

HP Select Identity Software

Software Version: 4.12

Connector Developer Guide

Software Release Date: March 2007

Document Release Date: March 2007



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

© 2003-2007 Hewlett-Packard Development Company, L.P.

No part of this document may be copied, reproduced, or translated into another language without the prior written consent of Hewlett-Packard Company. The information contained in this material is subject to change without notice.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>). Portions Copyright (c) 1999-2003 The Apache Software Foundation. All rights reserved.

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.

This product includes software developed by Teodor Danciu (<http://jasperreports.sourceforge.net>). Portions Copyright (C) 2001-2005 Teodor Danciu (teodord@users.sourceforge.net). All rights reserved.

Portions Copyright 1994-2005 Sun Microsystems, Inc. All Rights Reserved.

This product includes software developed by the Waveset Technologies, Inc. (www.waveset.com). Portions Copyright © 2003 Waveset Technologies, Inc. 6034 West Courtyard Drive, Suite 210, Austin, Texas 78730. All rights reserved.

Portions Copyright (c) 2001-2005, Gaudenz Alder. All rights reserved.

Trademark Notices

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.

WebSphere Application Server is a trademark of International Business Machines Corporation.

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.

Contents

1	Select Identity Connectors	11
	Connector Types	11
	Select Identity API Architecture	12
	Select Identity Connector Architecture	12
	Select Identity Connector API Interfaces and Classes	14
	New Features in the Connector Interface	17
	Select Identity Web Services	17
2	Functional Requirements and Development Phases	19
	Platform Support	19
	Agent Communication, Security, and Logging	19
	Identity Objects and Schema Mapping	20
	Supporting and Mapping Identity Objects	20
	Provisioning, Detecting Changes, and Post-provisioning	21
	User/Entitlements Discovery	21
	User Provisioning	21
	Entitlement Provisioning	22
	Change Detection	23
	Post Provisioning	24
	Internationalization Compliance	24
	Performance and Scalability	24
	Development Phases	24
	Requirements Phase	24
	Design Phase	26
	Implementation	27
	Integration	28
	Packaging	28
	Documentation	28
3	Connector SDK	35
	Simplified Connector Interface	37
	XML Schema Handling	37
	Generic JCA Interface and Connector Implementation	38
	Connector Tester Tool	38
	Connector Template	39
4	Implementing a Select Identity Connector	43
	Development Requirements	43
	Steps to Implement a Connector for Select Identity	44

Step 1: Start with the Connector Template	44
Step 2: Gather Connector Details	44
Resource Details	44
Resource Schema Details	44
Connector Code-Related Details	45
Step 3: Working with the Connector Template	45
1. Prepare the Connector Template Files	45
2. Check the Library JAR Files	48
3. Implement the Connector	49
Sample SPML Requests for Reconciliation	49
Authoritative Reconciliation SPML Requests	49
Add User.	49
Non-Authoritative Reconciliation SPML Requests	52
Add User Entitlements	52
Delete Service Membership.	53
Change User Entitlements	54
Building a Connector for Forward Provisioning.	55
Agent-less Forward Provisioning	55
Agent-based Forward Provisioning	61
Building a Connector for Reverse Provisioning	62
Agentless Reverse Provisioning	62
Implementing getChangeLog(ChangeLogCursor)	62
Agent-based Reverse Provisioning	64
Mapping Select Identity Attributes to the Resource Schema	66
General Attribute Information.	66
Creating a Mapping File.	70
Some Coding Guidelines	72
Key Value Return	73
Key Value Computation	73
User Modification	74
Loading Existing User From Resource.	75
Computing Changes to Be Made	75
Matching Managed Connections	79
Schema Reloading.	81
5 Connector Tester Tool	83
Installing the Tester Tool.	84
Testing the Connector	85
Step 1: Get the connection parameters of your connector.	85
Step 2: Prepare the properties file with names and values of these connection parameters.	86
Step 3: Test the connection to the resource.	86
Step 4: Run the Tester Tool client using one of the scripts.	87
Tester Tool Scripts	87
Initial Connectivity-Related Scripts	87
Provisioning-Related Scripts	88
Bulk Provisioning Scripts.	89

6	Installation and Deployment	95
	Installing a Connector On WebLogic	95
	Configuring a Connector in Select Identity	96
7	Connector Migration	99
	Reasons to Migrate	99
	Interface Changes	100
	Connector API Changes	100
	Classes and Interfaces	101
	Connector Interface	102
	Attribute Operations	105
	Schema Changes	105
	Entity Definition	106
	Relationship Definition	106
	XML Mapping File Changes	106
	Steps to Migrate Connectors	107
	Step 1: Change the Connector Implementation	107
	Implement the SIConnector	107
	ConnectorFactory Implementation	115
	Step 2: Change the XML Mapping File	115
	ObjectClass Definition	115
	Attribute Mapping Definitions	116
	Relationship Definition	117
	Step 3: Remove Deprecated Methods	117
	Step 4: Use Commons Logging	117
	Step 5: Package the Connector	118
	Step 6: Use the Connector Tester Tool	118
8	The Connector Template	127
	Template Files	127
	Connector Template Code	127
	Index	133

1 Select Identity Connectors

HP Select Identity (Select Identity) lets you connect to enterprise applications and resources to configure and manage users, groups, and entitlements. Access to resources from Select Identity occurs via **connectors**. Connectors are plug-in modules that are implemented as resource adapters, and are based on the J2EE Connector Architecture (JCA) framework.

Connectors synchronize the provisioning information between Select Identity and the resources that store identity information. That is, connectors replicate the changes performed in Select Identity to the resources and from the resources to Select Identity.

This chapter provides the high-level knowledge required to develop and implement connectors. Topics in this chapter include conceptual information about the framework, diagrams outlining the API and connector architectures, and descriptions of API classes and interfaces.

- [Connector Types](#) on page 11
- [Select Identity API Architecture](#) on page 12
- [Select Identity Connector Architecture](#) on page 12
- [Select Identity Connector API Interfaces and Classes](#) on page 14
- [New Features in the Connector Interface](#) on page 17

Connector Types

Select Identity connectors can be unidirectional or bi-directional (duplex).

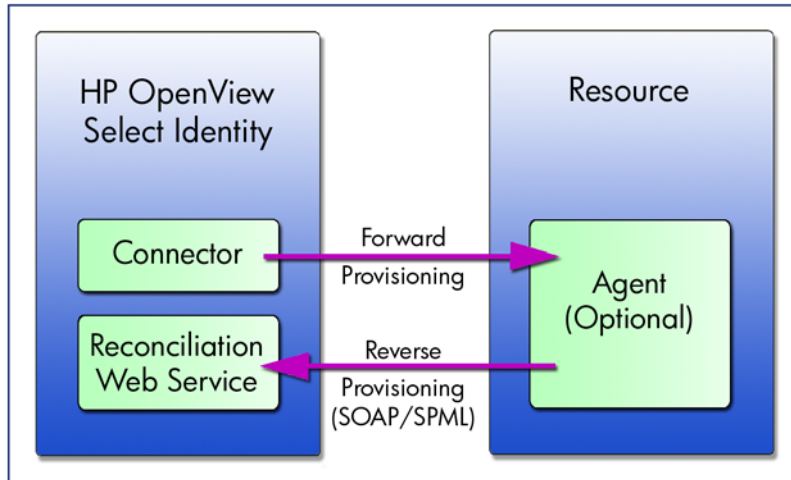
- **Unidirectional connectors** support only forward provisioning operations from Select Identity to the resource.
- **Bidirectional connectors** support reconciliation or reverse synchronization from the resource to the Select Identity server.

The connectors can also be classified as agent-less or agent-based connectors.

- **Agent-less connectors** interact with the resources directly through APIs or the remote protocols supported by the resource.
- **Agent-based connectors** use agents to communicate with the resource. Agents may be required based on various factors. For example, the resource does not provide remote APIs or resource-specific logic to be developed to reconcile the changes in the resource to Select Identity.

Figure 1 illustrates the flow of data between an Select Identity agent-based connector and an agent.

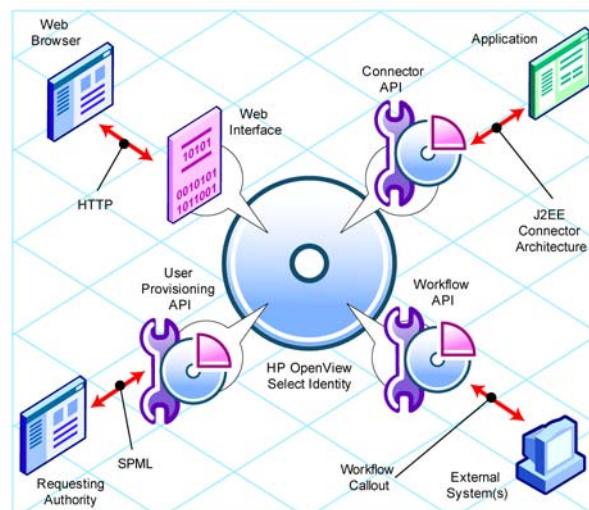
Figure 1 Data Flow Between Agent-Based Connector and Agent



Select Identity API Architecture

Figure 2 illustrates the Select Identity API architecture, showing the relationship of the Connector API to Select Identity and the other APIs.

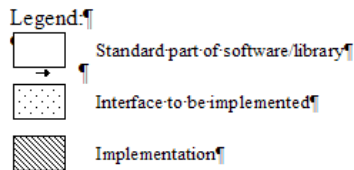
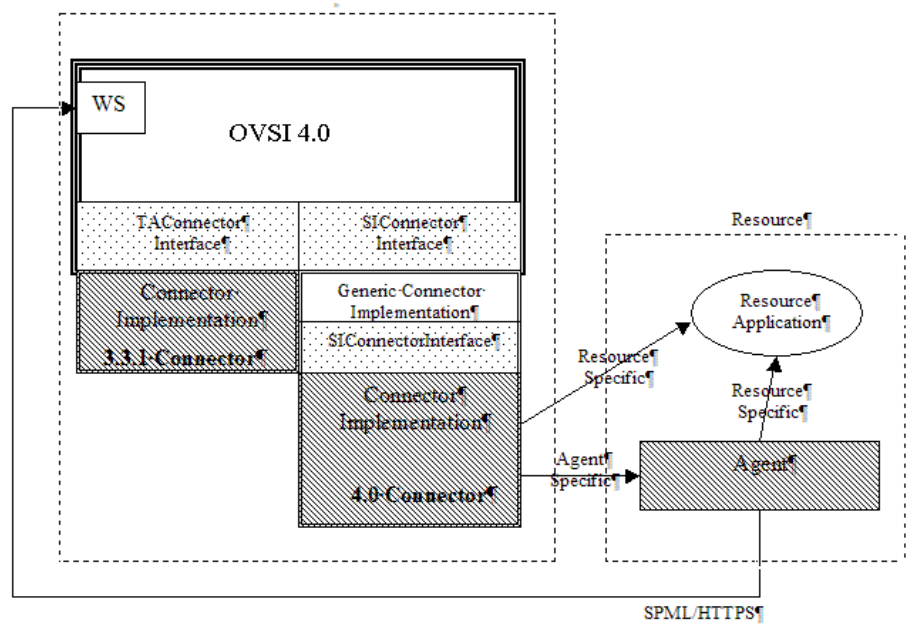
Figure 2 Select Identity API Architecture



Select Identity Connector Architecture

Figure 3 illustrates the high-level architecture of a Select Identity connector.

