involved in creating, modifying, deleting, organizing, and auditing identities.

M

Management

The ongoing maintenance of an object or set of objects, including creating, modifying, deleting, organizing, auditing, and reporting.

N

Notifications

The capability that enables you to create and manage templates that define the messages that are sent when a system event occurs.

P

Password Reset

The ability to set a password to a system-generated value. Select Identity uses a challenge and response method to authenticate the user and then allow the user to reset or change a password.

Policy

A set of regulations set by an organization to assist in managing some aspect of its business. For example, policy may determine the type of internal and external information resources that employees can access.

Process

A repeatable procedure used to perform a set of tasks or achieve some objective. Whether manual or automated, all processes require input and generate output. A process can be as simple as a single task or as complicated a multi-step, conditional procedure.

See also: Approval Process

Profile

Descriptive attributes associated with an identity, such as name, address, title, company, or cost center.

Provisioning

The process of assigning authentication credentials to identities.

R

Reconciliation

The process by which Select Identity accounts are synchronized with a system resource. Accounts can be added to the Select Identity system through the use of an SPML data file.

Registration

The process of requesting access to one or more resources. Registration is generally performed by an end user seeking resource access, or by an administrator registering a user on a user's behalf.

See also: Delegated Registration, Self Registration

Request

An event within the Select Identity system for the addition, modification, or removal of a user account. Requests are monitored through the Request Status capability.

Resource

Any single application or information repository. Resources typically include applications, directories, and databases that store identity information.

Role

A simple abstraction that associates entitlements with identities. A role is an aggregation of entitlements and users, typically organized by job function.

See also: Admin Role

Rule

A programmatic control over system behavior. Rules in Select Identity are typically used for programmatic assignment of Services. Rules can also be used to detect changes in system resources.

S

Self Registration

Registration performed by an end user seeking access to one or more resources.

See also: Delegated Registration

Self Service

The ability to securely allow end users to manage aspects of a system on their own behalf. Select Identity provides the following self-service capabilities: registration, profile management, and password management (including password change, reset, and synchronization).

Service

A business-centric abstraction representing resources, entitlements, and other identity-related entities. Services represent the products and services that you offer to customers and partners.

Service Attribute

A set of attributes and values that are available for or required by a Service. Attributes are created and managed through the Attributes pages.

See also: Attribute

Service View

A restricted view of a Service that is valid for a group of users. Views enable you to define a subset of Service registration fields, change field names, reorder fields, and mask field values for specific users.

Single Sign-On (SSO)

A session/authentication process that permits a user to enter one set of credentials (name and password) in order to access multiple applications. A Web SSO is a specialized SSO system for web applications.

SPML Data File

See: Data File

U

Users

The Select Identity capability that provides consistent account creation and management across Services.

W

Workflow Process

The tasks, procedural steps, organizations or people involved, and required input and output information needed for each step in a business process. In identity management, the most common workflows are for provisioning and approval processes.

Workflow Studio

The Select Identity capability that enables you to create and manage workflow templates.

Workflow Template

A model of the provisioning process that enables Select Identity to automate the actions that approvers and systems management software must perform.

index

R

S

T

X