Figure 3 Select Identity connector architecture



The Select Identity connector is J2EE Connector Architecture (JCA) compliant and is designed to be deployed as a RAR (Resource Archive) module on the Select Identity server. The connector runs as a plug-in RAR module on the same application server as Select Identity.

All connectors must implement the **SIConnector** interface and other JCA-related interfaces. To simplify the connector development process from the intricacies of JCA, a generic connector implementation is provided. This implements the required **SIConnector** and JCA interfaces. This generic connector only requires implementation of the **SIConnector** interface. For more details, see [Select Identity Connector API Interfaces and Classes](#) on page 14.

Thus, there are two ways to develop a connector:

- Option 1: Implement the **SIConnector** interface and all JCA-related interfaces.
- Option 2: Use the Generic Connector implementation (which also provides a generic implementation for the **SIConnector** interface and JCA interface specification) and implement only the **SIConnectorInterface** interface.

As the picture shows above, it is also possible to use existing 3.3.x connectors with Select Identity 4.11.

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

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



Select Identity implements only the Connector Management portion of the JCA specification.

Select Identity Connector API Interfaces and Classes

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

- **SICConnector**

Provides the top-level interface that maps identity information to a resource type. This interface is an extension of the JCA CCI Connection interface. This is the main interface to implement to build an Select Identity connector to any resource.

- **SICConnectorFactory**

Factory to create instances of connection handles for resources. The connection handle is an implementation of **SICConnector**.

- **SIUserModel**

The interface that contains user information that is being provisioned into a resource.

- **SIJCAUserModel**

Implementation of the **SIUserModel** interface. All user attribute information passed from Select Identity to connectors is passed in an instance of this class.

- **EntitlementModel**

Interface that contains the identity of an entitlement in the resource. Represents all types of entitlements on a resource including groups, roles, privileges, access control lists (ACLs), responsibilities, and any generic entitlement.

- **JCAEntitlementModel**

Implementation of the **EntitlementModel**. Entitlement information passed from Select Identity to connectors is passed in an instance of this class.

- **SICChangeLogModel**

Class representing the changes that occurred in the resource. This contains `ChangeLogEntry` instances representing each specific change (add, modify, delete) made for each user.

- **ChangeLogEntry**

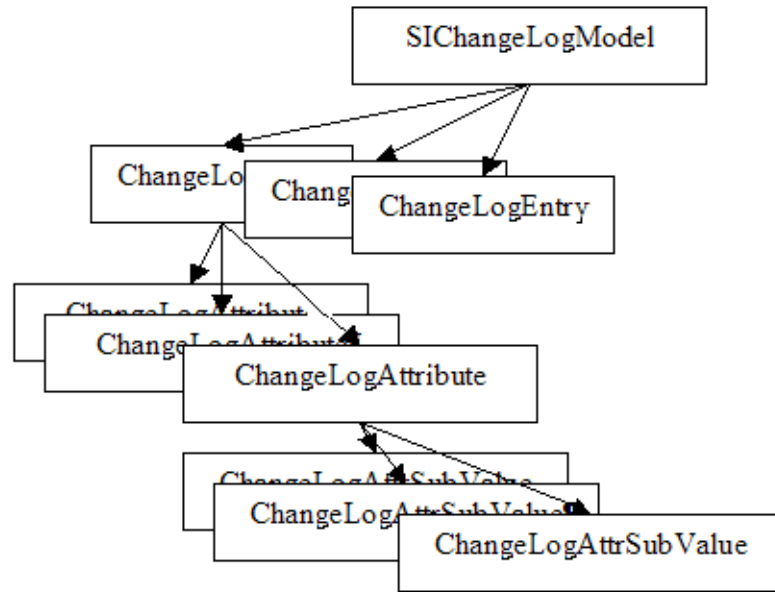
Represents a change in resource for each user (example add/modify/delete). There can be multiple instances of this class in the **SICChangeLogModel**.

- **ChangeLogAttribute**

Models changes to user attributes. This class represents one attribute in the change log along with the operation performed.

Following is a diagram illustrating the structure of **SICChangeLogModel**:

Figure 4 SIChangeLogModel Structure



- **ChangeLogAttrSubValue**

Models changes to user attributes. This class represents each of the sub-values of an attribute along with the operation performed.

- **ChangeLogCursor**

Models a cursor used with change detection. This class represents a cursor that models a check point to a previous call to `getChangeLog()` invoked on the connector.

- **TAConectorRequestIntf**

Provides a generic interface that sends a request to the connector. Use this interface if there is a requirement that cannot be supported by the existing provisioning API methods.

- **TAConectorRequest**

Implements the **TAConectorRequestIntf** interface. This is to be used as an extension to the existing API methods.

- **TAConectorResponseIntf**

Provides a generic interface that stores responses from the connector. Use this interface if there is a requirement that cannot be supported by the existing provisioning API methods.

- **TAConectorResponse**

Implements the **TAConectorResponseIntf** interface. This is to be used as an extension to the existing API methods.

- **TAAtrValueBean**

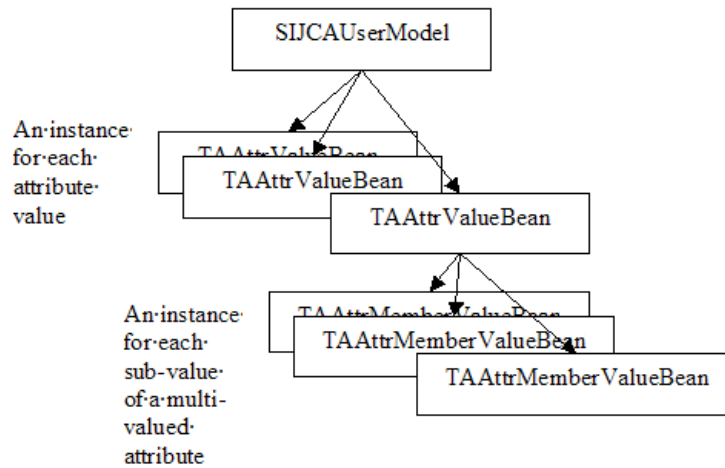
Main class containing the attribute value passed in **SIJCAUserModel**. This contains details on the attribute name. The attribute value could be single or multi-valued. This also contains attribute-level operations, which are useful in the case of a user modify operation.

- **TAAtrMemberValueBean**

Represents each of the multiple values of the attribute given in **TAttrValueBean**.

Following is a diagram illustrating the contents of **SIJCAUserModel**:

Figure 5 SIJCAUserModel Contents



- **TACConnectorParamBean**

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.

- **TACConnectorParameterFactory**

Obtains connection-specific beans that contain connection parameter values.

- **TACConnectorParamValueBean**

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.

- **TAStatus**

Represents the status of an operation. This class contains the operation called as well as the actual operation performed by the connector on the resource, including any details. This is used as a return value of most of the methods on **SICconnector**.

- **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.

- **RelationSupport**

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

- **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.

- **SICconnectorInterface**

Simplified version of the **SICConnector** interface that is used by the connector developer using the Generic Connector implementation.

SICConnectorInterface helps you focus on the efforts involved in provisioning to the resource while avoiding the details of JCA and the Select Identity user model.



If you are implementing the simplified connector interface **SICConnectorInterface**, you will not be working with most of the above classes/interfaces. The main classes and interfaces you will be working with are: **SICConnectorInterface**, **TAAAttrValueBean**, and **TAAAttrMemberValueBean**.

New Features in the Connector Interface

The Connector API has been enhanced to include support for the following additional features:

- Multi-valued attributes
- Large attribute values
- Addition/deletion/emptying of attributes
- Association/dissociation of entitlements in bulk
- Query criteria with multiple filters for entitlements retrieval
- Consolidated model for supporting different types of entitlements, such as groups, roles, privileges, ACLs and so on
- Connector SDK

Software Development Kit, which can be used to easily develop Select Identity connectors (see [Connector SDK](#) on page 35 for details). The SDK also contains the Connector Template, which provides a real example of connector implementation (see [The Connector Template](#) on page 127 for details).

Select Identity Web Services

The *HP OpenView Select Identity Web Services Guide* provides information about using SPML Web Services requests. Refer to this guide when developing SPML files for use with Select Identity.

2 Functional Requirements and Development Phases

Before implementing a connector, ensure that you meet the high-level requirements outlined in this chapter. Then, review the development phases to ensure that the connector is robust and all questions are answered before implementation.

This chapter contains the following sections:

- Platform Support
- Agent Communication, Security, and Logging
- Identity Objects and Schema Mapping
- Internationalization Compliance
- Performance and Scalability
- Development Phases

Platform Support

The connector must be implemented and deployed as a plug-in module (J2EE Connector Architecture 1.0 resource adapter) on a J2EE-based application server hosting the Select Identity server. Specifically, the connector is required to run in the following environment:

- **Select Identity version:** 4.x or higher
- **Application server:** BEA WebLogic 8.1.4
- **Operating system:** Red Hat Linux 11
- **Database:** Oracle 10G
- **Resource platform:** Depends on the resource, though in general, the most recent version is required to be supported on the most commonly supported operating systems



Not all combinations of these platforms and systems are supported by Select Identity. The connector is required to run on a valid combination as published in the *HP OpenView Select Identity Release Notes*.

Agent Communication, Security, and Logging

If an agent is implemented for the connector, the agent must be implemented as a continuously-running daemon or process that is deployed on the resource. It must handle requests sent from the connector and send responses to the connector synchronously. 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 (version 1.0) payload over HTTP or HTTPS.

The connector is required to communicate with the resource (or the agent on the resource) over a secure channel, to ensure the security of the user data that is exchanged. The following encryption standards or protocols are required to be supported:

- 128 bit AES
- SSL

The connector must support logging at all levels to a configurable file that may be different from the one used by the Select Identity server.

Identity Objects and Schema Mapping

This section describes the identity objects (attributes) and operations that must be supported by the connector. It also describes the Attribute Mapping Utility, which can be used to retrieve and map resource attributes to Select Identity attributes.

Supporting and Mapping Identity Objects

The following identity objects must be supported by the connector:

- **Users** — This is the primary identity object that must be supported. The Select Identity User object is mapped to the resource user, and the connector must support all attributes of the user that are supported by the resource, including single-valued and multi-valued attributes.
- **Entitlements** — Entitlements include organizational units, groups, entitlements, privileges, and access control lists on the resource. A user profile can be assigned to and de-assigned from an entitlement. Select Identity entitlements are mapped to resource entitlements

An Attribute Mapping Utility is provided by Select Identity to retrieve user and entitlement schema data from a resource and to map Select Identity attributes to resource attributes. The utility can retrieve the complete schema from the resource, including user and group profiles and their relationships. The Attribute Mapping Utility is integrated and invoked from Select Identity, so a common interface is provided. You can implement the connector to use the Attribute Mapping Utility. If so, user interface pages are required for displaying the resource schema and allowing resource attributes to be mapped to Select Identity attributes.

The following is a list of user attributes that can be retrieved:

- Name
- Type (text or binary)
- Size
- Permissions (create, read, update, or delete)
- Operation support (user creation, user update, reset password, ignore all operations)
- Format/Pattern
- Description (rules to consider while providing values)

- Encryption Required
- Is Password
- Is Sensitive
- Is Multi-valued
- Entitlement relationships

The following is a list of entitlement attributes that can be retrieved:

- Name
- Type (Group, Role, Entitlement, Access Level, Privilege, or Resource profile)
- Size
- Format/Pattern
- Description (rules to consider while providing values)
- User relationships (whether the entitlement can be associated or dissociated with or from a user)

Provisioning, Detecting Changes, and Post-provisioning

The following provisioning operations must be supported by the connector:

- [User/Entitlements Discovery](#)
- [User Provisioning](#)
- [Entitlement Provisioning](#)
- [Change Detection](#)
- [Post Provisioning](#)

User/Entitlements Discovery

The following operations must be supported by the connector for User discovery:

- Retrieval of user IDs from the resource with filtering
- Retrieval of details of a given user from the resource
- Retrieval of all entitlement IDs from the resource with filtering

User Provisioning

The following provisioning operations must be supported by the connector:

- Add users — Add a new user object to the resource.
- Check for users' existence — Verify that the user exists on the resource.
- Modify users — Modify user attributes on the resource, including changing the value or number (single-valued or multi-valued) of the attribute, removing an attribute, or adding an attribute. If multi-valued attributes are supported by the resource, the connector must support the following modifications to that type of attribute:

- Add one or more attributes
- Remove one or more attributes
- Replace an attribute value with a new value
- Modify the attribute value
- Add one or more sub-values
- Remove one or more sub-values
- Modify entitlements of users — Add one or more entitlements to the user or remove one or more entitlements from the user. The association can be one-way or two-way:
user → entitlement, entitlement → user
This operation can associate a user with an entitlement on the resource or associate an entitlement with a user, or both.
- Get user details — Retrieve the details of a user from the resource.
- Reset password — Change the password of the user to a new password.
- Expire password — Set the password as expired on the resource or set the password of a user as un-expired on the resource.
- Delete user — Delete an existing user from the resource.
- Retrieve all entitlements associated with a user — Retrieve the IDs and types of all entitlements to which a user is assigned. The connector must also support filtering on entitlements to be retrieved.
- Retrieve all users associated with an entitlement — Retrieve the IDs of all users that are assigned to a given entitlement.
- Filter users to be retrieved — Filter the retrieved users based on criteria.
- Disable user — Disable the user on the resource. Following are examples of the result of this operation on the resource:
 - Disable
 - De-activate
 - Revoke account
 - Revoke login access
 - Delete all entitlements
- Enable user — Enable the user on the resource. Following are examples of the result of this operation on the resource:
 - Enable
 - Re-activate
 - Restore account
 - Grant login access
 - Add previously held entitlements back to user

Entitlement Provisioning

The following provisioning operations must be supported by the connector:

- Add entitlement — Add a new entitlement object to the resource.
- Modify entitlement — Modify the entitlement attributes on the resource.
- Check for entitlement existence — Verify the existence of the entitlement on the resource.
- Get entitlement details — Retrieve the details of an entitlement from the resource.
- Delete entitlement — Delete an existing user from the resource.
- Add a child entitlement — Add an entitlement as a child of another entitlement with a link to the parent.
- Delete a child entitlement — Remove a parent-child relationship.
- Get children of an entitlement — Get IDs of all child entitlements of the given entitlement.
- Get parent entitlement — Get the ID of the parent entitlement of the given entitlement.
- Enable entitlement — Enable an entitlement on the resource.
- Disable entitlement — Disable an entitlement on the resource.

Change Detection

Synchronizing Select Identity with identity changes on the resource. This can be implemented in the following ways:

- Detection methods — The connector can support a pull model where the connector implements the **SIChangeLogModel** class. Either Select Identity or a standalone program calls the connector to get the changes from the last call. The connector can also support a push model where the connector or the agent detects changes on the resource and prepares and sends an SPML Web Service request to Select Identity over HTTP or HTTPS.
- Add user — Add a user to the resource.
- Modify user — Detect attribute value, single-value type, or multi-value type changes on the resource, remove an attribute, or add an attribute. If an attribute value change is detected, the connector must capture the new value. Depending on resource support, the connector is required to support multi-valued attribute modifications. If multi-values are supported, the following operations must be supported:
 - Replace the complete multi-value
 - Add one or more sub-values
 - Remove one or more sub-values
- Password changes — Capture the new password of the user.
- Modify entitlements of user — Change the assigned entitlements including adding or removing one or more entitlements.
- Delete user — Delete a user in the resource.
- Disable user — Disable a user in the resource.
- Enable user — Enable a user in the resource.
- Move user — Move a user from one container to another. This is usually not detected as an attribute change. A container could be an organizational unit.

Post Provisioning

This is the support of an interface that is called by the connector after a provisioning operation. The implementation of this interface is independent of the connector.

Internationalization Compliance

All modules of the connector are required to support internationalization (I18N), which enables the complete connector to be localized to any foreign language without code changes. The following must be I18N-compliant:

- If the connector is implemented to use the Attribute Mapping Utility for schema mapping, all strings displayed on the Attribute Mapping Utility console, including attribute names and values
- All messages generated within the connector that are directly propagated to Select Identity or combined with resource messages
- All values for user and entitlement attributes
- All user and entitlement attribute names
- If an agent is implemented, all text displayed on the agent console

Performance and Scalability

The connector must support at least 100 provisioning transactions per minute. One transaction could mean one user addition or modification or deletion, and so on. Likewise, the connector must support at least 100 change-detection transactions per minute.

Regarding scalability, the connector must support one million users and one million entitlements, and is required to scale to support 10,000 resources.

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. The unique identifier can also be a combination of two or more attributes. In such a case, the identity object will be a combination of these attributes. The connector will build/parse this unique key within to address the identity.
 - 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 many 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 Determine 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 “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 length, minimum length, maximum number of alphabetic characters, minimum number of numeric characters, 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 [Implementing a Select Identity Connector](#) on page 43. This procedure provides a general overview.

- 1 Start with a sample application that can provision identity objects and perform entitlement assignments on the identity objects in the resource.
- 2 Implement all of the required Select Identity connector methods to create, read, update, and delete identity objects, leveraging the connector template. The main interface to implement is `SICConnectorInterface`.
- 3 Implement all entitlement association and dissociation methods.
- 4 Pick up all the Log and error strings from a Resource Bundle so that they can be localized.
- 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, deploy the connector Tester Tool for testing the connector.
- 9 Use IDEs for the development and Apache ANT for build tools.
- 10 Use the JDK, J2EE, and third-party libraries for further development.

Integration

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

- 1 Verify that Select Identity is able to look up and use 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

Package the connector as follows:

- 1 Include all libraries required by the connector in a RAR file.
- 2 If you are packaging the JAKARTA project JAR files (`commons-*.jar`), they should be of the same version as being used with Select Identity.
 - ▶ You need not package some of the generic JAR files that are available with Select Identity.
- 3 Test the client for unit testing.
- 4 Determine any schema information (`ddl`, `dml`) needed by the connector.
- 5 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 footprint, and so on
- Requirements on the system administrator to install the agent on the resource
- Administration documents

3 Connector SDK

As of Select Identity 4.0, a connector Software Development Kit (SDK) is included to facilitate connector development. The SDK includes the following modules:

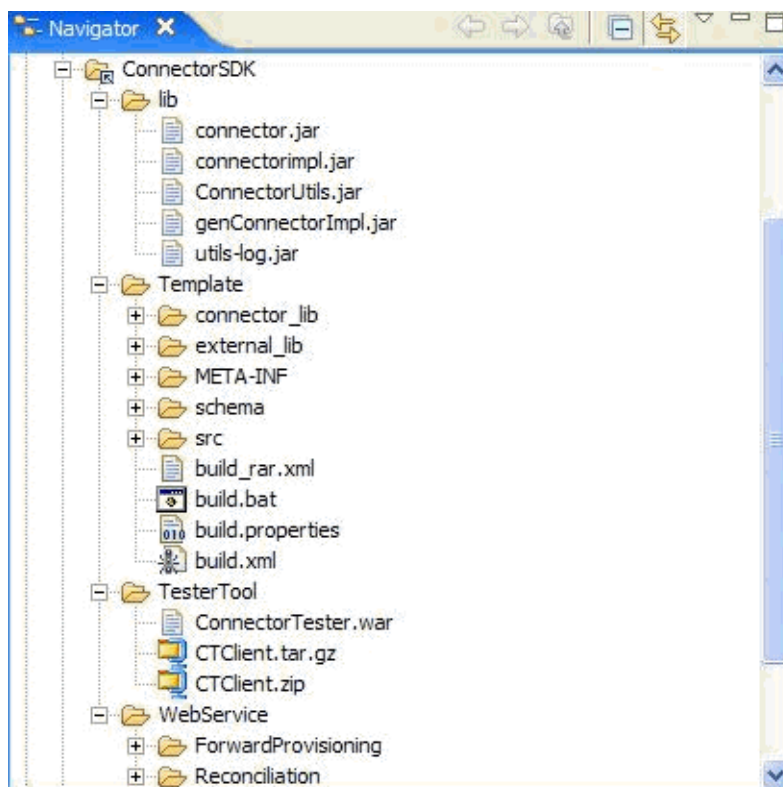
- [Simplified Connector Interface](#)
- [XML Schema Handling](#)
- [Generic JCA Interface and Connector Implementation](#)
- [Connector Tester Tool](#)
- [Connector Template](#)

The SDK provides a generic framework to quickly develop connectors for Select Identity. There is no need to know details of JCA and RAR packaging to use this SDK. Most of the details of the Connector mapping file parsing and interpreting are transparent. You just need to focus on the actual connectivity to the resource, and how best to provision user and entitlement information into it.

The SDK includes a connector template to speed up development, and shows how this template can be customized to build your own connector.

The Connector SDK includes the following folders/files:

Figure 6 Connector SDK Folder/File Structure



Following is a brief description of the contents of the SDK:

Folder	File	Description
Lib		Contains the library Jar files used to develop your connectors.
	Connector.jar	Main Connector interface Jar that includes SConnector interface, SUserModel interface and related interfaces.
	connectorimpl.jar	Implementation classes for SUserModel , XML schema loader and so on.
Lib	genConnectorImpl.jar	Generic Implementation of the SConnector interface along with JCA interface implementation. This provides the simplified connector interface SConnectorInterface .
	utils-log.jar	Some utility classes include common log implementation.
	ConnectorUtils.jar	SPML utility classes.
Template		Contains the connector template. A fully compilable and deployable dummy connector implementation that can be customized to write your own connectors easily. This uses the generic connector implementation and implements the simplified connector interface.
TesterTool		Contains a .war file that includes a tester servlet that can be deployed and used to test any Select Identity connector. Also included is a tar.gz file (for UNIX) that contains the tester tool client and scripts to invoke the tester. Several sample scripts are included that can be used to send all kinds of provisioning operations to the connector.
WebService		This includes sample SPML requests for Web service provisioning into Select Identity. It has two folders for forward provisioning and reconciliation samples.

Simplified Connector Interface

In addition to the existing **SConnector** interface, a simplified interface is introduced in the Select Identity 4.0 connector framework. This can be used to quickly develop Select Identity connectors.

The following diagram illustrates the new architecture.

Figure 7 Simplified Connector Interface Architecture

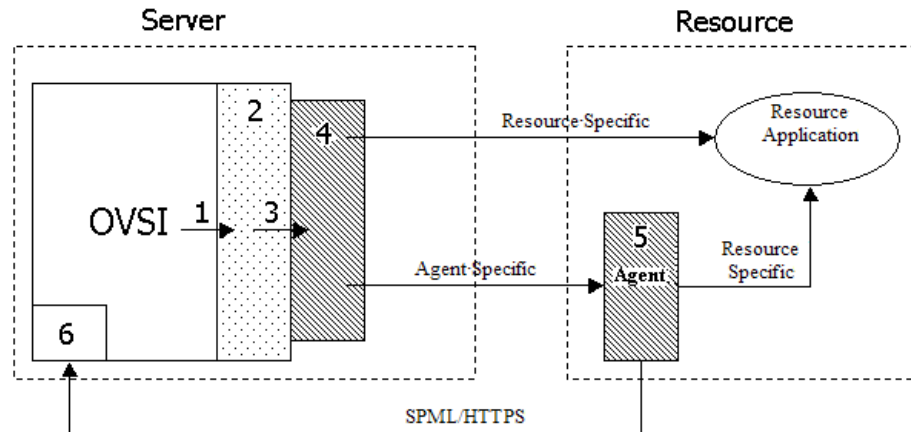


Figure 4 is annotated as follows:

- 1 Select Identity
- 2 Generic Connector Implementation Library
- 3 SConnectorInterface
- 4 Resource-specific part of the connector that communicates directly with the resource or with the agent
- 5 Agent implementation (for agent-based connectors)
- 6 Select Identity Web Service

Highlights of the Simplified Connector interface are described in the following sections.

XML Schema Handling

The XML schema mapping file is pre-processed and resource attributes are provided in the simplified interface. Therefore, you do not need to understand the structure of the XML file and how to parse it.

Advanced users can still access the schema mapping. A java image of the XML file data is passed to the connector implementation and can be used to get details of the mapping.

Generic JCA Interface and Connector Implementation

All the required JCA class/interface implementations are provided. You only need to focus on the resource interface. This eliminates the need to spend time trying to understand JCA architecture and data flow.

A single jar file `genConnectorImpl.jar` is provided with all the required Generic Connector implementation files. The `genConnectorImpl.jar` file needs to be packaged with the connector along with your resource-specific part of the connector.

You also need to implement the simplified interface `SICconnectorInterface`, which contains all provisioning operations.

Connector Tester Tool

The SDK includes a Tester Tool that can be used to test and certify your connector before you deploy it in Select Identity.

After you build the connector, you can use the connector tester tool to test provisioning operations. The tool consists of the following:

- A standalone servlet WAR module, which invokes the connector directly.
- Sample scripts with SPML requests for all provisioning operations.
- A simple HTTP client to send requests to the tester servlet.

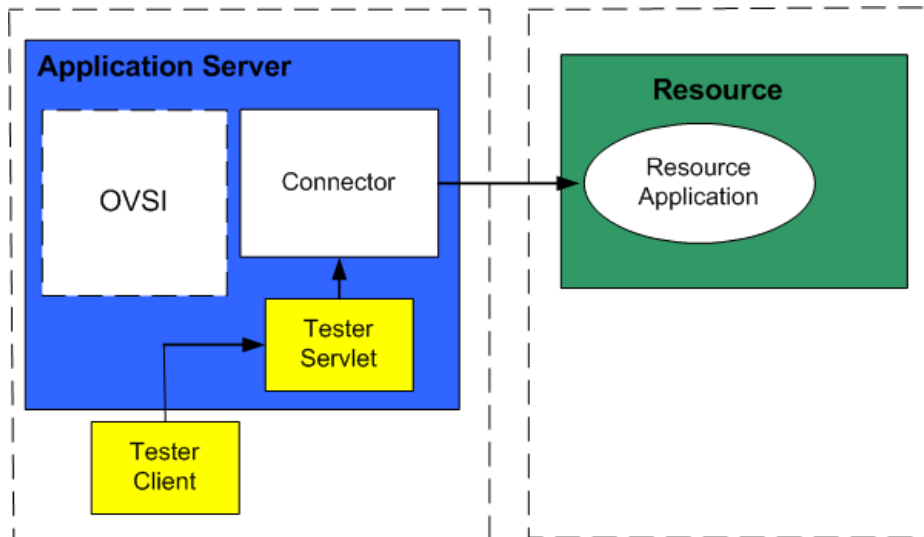
You use the client to run the SPML scripts, which send SPML requests to the servlet. These SPML requests include instructions for forward-provisioning operations. The servlet converts SPML requests to Connector API requests and invokes the connector. The connector then sends the results or errors back to the servlet. All components of the tester tool are deployed or installed on the application server where the Select Identity server and target connector reside.

To use the tester tool, you must deploy a WAR file on the application server (in addition to your connector implementation). Then, you can issue SPML requests to the servlet using the client. The servlet sends the requests to the connector, to verify that it can receive requests and issue responses.

See [Connector Tester Tool](#) on page 83 for details.

The following diagram shows how the Tester Tool is used.

Figure 8 Connector Tester Tool



Some highlights of the Tester Tool are:

- Standalone connector development

The servlet can be deployed in the application server along with the connector and testing can proceed without Select Identity. The tester servlet directly talks to the connector and this gives the advantage of pre-certifying the connector before it is staged to Select Identity.

- Test Scripts

The Tester Tool provides many scripts with SPML requests for all operations that can be done through Select Identity.

- Performance and Scalability tests

The Tester client can be used to perform bulk/iterative operations to regression test the connector. Multiple clients can also be used to drive one connector.

Connector Template

The SDK comes with a connector template named Dummy connector. This is a fully implemented connector and serves as a real example of connector implementation. The only missing part is the resource interaction which is different for different resources.

The Dummy connector implements the simplified interface and can be used as a reference for developing connectors. See [The Connector Template](#) on page 127 for details.

4 Implementing a Select Identity Connector

This chapter describes how to build a connector, including the connector interface methods to be implemented and the requirements of the agent.

This chapter contains the following sections:

- [Development Requirements](#)
- [Steps to Implement a Connector for Select Identity](#)
- [Building a Connector for Forward Provisioning](#)
- [Building a Connector for Reverse Provisioning](#)
- [Mapping Select Identity Attributes to the Resource Schema](#)
- [Some Coding Guidelines](#)

Development 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 SDK that you can use to write your own connectors (see [Connector SDK](#) on page 35 for details). 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>**.

For an overview of the Select Identity Connector APIs, see [Select Identity Connector API Interfaces and Classes](#) on page 14.

When implementing a connector using the J2EE Connector APIs and the Select Identity Connector APIs, 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.
- Connector SDK

Steps to Implement a Connector for Select Identity

The following steps describe how to implement a connector for Select Identity:

- [Step 1: Start with the Connector Template](#)
- [Step 2: Gather Connector Details](#)
- [Step 3: Working with the Connector Template](#)

Step 1: Start with the Connector Template

The connector template and details of files included are explained in [The Connector Template](#) on page 127. This is a starting point to build your own connector for forward provisioning and also for communicating with an agent, in the case of an agent-based connector.

The template includes build files, a property file, library Jar files, an XML mapping file, and a dummy implementation of the `SIConnectorInterface`. You need to gather your connector-specific information to be able to work with the template.

Step 2: Gather Connector Details

Before starting to write your own connector, you must collect the following information:

Resource Details

- Connection parameters

Details needed to access the resource and perform the provisioning operations, such as IP address, port, admin account, password etc.

Along with the names of these parameters, it is required to have further details about these connection parameters such as `displayName`, `default Value`, `min-length`, `max length`, `required or not`, `encryption needed or not`. This information is needed to populate the GUI page that is integrated with SI where you provide values for these connection parameters at the time of deploying the resource.

- Max number of supported connections
- Resource EIS details

Such as `Name`, `Provider`, `Version`. This is needed as part of the RAR definition.

Resource Schema Details

- Name and details of resource user attributes:

Attribute names such as `UserName`, `Email`, `Password`, `CN`, `SN`, `GivenName` and so on.

The resource may support and store attributes in one of many ways. Following 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, the 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.

Also needed are further details about each attribute such as: default value, display name, min-length, max-length, required or not.

- Entitlements supported

Connector Code-Related Details

- Connector Name
- Connector Short Name — Used as a prefix for re-naming the template files
- Java Package name

Step 3: Working with the Connector Template

Once you have the details on developing the connector, as listed in the above section, you can use the Connector Template to customize and build your connector.

Following are some of the main steps involved. See each step for details:

1. Prepare the Connector Template Files
2. Check the Library JAR Files
3. Implement the Connector

1. Prepare the Connector Template Files

Complete the following steps to prepare the Connector Template files:

- Create folders for the connector template files.

We will assume the following sample values for the connector parameters:

Parameter Name	Parameter Value	Description
Connector <code>src</code> Folder	MainDir	This is the main folder with all the connector-related files.
Connector Name	My Sample Connector	A descriptive name for the connector.
Connector Short Name	com.my.sample	
Connection Parameters	hostName port userName password	

Parameter Name	Parameter Value	Description
Max # of connections	10	
Resource User Attributes	userId password directory firstName lastName middleName fullName department	userId is the primary Key in the resource.
Resource Entitlement ID	groupId	groupId is the primary Key in the resource.

- Create a folder by the name `MyConnector` and copy the connector template files under the name `dummy` into this new folder.
- Create folders `MainDir/src/com/my/sample` and move the files under `src/com/hp/Select Identity/connector/dummy` to this folder.
- Go to `MainDir/src/com/my/sample` and rename the files `DummyConnector.java` to `MyConnector.java` and `DummyParamResources.properties` to `MyConnectorParamResources.properties`.

b Edit the Connection Properties file.

- Edit `MyConnectorParamResources.properties` so that the connection parameters are defined according to your resource requirements.

Each connection parameter has some properties associated with it as shown below for the "hostName" connection parameter:

```
hostName-displayName=Host Name
hostName-defaultValue=MyHost
hostName-helpString=Host name of the server
hostName-minLength=1
hostName-maxLength=80
hostName-pattern=[.]+
hostName-required=false
hostName-tipString=
hostName-type=java.lang.String
hostName-encryptValue=false
```

- Provide the details for all connection parameters in this file.
- Finally, set the order of the connection parameters to appear in the property `paramOrder` at the end of this file.

Each connection parameter is separated by commas. This is the order in which `Select Identity` shows the connection parameters when the resource is being deployed/modified/viewed.

c Edit the Connector Implementation file.

The connector main code is to be started from `MyConnector.java` which is explained in step 3. [Implement the Connector](#) on page 49. Edit this file to change all occurrences of the word `DummyConnector` in this file to `MyConnector`.

Change the package name to `com.my.sample`.

d Edit the XML Mapping file.

Go to `MainDir/schema`, and rename `DummyConnectorMapping.xml` to `MyConnectorMapping.xml`.

This file contains the mappings of user and entitlement attributes from the Select Identity model to the resource schema. For example, if you called an attribute `userid` in the resource, you may have called it **User Name** in Select Identity. This mapping must be given even if the names are the same.

As explained in [Resource Schema Details](#) on page 44, there may not be any physical attribute on the resource that you can map to, but just have an API. In such cases, you still need to come up with a set of logical attributes for the user and use this mapping file to map onto these logical attributes. In your connector code, you take these logical attribute names and provision the user with their values.

Each user attribute mapping is specified in the `objectClassDefinition` with name "SIUser", using an `attributeDefinitionReference` shown as follows:

```
<attributeDefinitionReference name="User Name" required="true"
concero:init="true" concero:tafield="User Name"
concero:resfield="userName" concero:isKey="true" />
```

The above line in the XML mapping file has the following meanings:

Mapping Name	Description
<code>required</code>	Specifies whether this attribute is required or an optional attribute. The values it can take are "true", "false".
<code>concero:init</code>	Specifies whether this attribute is needed during the creation of the user. "true" means this is needed, "false" means not needed.
<code>concero:tafield</code>	Name of Select Identity attribute.
<code>concero:resfield</code>	Name of the resource attribute or the logical attribute.
<code>concero:isKey</code>	Specifies whether this is the identifying attribute of the user in the resource. This can be "true" only on one mapping.

You can have a combination of Select Identity attributes mapped onto the same resource attribute. In this case, the `tafield` will have a combination. Following is another example:

```
<attributeDefinitionReference name="Full Name" required="false"
concero:tafield="[First Name] [Middle Name] [Last Name]"
concero:resfield="fullName"/>
```

In this example, you enclose the Select Identity attribute names in square brackets to prepare the combination.

e Edit the `ra.xml` file.

Go to the `MainDir/META-INF` folder and edit the file `ra.xml`, to change the Resource adapter-specific parameters. Change the following:

Change	To
<code>display-name value</code>	My Sample Connector
<code>vendor-name value</code>	Name of your company
<code>eis-type value</code>	Type of the Resource EIS. Example UNIX box, My Application, and so on
<code>Version value</code>	Resource EIS version
<code>eis/DummyConnector-ParamFactory</code>	"eis/MyConnector-ParamFactory"
<code>com.hp.ovsi.connector.dummy.DummyConnector</code>	"com.my.sample.MyConnector"
<code>com/hp/ovsi/connector/dummy/DummyParamResources.properties</code>	"com/my/sample/MyConnectorParamResources.properties"

f Edit the `weblogic-ra.xml` file.

If you are going to use BEA WebLogic to deploy this connector, go to `MainDir/META-INF`, and edit the file named `weblogic-ra.xml`:

<code>eis/DummyConnector</code>	"eis/MyConnector"
<code>initial-capacity value</code>	0
	This is the number of connections to be open when the connector is deployed. Typically this is 0 and increased upon demand.

g Edit the `build.properties` file.

Edit the `build.properties` file to enter all the packaging-related information:

<code>connector.build.dir</code>	/tmp	This is the folder where you would like the build files to be placed.
<code>connector.pkg</code>	com/my/sample	
<code>connector.rar.file</code>	MyConnector.rar	
<code>schema.jar.file</code>	MyConnectorSchema.jar	
<code>connection.params.props.file</code>	MyConnectorParamResources.properties	
<code>connector.name</code>	My Sample Connector	
<code>connector.version</code>	1.0.1	

2. Check the Library JAR Files

Make sure you have the latest versions of the jar files under the following folders:

- `connector_lib`

- external_lib

3. Implement the Connector

You start with writing code in `MyConnector.java`.

The files as they are prepared in the previous section should be ready to compile. First, make sure you can compile the source using the Apache ANT tool for the `build.xml` file in the `MainDir` directory and see if you can output the following files:

- `MyConnector.rar`
- `MyConnectorSchema.jar`
- Verify that `MyConnector.rar` contains `MyConnector.class`, `ra.xml`, `weblogic-ra.xml`.
- Verify that `MyConnectorSchema.jar` contains `MyConnectorMapping.xml`.

Sample SPML Requests for Reconciliation

There are two types of SPML requests based on the type of resource. The resource could be an authoritative resource or non-authoritative.

Authoritative Reconciliation SPML Requests

This section shows sample authoritative reconciliation SPML requests for the following actions:

- Add user
- Delete user
- Modify user

Add User

Following is a sample SPML request for an auth-add reconciliation request:

Note the following:

- **keyFields must be** `<resourceName>_KEY`
- **all attribute names are Select Identity resource attribute names or the names given in the `concero:tafield` in the XML mapping file**

```

    <addRequest requestID="12345"
execution="urn:oasis:names:tc:SPML:1:0#asynchronous">
    <operationalAttributes>
    <attr
name="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
        <value>sis</value>
    </attr>
    <attr name="urn:trologica:concero:2.0#password">
        <value>abc123</value>
    </attr>
    <attr name="urn:trologica:concero:2.0#resourceId">
        <value>LDAP71</value>
    </attr>

```

```

    <attr name="urn:trologica:concerro:2.0#keyFields">
      <value>LDAP71_KEY</value>
    </attr>
    <attr name="urn:trologica:concerro:2.0#reverseSync">
      <value>true</value>
    </attr>
    <attr name="urn:trologica:concerro:2.0#taUserName">
      <value>WSu7301</value>
    </attr>
    <attr name="urn:trologica:concerro:2.0#taResourceKey">
      <value>WSu7301</value>
    </attr>
  </operationalAttributes>

  <attributes>
    <attr name="UserName">
      <value>WSu7301</value>
    </attr>
    <attr name="Email">
      <value>user.email@hp.com</value>
    </attr>
    <attr name="State">
      <value>TX</value>
    </attr>
    <attr name="FirstName">
      <value>Abigail</value>
    </attr>
    <attr name="LastName">
      <value>Anderson</value>
    </attr>
    <attr name="Employee ID">
      <value>HP</value>
    </attr>
    <attr name="Zip">
      <value>75000</value>
    </attr>
  </attributes>
</addRequest>

```

Delete User

Following is a sample SPML request for auth-delete recon request:

Note the following:

- **keyFields must be** <resourceName>_KEY
- **resource key value of the user is given in the identifier section of the SPML request**

```

  <deleteRequest requestID="12345"
  execution="urn:oasis:names:tc:SPML:1:0#asynchronous">
    <operationalAttributes>
      <attr
  name="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
        <value>sis</value>
      </attr>
      <attr name="urn:trologica:concerro:2.0#password">
        <value>abc123</value>

```

```

        </attr>
        <attr name="urn:trologica:concerro:2.0#resourceId">
            <value>LDAP71</value>
        </attr>
        <attr name="urn:trologica:concerro:2.0#keyFields">
            <value>LDAP71_KEY</value>
        </attr>
        <attr name="urn:trologica:concerro:2.0#reverseSync">
            <value>true</value>
        </attr>
    </operationalAttributes>

    <identifier xmlns=""
type="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
        <id>WSu7224</id>
    </identifier>
</deleteRequest>

```

Modify User

Following is a sample SPML request for auth-delete recon request:

Note the following:

- **keyFields must be** <resourceName>_KEY
- **resource key value of the user is given in the identifier section of the SPML request**
- **all attribute names are SI resource attribute names or the names given in the concerro:tafield in the XML mapping file**
- **only the mapped attributes must be passed in the request**

```

    <modifyRequest requestID="12345"
execution="urn:oasis:names:tc:SPML:1:0#asynchronous">
        <operationalAttributes>
            <attr
name="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
                <value>sis</value>
            </attr>
            <attr name="urn:trologica:concerro:2.0#password">
                <value>abc123</value>
            </attr>
            <attr name="urn:trologica:concerro:2.0#resourceId">
                <value>LDAP71</value>
            </attr>
            <attr name="urn:trologica:concerro:2.0#keyFields">
                <value>LDAP71_KEY</value>
            </attr>
            <attr name="urn:trologica:concerro:2.0#reverseSync">
                <value>true</value>
            </attr>
        </operationalAttributes>

        <identifier xmlns=""
type="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
            <id>WSu7212</id>
        </identifier>
    </modifyRequest>

```

```

</identifier>

<modifications>
  <modification name="FirstName" operation="replace">
    <value>ANNA</value>
  </modification>
  <modification name="LastName" operation="replace">
    <value>ALENDALE</value>
  </modification>
  <modification name="Address 1" operation="replace">
    <value>1525 EAST GATE DRIVE, WITCHITA</value>
  </modification>
  <modification name="Zip" operation="replace">
    <value>62005</value>
  </modification>
  <modification name="State" operation="replace">
    <value>KS</value>
  </modification>
</modifications>
</modifyRequest>

```

Non-Authoritative Reconciliation SPML Requests

Following are sample requests from a non-authoritative resource. A non-authoritative resource can only send changes to user entitlements in Select Identity or delete service membership.

This section shows sample non-authoritative reconciliation SPML requests for the following actions:

- Add user entitlements
- Delete service membership
- Change user entitlements

Add User Entitlements

Following is a request to add entitlements:

Note the following:

- `keyFields` must be `<resourceName>_KEY`
- resource key value of the user is given in the identifier section of the SPML request

```

<addRequest requestID="12345"
execution="urn:oasis:names:tc:SPML:1:0#asynchronous">
  <operationalAttributes>
    <attr
name="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
      <value>sis</value>
    </attr>
    <attr name="urn:trologica:concerro:2.0#password">
      <value>abc123</value>
    </attr>
    <attr name="urn:trologica:concerro:2.0#resourceId">
      <value>LDAP70</value>
    </attr>

```

```

        <attr name="urn:trologica:concerro:2.0#keyFields">
            <value>LDAP70_KEY</value>
        </attr>
        <attr name="urn:trologica:concerro:2.0#reverseSync">
            <value>true</value>
        </attr>
        <attr name="urn:trologica:concerro:2.0#taResourceKey">
            <value>WSu7221</value>
        </attr>
    </operationalAttributes>

    <identifier xmlns=""
type="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
        <id>WSu7221</id>
    </identifier>

    <attributes>
        <attr name="urn:trologica:concerro:2.0#groups">
            <value>HR Managers</value>
            <value>PD Managers</value>
        </attr>
    </attributes>
</addRequest>

```

Delete Service Membership

Note the following:

- keyFields **must be** <resourceName>_KEY
- resource key value of the user is given in the identifier section of the SPML request

```

<deleteRequest requestID="12345"
execution="urn:oasis:names:tc:SPML:1:0#asynchronous">
    <operationalAttributes>
        <attr
name="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
            <value>sis</value>
        </attr>
        <attr name="urn:trologica:concerro:2.0#password">
            <value>abc123</value>
        </attr>
        <attr name="urn:trologica:concerro:2.0#resourceId">
            <value>LDAP70</value>
        </attr>
        <attr name="urn:trologica:concerro:2.0#keyFields">
            <value>LDAP70_KEY</value>
        </attr>
        <attr name="urn:trologica:concerro:2.0#reverseSync">
            <value>true</value>
        </attr>
    </operationalAttributes>

    <identifier xmlns=""
type="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
        <id>WSu7235</id>
    </identifier>

```

```

        </identifier>
    </deleteRequest>

```

Change User Entitlements

Note the following:

- keyFields **must be** <resourceName>_KEY
- resource key value of the user is given in the identifier section of the SPML request

```

    <modifyRequest requestID="12345"
execution="urn:oasis:names:tc:SPML:1:0#asynchronous">
    <operationalAttributes>
        <attr
name="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
            <value>sis</value>
        </attr>
        <attr name="urn:trologica:concerro:2.0#password">
            <value>abc123</value>
        </attr>
        <attr name="urn:trologica:concerro:2.0#resourceId">
            <value>LDAP70</value>
        </attr>
        <attr name="urn:trologica:concerro:2.0#keyFields">
            <value>LDAP70_KEY</value>
        </attr>
        <attr name="urn:trologica:concerro:2.0#reverseSync">
            <value>true</value>
        </attr>
    </operationalAttributes>

    <identifier xmlns=""
type="urn:oasis:names:tc:SPML:1:0#UserIDAndOrDomainName">
        <id>WSu7233</id>
    </identifier>

    <modifications>
        <modification name='urn:trologica:concerro:2.0#groups'
operation='delete'>
            <value>HR Managers</value>
        </modification>

        <modification name='urn:trologica:concerro:2.0#groups'
operation='add'>
            <value>$UNIX2</value>
        </modification>
    </modifications>

</modifyRequest>

```

Building a Connector for Forward Provisioning

Forward provisioning is the process of provisioning users and their entitlements on the resource. To build a connector that performs this function, implement an agent-less or agent-based connector, as described in the following sections.

Agent-less Forward Provisioning

This section describes the required methods of implementing `SIConnectorInterface`. This part is common for both agent-based and agent-less implementations. The only difference is that for agent-based implementation, you implement the methods so that you communicate with your agent, where as in agent-less implementation you need to directly provision into the resource.

To build an agent-less connector, follow these guidelines:

- 1 Implement a connector for Select Identity as described in [Steps to Implement a Connector for Select Identity](#) on page 44.
- 2 Implement the `SIConnectorInterface`.

The generated template includes a skeleton implementation of `SIConnectorInterface`. You must expand each of the methods in the generated template to communicate with the resource and perform the provisioning process.

You *must* implement the following methods:

- `doTest()`

Purpose: Performs a connectivity test.

Usage: This is called when a new resource is deployed, or an existing resource is modified in Select Identity and the user clicks **Test and Submit**.

Implementation: The connection parameters bean instance called `mConInfoBean` contains values for all parameters. Using this connection information, try to establish a connection to the resource and validate the connection information.

- `isUserExists(String keyField, String keyValue, boolean keyExistsFlg)`

Purpose: Checks for the existence of a user.

Usage: This method is used to check if a user is present in the resource. The `keyField` argument identifies the attribute that is marked as key in the attribute mapping and `keyValue` contains the value of this key field.

In most cases, the key field of a user in Select Identity is the same as the key field of the user in the resource. However, in certain cases, they might be different and sometimes indeterminate until the user is actually created in the resource.

The `createUser()` method should get the user's key field value after the user is created in the resource and return it the value to the caller (and eventually Select Identity). In this case, the `keyExistsFlg` argument will be true, which means that the user was created earlier, after an initial successful user creation. You can use this to determine if this method is being called for the first time or to verify if the user exists in the resource.

Implementation: Using the key value to check if the user exists in the resource and return the result.

- `createUser(String keyField, String keyValue, String passwd, Map attrMap)`

Purpose: Creates a user.

Usage: This method is called to add a user to the resource.

Implementation: Create the user account in the resource using the given key value, password, and attribute name-value pairs. All attribute name-value pairs are passed in the `attrMap` argument. The name of the attribute is the resource attribute name as specified in the mapping file. The value is in the form of `Object[]` representing multiple sub-values of the attribute, if any. For single-valued attributes, only the first element of the array is populated.

The following sample code can be used to read attribute values:

```
String lAttrId = null;
Object[] lAttrValue = null;
Iterator lIter = attrMap.keySet().iterator();
while (lIter.hasNext())
{
    lAttrId = (String) lIter.next();
    lAttrValue = (Object[]) attrMap.get(lAttrId);
    // ...
}
```

Use the `lAttrValue` to set the value of the attribute for creation. If the resource supports multi-valued attributes, use the `Object[]` to set all of the multiple values. If not, you may either set the first one in the array or combine them into a single value.

After a user is successfully created, the effective key value of the user should be returned. This may be the same as the one passed in by `Select Identity`. `Select Identity` will save it and pass it back to the connector for subsequent operations on the same user.

- `updateUser(String keyField, String keyValue, Map attrMap)`



The `keyValue` argument that is passed will be the new key value that is returned if the user is successfully created.

Purpose: Updates a user.

Usage: This method is called when user's attributes are modified.

Implementation: The attributes of the user to be modified are given in `attrMap` with values as instances of `TAAAttrValueBean`. The **Operation** field in the `TAAAttrValueBean` defines what must be done on each of the attributes passed in the `attrMap`: `replace`, `add`, `delete`, or `modify`, where `modify` could mean the addition or deletion of sub-values.

Details of `TAAAttrValueBean` for a user modify operation:

The following examples highlight the user modify operation and detail of the contents of `TAAAttrValueBean` passed in the above method

The following abbreviations are used in the examples provided in the tables:

`TAAAttrValueBean=av`

`TAAAttrMemberValueBean=amv`

Example 1: User Modify

User Modify Operation Details

Attribute a1 changed its value to a new value

Attribute a2 has a new subvalue

Attribute a3 has not changed

Attribute a4 has a new attribute added

Attribute a5 is deleted

Contents of TAttrValueBean

a1 - av=[OP_REPLACE, List=[{--, a1sv2}]]
(a1sv2 is the new value.)

a2 - av=[OP_MODIFY, List=[amv={OP_ADD, a2mv3}]]
(This adds subvalue a2mv3 to the two subvalues.)

a3 - av=[OP_NOCHANGE, List=[amv={---, a3sv1}]]
(only given for required fields)

a4 - av=[OP_ADD, List=[amv={---, a4sv1}]]
(This adds a new attribute.)

a5 - av=[OP_DELETE, List=null]
(An attribute in Select Identity is deleted.)

Example 2: User Modify

User Modify Operation Details

Attribute a1 is deleted

Attribute a2 has changed its value to a new value

Attribute a3 is nullified

Attribute a4 is emptied

Contents of TAttrValueBean

a1 - av=[OP_DELETE, List=null]

a2 - av=[OP_REPLACE, List=[amv={---, a2sv4}]]
(The value changed to a single subvalue.)

a3 - av=[OP_REPLACE, List=[amv=null]]
(only given for required fields)

a4 - av=[OP_REPLACE, List=[amv={---, ""}]]
(Use "" as a way to empty the value.)

Please note that all the attributes marked in the Service are required and are sent no matter if their value has changed. If the value has not changed, the contents of TAttrValueBean for those attributes will have an operation type of TAttrValueBean.OP_NOCHANGE.

Exception: Throw `ObjectNotFoundException` if the user does not exist in the resource. For all other error conditions, throw `TConnectorException` with the appropriate resource-specific error message.

- `deleteUser(String keyField, String keyValue)`



The `keyValue` argument that is passed will be the new key value that is returned if the user is successfully created.

Purpose: Deletes a user.

Usage: This is called when a user is terminated.

Implementation: Perform the delete operation on the resource.

Exception: Throw `ObjectNotFoundException` if the user does not exist in the resource. For all other error conditions, throw `TACConnectorException` with the appropriate resource-specific error message.

- `setUserStatus(String keyField, String keyValue, int status)`
 - ▶ The *keyValue* argument that is passed will be the new key value that is returned if the user is successfully created.

Purpose: Enables or disables a user.

Usage: This is called when a Disable All Services or Enable All Services operation is performed using the Select Identity console, with the value of status given as `TACConnector.DISABLED` or `TACConnector.ENABLED` respectively.

Implementation: Depending on the resource support, the user account is required to be disabled or enabled, which also might mean performing lock or unlock operations or revoke or restore operations.

Exception: Throw `ObjectNotFoundException` if the user does not exist in the resource. For all other error conditions, throw `TACConnectorException` with the appropriate resource-specific error message.

- `findUser(String keyField, String keyValue, Map attrMap, boolean keyExistsFlg)`
 - ▶ The *keyValue* argument that is passed will be the new key value that is returned if the user is successfully created.

Purpose: Gets user details.

Usage: This is an optional method and Select Identity does not call on this directly. However, it is required using user discovery phase where all users are retrieved from the resource to synchronize with the Select Identity database. In this case, this method is used to get all attributes of the user from the resource.

Implementation: Get all attribute values from the resource for the user, if the user exists. The value of the attribute should be an instance of `TAAttrValueBean`.

Exception: Throw `ObjectNotFoundException` if the user does not exist in the resource. For all other error conditions, throw `TACConnectorException` with the appropriate resource-specific error message.

- `resetPassword(String keyField, String keyValue, String passwd)`
 - ▶ The *keyValue* argument that is passed will be the new key value that is returned if the user is successfully created.

Purpose: Resets a user's password.

Usage: This method is called when the user's password is to be changed to a new password.

Implementation: Use the new password passed in by the `passwd` argument to change the user's password in the resource.

Exception: Throw `ObjectNotFoundException` if the user does not exist in the resource. For all other error conditions, throw `TACConnectorException` with the appropriate resource-specific error message.

- `expirePassword(String keyField, String keyValue, boolean flg)`
 - ▶ The `keyValue` argument that is passed will be the new key value that is returned if the user is successfully created.

Purpose: Expires or unexpires a user's password.

Usage: This is called when a user's password is required to be expired or unexpired.

Implementation: If the value of `flg` is true, expire the user's password. After this operation is successful, the user should not be able to use the existing password to log in to the resource. If the value of `flg` is false, unexpire the password.

This method is different from `resetPassword()` in that the old password is still valid on the resource but the user will not be able to use it.

Exception: Throw `ObjectNotFoundException` if the user does not exist in the resource. For all other error conditions, throw `TACConnectorException` with the appropriate resource-specific error message.

- `getAllUsers(String keyField, TAQuery query)`

Purpose: Gets all users in a resource.

Usage: This method is not called by Select Identity but is useful when the connector is used to get all users in the resource for user discovery.

Implementation: Get the key values (IDs) of all users in the resource that match the given search criteria in the query argument.

- `getUsers(String userKeyField, String entKeyField, String entType, String entKeyValue)`

Purpose: Gets all users with a specified entitlement.

Usage: This method is not called by Select Identity for general provisioning but is used for reporting purposes when all entitlements of a user are required to be reported.

Implementation: Using the `entKeyField`, `entType`, and `entKey` values, locate the entitlement in the resource and return IDs of all users that are assigned to this entitlement.

- `getEntitlementTypes()`

Purpose: Gets all additional entitlement types.

Usage: This method is called during the initial deployment of the resource in Select Identity and is useful only when the connector and resource support multiple types of entitlements, such as groups, roles, entitlements, privileges, ACLs, and responsibilities. If the connector and resource support only one kind of entitlement, this method is optional and the default entitlement type is used.

Implementation: Return all additional types in the form of `String` instances.

- `getAllEntitlements(String keyField, TAQuery query)`

Purpose: Gets all entitlements in the resource.

Usage: The following uses are supported:

- **Retrieve all entitlements in the resource**
This is the normal usage of this method, where VSI calls on the connector to retrieve all entitlements before provisioning a user. These entitlements are assigned to the user after a successful create operation.

The value and operation fields of the `TAFilter` elements in `TAQuery` are used to enforce a search criteria. The name field specifies the entitlement type to indicate retrieval of all entitlements of the given type.

– **Validation of a given entitlement**

This is used when Select Identity tries to verify that a given entitlement is present in the resource. The value field of `TAFilter` element in `TAQuery` identifies the entitlement and operation field is `EQUALITY`.

– **Retrieval of all possible values of a given attribute**

This is useful if you want to provide a list of possible values to any attribute during user provisioning. Generally, the value is entered but giving a choice restricts invalid values. This is done by configuring the attribute in Select Identity with a Search Connector function. The name field in the `TAFilter` contains the name of the attribute.

Implementation: If the resource does not support entitlements, this is an optional method and should return an empty collection.

Use the `TAQuery` parameter to identify the usage of this method and retrieve all values from the resource. In some cases, the resource supports a primary or default entitlement on a user, which is set automatically when the user is created in the resource. The connector should filter and not return such entitlements to Select Identity. Such entitlements can be managed through separate attributes on the user rather than having them as entitlements.

- `getEntitlements(String userKeyField, String userKeyValue, String entKeyField)`

Purpose: Gets all entitlements of a user.

Usage: This method is not called by Select Identity for general provisioning but is used when all entitlements of a given user are required to be reported.

Implementation: Return a Collection of `EntitlementModel` instances.

Exception: Throw `ObjectNotFoundException` if the user does not exist in the resource. For all other error conditions, throw `TACConnectorException` with the appropriate resource-specific error message.

- `link(String userKeyField, String userKeyValue, String entKeyField, String entType, List entIds)`

Purpose: Assigns entitlements to a user.

Usage: This method is called when trying to assign entitlements to a user that already exists in the resource. It is required if entitlements are supported by the connector and resource.

Implementation: Assigning an entitlement to a user might mean adding the user to the entitlement or adding the entitlement to the user (or both). These can be different depending on the resource. Once the user is assigned to the given entitlement, the user gets the underlying privilege or authority when he or she accesses the resource.

The connector should ignore the error condition arising from the situation where the entitlement is already assigned to the user. It can simply log the error and not throw an exception to Select Identity.

Exception: Throw `ObjectNotFoundException` if the user does not exist in the resource. For all other error conditions, throw `TACConnectorException` with the appropriate resource-specific error message.

- `unlink(String userKeyField, String userKeyValue, String entKeyField, String entType, List entIds)`

Purpose: Unassigns entitlements from the user.

Usage: This method is called when trying to unassign entitlements from a user that exists in the resource and is required if entitlements are supported by the connector and resource.

The connector should ignore the error condition arising from the situation where the entitlement is already unassigned from the user. It can simply log the error and not throw an exception to `Select Identity`.

Implementation: This is the reverse operation of assigning entitlements to user.

3 Package the connector.

You can use the Apache ANT build scripts and deployment descriptors that are provided with the generated templates. Run Apache ANT using the `build.xml` script. The following are the files that are generated by this script:

- RAR (resource adaptor archive) file, which is a deployable module that contains all of the classes and library JAR files for the connector.
- Schema (JAR) file, which contains the XML attribute mapping file and any XSL file, for reverse mapping, to be used with the connector. The contents of this file must be extracted into a folder that is in the class path of the application server.

Agent-based Forward Provisioning

Developing a connector that uses an agent for forward provisioning involves developing an agent that resides on the resource platform. The agent communicates with the resource application for all forward provisioning operations.

The following are some of the general requirements for an agent to perform provisioning:

- Implementing a connector for `Select Identity`— See [Steps to Implement a Connector for Select Identity](#) on page 44 for details.
- Provisioning users and entitlements — The agent is required to support provisioning of users and their entitlements on the resource.
- Availability — The agent must be a constantly running daemon or process so that it can accept and process provisioning requests from the client-side of the connector.
- Scalability — The agent must support a large number of requests.
- Install and Uninstall — The agent must be cleanly installable and uninstallable. This is particularly important if the agent is deployed on a large number of resource platforms.
- Security and Reliability — Communication between the client-side of the connector and agent must be over a secure channel.

The client side of the connector is the JCA side that implements **`SIConnectorInterface`**. The implementation should prepare and send requests for all forward provisioning operations to the agent residing on the resource platform and handle responses.

The agent should handle all requests by performing provisioning operations on the resource application and return the results or error cases.

Building a Connector for Reverse Provisioning

Reverse provisioning is the process of synchronizing changes that occurred in the resource with Select Identity. Changes can include user additions, modifications, deletions, password resets, entitlement changes, and so on. The connector can implement this as described in the following sections.

Agentless Reverse Provisioning

When an agent is not implemented to detect changes on the resource, the connector should be able to poll the resource for changes. You must implement the **SICConnectorInterface** interface and the `getChangeLog(ChangeLogCursor cursor)` method. The purpose of this method is to receive a change that occurred in the resource. Select Identity can be configured to poll any connector to detect changes in the resource. When configured this way, Select Identity calls this method with the appropriate value of the cursor.

To implement `getChangeLog`, the connector must detect changes that occurred in the resource and prepare a data structure (**SICChangeLogModel**) with the change log entries. This instance can hold all changes that occurred in one particular polling interval in the `ChangeLogEntry` instances. As a checkpoint of the last change detected, a cursor is maintained on the resource so that the connector's next call only retrieves changes made since the last call. This checkpoint could be a timestamp or some sequence number (for example highest USN). Select Identity will store this value and give it back to the connector when it calls for the next time.

Implementing `getChangeLog(ChangeLogCursor)`

This method should check the resource for all changes that occurred after the previous call to this method and must prepare an instance of **SICChangeLogModel** with the details of these changes.

SICChangeLogModel represents the changes that occurred in the resource, in a normalized format. Any resource-specific API return values or format returned must be parsed and converted into an instance of this class. This class contains the following main methods:

- `setCursor(ChangeLogCursor)`

The new value of the cursor must be set in `SICChangeLogModel`. A cursor identifies a checkpoint in the resource change log so that a next call to `getChangeLog()` will read the changes past this checkpoint.

The cursor class contains an `int` member to represent the changelog number. The `changelog` number is determined by the resource and/or how you detect changes occurred in the resource. The resource might already be using a number like this, for example the USN number in case of Active Directory. This change log number could also be a derivation of timestamp on the change that occurred in the resource. For example, it could be a number of seconds since Jan 1 1970 and an `int` is good till the end of the year 2036.

Use this integer value on the `ChangeLogCursor.setNumber(int)` method.

- `addCLEntry(ChangeLogEntry)`

One instance of `SICChangeLogModel` can contain multiple instances of `ChangeLogEntry` instances, which represents each change. A change could be that a user is added, modified, deleted.

Internally, **SICChangeLogModel** uses an `ArrayList` as a collection of `ChangeLogEntry` instances, and so the order of the entries is maintained when the changes are sent to Select Identity for reconciliation.

`ChangeLogEntry` has the following methods:

- `setUserId(String)`

This is used to set the id of the user in the resource.

- `setChangeType(int)`

This is to set the type of change that occurred in the resource. The possible types are as follows::

```
ChangeLogEntry.USER_ADDED
ChangeLogEntry.USER_MODIFIED
ChangeLogEntry.USER_DELETED
ChangeLogEntry.USER_ENABLED
ChangeLogEntry.USER_DISABLED
ChangeLogEntry.USER_RESET_PASSWD
addAttrEntry(ChangeLogAttribute)
```

`setChangeType(int)` is used to add the attribute value in the change. This contains the id and value of the attribute. The id should represent the Select Identity id of the attribute and not the physical resource attribute. If these are different, re-mapping of the name must be done.

As an illustration, let us say you have the forward mapping of attribute as follows:

Select Identity Resource Attribute	Physical Resource Attribute
UserName	uid
Email	mail
FirstName	givenName
LastName	Sn

The name on the left side (Select Identity resource attribute) is the attribute that Select Identity gives to the connector in `SIUserModel` during forward provisioning. The name on the right is the name of the attribute in the resource that you must set on the resource with the given value.

Now, when you report the attributes in `ChangeLogAttribute`, the change detection on the resource will give you the physical resource attribute name, such as `mail` or `givenName`, and this must be converted back to `Email` or `FirstName`.

If you are using XML mapping file for forward provisioning, the connector framework provides you access to the Java representation of this mapping in the form of an instance of `TASchema` class, by calling the `setSchema(TASchema)` method on **SICConnectorInterface**. The default implementation of this method is to store this instance in a member variable of the **SICConnectorInterface** implementation, and this can be used for reverse mapping of the attribute names. Here is an example of how to use this.

Assume we have the following definition:

```
Private TASchema mTaSchema;
```

Then, in `getChangeLog()` implementation you could access the following:

```
TASchemaParamBean[] lBeans = mTaSchema.getUserSchema();  
for (int ii=0; ii<lBeans.length; ii++)  
{  
    TASchemaParamBean lBean = lBeans[ii];  
    lBean.getMappingField() ... // gives the SIResourceAttribute (or  
    tafield in XML file)  
    lBean.getResField() ... // gives the Physical resource attribute (or  
    resfield)
```

Standard name for entitlement identifiers (IDs):

When you are reporting `ChangeLogAttribute` for entitlements, you use the following keyword to represent an entitlement:

```
urn:trulogica:concerro:2.0#groups
```

Agent-based Reverse Provisioning

The agent is notified of changes that occurred in the resource and notifies Select Identity with all detected changes. The agent notifies Select Identity using SPML requests sent over HTTP or HTTPS to the Select Identity Web Service URL. Select Identity consumes these changes by processing them for reconciliation.

In addition to the requirements mentioned in [Agent-based Forward Provisioning](#) on page 61, the following are also required:

- **Configuration** — The agent's operational parameters for the SPML requests, the Select Identity Web Service URL, resource identification, and response handling must be configurable.
- **Response handling** — The agent should be able to handle the response to the SPML request sent to Select Identity. If the response indicates a success, proceed with the next event. If not, check if a retry policy should be applied.
- **Retry Policy** — The agent should support a policy that defines what must be done if a request does not reach Select Identity or if Select Identity cannot process the request for any reason. Some of the points to consider are as follows:
 - Retry count
 - Retry interval
 - Event drop
 - Request delay

The agent's change detection capabilities can include adding, modifying, or deleting a user, or resetting a user's password. The agent must prepare and send an SPML request to the Select Identity Web Service URL to notify Select Identity of this change. 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 propagate this change back to Select Identity, an SPML `<addRequest>` must be sent that includes all of the user's attributes.

- **Changing user attributes**
User attributes are modified on the resource. An SPML `<modifyRequest>` 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>` 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>` 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 propagate 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. (See [Mapping Select Identity Attributes to the Resource Schema](#) on page 66 for more information about creating this mapping file.)

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

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

- **urn:trologica:concer:2.0#reverseSync**
Specifies 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:trologica:concerro: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:trologica:concerro:2.0#password**
The password of the administrative user.
- **urn:trologica:concerro: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.

See [SPML Examples](#) on page 125 for examples that are generated for each type of user change on the resource.

Mapping Select Identity Attributes to the Resource Schema

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/trologica/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

Following 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

Following is an example:

```
<objectClassDefinition name="SIUser" 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>
  </properties>
  ...
</objectClassDefinition>
```

- **<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 is followed 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. For 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.
- `concero:isPassword` — Set this to `true` if the attribute is a password.
- `concero:isMulti` — Set this to `true` if the resource attribute is multi-valued.
- `concero:isSensitive` — Set this to `true` if the attribute is case-sensitive.

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>

```

```

    <concerro:entitlementMap name="Replicator" />
</concerro:entitlementMappingDefinition>
<concerro:entitlementMappingDefinition>
    <concerro:entitlementMap name="Users" />
</concerro:entitlementMappingDefinition>

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

```

- **<concerro:objectStatus>**

Defines how to assign status to a user.

- **<concerro:relationshipDefinition>**

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

```

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

```

This example defines the following:

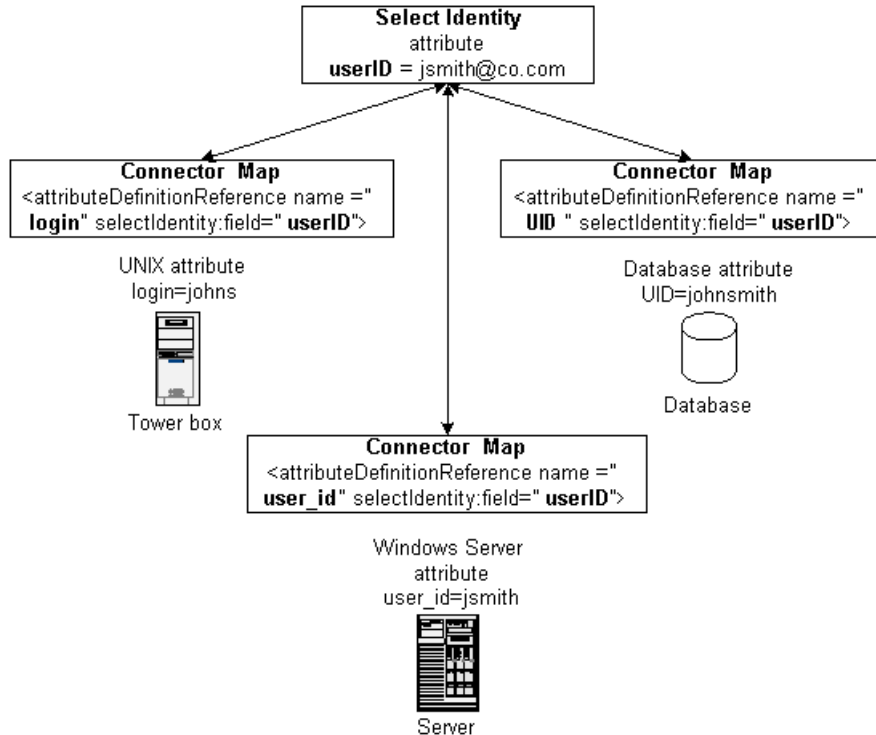
- user-to-group link can be created
- connector and resource support this operation
- user-to-group link may be deleted
- user can be unassigned from an entitlement

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.

Figure 9 Relationship Between Fields in Select Identity, Connector and Resource



Instances of `SIUserModel` and `EntitlementModel` are populated and provided by Select Identity when it calls the `SIConnector` 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>
  </properties>
</objectClassDefinition>
  
```

```

</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" />

```

Some Coding Guidelines

Following are code examples for some of the commonly used functions. This assumes that you are implementing `SICConnector` to implement the connector.



If you are directly implementing the simplified connector interface `SICConnectorInterface`, you can still make use of the logic in these snippets, but the exact methods will not be the same.

The code examples are in the following sections:

- [Key Value Return](#)
- [Key Value Computation](#)
- [User Modification](#)

Key Value Return

After a successful creation of the user in the resource, the connector is supposed to know the key value of the user in the resource. This needs to be propagated back to Select Identity for later referral.

Following is a code snippet to do the above. Basically, you need to call the `SIUserModel.setResUserId()` method.

```
public TStatus add( SIUserModel userModel ) throws TConnectorException {
    String lFuncName = "add(SIUserModel)";
    if ( msLogger.isDebugEnabled() ) {
        msLogger.debug("ENTER:\r\n"+userModel );
    }
    TStatus lStatus = new TStatus( TStatus.OP_CREATE );

    // compute key value
    keyValue = ...;

    ...
    // add the user in resource
    ...

        userModel.setResUserId( keyValue );
    catch ( TConnectorException tae ) {
        throw tae;
    }
    catch ( Exception e ) {
        if ( msLogger.isWarnEnabled() ) {
            msLogger.warn("Unable to create user:" + keyValue, e );
        }
        TConnectorException tac = new TConnectorException( "Unable to create
user:" + userModel.getResUserId() );
        tac.setLinkedException( e );
        throw tac;
    }
    finally {
        if ( msLogger.isDebugEnabled() ) {
            msLogger.debug("EXIT" );
        }
    }
}
```

Key Value Computation

At several occurrences during the writing of the connector, the user in the resource needs to be identified with a key value, and this key value needs to be used with the underlying resource API or other methods.

It is quite possible that the key value of the user in Select Identity might be different from the key value of the same user in the resource.

To properly address this user, you may want use the following code snippet:

```
private String getKeyValue(SIUserModel um, TASchemaParamBean[] schema)
    throws TConnectorException
{
```

```

String keyValue = null;

// first use the resource key value
if (um.getResUserId() != null)
{
    keyValue = um.getResUserId();
}

// Check if the key attribute value is set
if ( ( null == keyValue ) || ( keyValue.trim().length() == 0 ) ) {
    for ( int i = 0; i < schema.length; i++ ) {
        if ( schema[ i ].isKey() ) {
            keyValue = getSingleValue( um, schema[ i ].getMappingField() );
            break;
        }
    }
}

// finally use the OVSI user id
if ( ( null == keyValue ) || ( keyValue.trim().length() == 0 ) ) {
    keyValue = um.getUserId();
}

if ( ( null == keyValue ) || ( keyValue.trim().length() == 0 ) ) {
    String lError = "No primary key specified";
    if ( msLogger.isWarnEnabled() ) {
        msLogger.warn(lError);
    }
    throw new InvalidParameterException(lError);
}
keyValue = keyValue.trim();

if (msLogger.isInfoEnabled()) msLogger.info("Resource Key
Value="+keyValue);
um.setResUserId(keyValue);
return keyValue;
}

```

The above code makes sure that you compute the key in this order:

- resource user id field of `SIUserModel` if available, or
- value of the key attribute as defined in the mapping
- value of the Select Identity UserID field of `SIUserModel`

If all of the above fail to produce a valid key field value, then you should throw an exception to the caller.

User Modification

User modification might mean any of the following:

- Add a new attribute (with some value)
- Delete an existing attribute
- Add a new value to an existing multi-valued attribute

- Delete an existing value from a multi-valued attribute
- Clearing out all values of an attribute
- Replacing an attribute with a new value

Loading Existing User From Resource

To perform most of the above operations, it might be necessary to load the existing value of the user from the resource. In such cases, you may want to load only those attributes that are being modified.

In Select Identity 4.0, it is now possible to find out what attributes of the user are being modified by looking at the contents of `SIUserModel`. Following is a code snippet to do this:

```

// List of attr Ids to be loaded from resource, for update

TASchemaParamBean[] schemaBeans = getUserSchema();
String resKeyField = getKeyField( schemaBeans, "User" );
String keyValue = getKeyValue(userModel, schemaBeans);

List lRetAttrIds = new ArrayList();
for ( int index = 0; ( index < schemaBeans.length ); index++ ) {
    bean = schemaBeans[ index ];
    lTaField = schemaBeans[ index ].getMappingField();

    // pick up only those attrs that can be updated
    if ( ( !bean.isUpdate() ) || // not updateable
        (!userModel.isAttrPresent(lTaField)) || // not being modified
        bean.getResField().equalsIgnoreCase( resKeyField ) || // key field
        hasCompositeMapping( schemaBeans[ index ] )) // has composite
mapping
    {
        continue;
    }

    lRetAttrIds.add(bean.getResField());
}

String ctx = getContext( schemaBeans, userModel, "User", false );
Attributes lResAttrs = loadResourceAttrs(ctx, resKeyField, keyValue,
lRetAttrIds);

```

Computing Changes to Be Made

Once the user attributes are loaded, a thorough check is needed on each attribute to see what value is being changed and the difference to be executed on the resource.

Following is a code snippet from an LDAP connector to compute this change and prepare a modification:

```

/**
 * Build ModificationItems based on functions on th attribute value and attr
member values
 *
 * @param mods ModificatoionItem List to be updated

```

```

* @param userModel passed in by OVSI
* @param bean the attribute being modified
* @param attrValueBean value of the attribute given by OVSI
* @param resAttr Value of the attribute in resource
*/
public static void buildAndAddModifications(
    List mods,
    SIUserModel userModel,
    TASchemaParamBean bean,
    TAAAttrValueBean attrValBean,
    Attribute resAttr)

    throws Exception
{
    String lFuncName = "buildAndAddModifications()";
    if ( msLogger.isDebugEnabled() ) {
        msLogger.debug("ENTER" );
    }

    Object[] value = null;
    ModificationItem modItem = null;
    Attribute siAttr = null;
    Attribute lMergedAttr = resAttr;

    switch ( attrValBean.getOperation() ) {
        case TAAAttrValueBean.OP_REPLACE:
            {
                value = attrValBean.getValues();
                siAttr = buildLdapAttribute( bean.getResField(), value );
                mods.add( new ModificationItem( DirContext.REPLACE_ATTRIBUTE,
siAttr));
            }
            break;

        case TAAAttrValueBean.OP_ADD:
            {
                value = attrValBean.getValues();
                siAttr = buildLdapAttribute( bean.getResField(), value );
                lMergedAttr = mergeAttributes(resAttr, DirContext.ADD_ATTRIBUTE,
siAttr);
                if (lMergedAttr != null)
                {
                    mods.add( new ModificationItem( DirContext.REPLACE_ATTRIBUTE,
lMergedAttr));
                }
            }
            break;

        case TAAAttrValueBean.OP_DELETE:
            {
                if ((resAttr != null) && (resAttr.size() > 0))
                {
                    mods.add(new ModificationItem( DirContext.REMOVE_ATTRIBUTE,
                    new BasicAttribute( bean.getResField(), null ));
                }
            }
    }
}

```

```

    }
    break;

    case TAttrValueBean.OP_MODIFY:
    {
        ArrayList lAddValues = new ArrayList();
        ArrayList lDelValues = new ArrayList();
        attrValBean.categorizeValues( lAddValues, lDelValues );

        if ( lAddValues.size() > 0 ) {
            siAttr = buildLdapAttribute(
                bean.getResField(), lAddValues.toArray( new String[ 0 ] ) );
            lMergedAttr = mergeAttributes( lMergedAttr,
DirContext.ADD_ATTRIBUTE, siAttr);
            if ( lMergedAttr == null )
            {
                lMergedAttr = resAttr;
            }
        }
        if ( lDelValues.size() > 0 ) {
            siAttr = buildLdapAttribute(
                bean.getResField(), lDelValues.toArray( new String[ 0 ] ) );
            lMergedAttr = mergeAttributes( lMergedAttr,
DirContext.REMOVE_ATTRIBUTE, siAttr);
        }
        if ( lMergedAttr != null )
        {
            mods.add( new ModificationItem( DirContext.REPLACE_ATTRIBUTE,
lMergedAttr ) );
        }
    }
    break;

    default:
    }
}

/**
 * Merge the resource and si attrs based on operation:
 * return (resAttr - siAttr) or
 * return (resAttr + siAttr) or
 * return siAttr if (resAttr != siAttr)
 *
 * @param siAttr
 * @param op
 * @param resAttr
 * @return
 * @throws Exception
 */
public static Attribute mergeAttributes( Attribute resAttr, int op, Attribute
siAttr)
    throws Exception
{
    Attribute lAttr = null;
    Object lAttrVal = null;

```

```

boolean lNoChangeFlg = true;

switch (op)
{
  case DirContext.ADD_ATTRIBUTE:
  {
    if (resAttr == null)
    {
      lAttr = siAttr;
      lNoChangeFlg = false;
    }
    else
    {
      lAttr = (Attribute) resAttr.clone();
      NamingEnumeration ne = siAttr.getAll();
      while (ne.hasMoreElements())
      {
        lAttrVal = ne.nextElement();
        if (!lAttr.contains(lAttrVal))
        {
          lAttr.add(lAttrVal);
          lNoChangeFlg = false;
        }
      }
    }
  }
  break;

  case DirContext.REMOVE_ATTRIBUTE:
  {
    if (resAttr != null)
    {
      lAttr = (Attribute) resAttr.clone();
      NamingEnumeration ne = siAttr.getAll();
      while (ne.hasMoreElements())
      {
        lAttrVal = ne.nextElement();
        if (lAttr.contains(lAttrVal))
        {
          lAttr.remove(lAttrVal);
          lNoChangeFlg = false;
        }
      }
    }
  }
  break;

  default:
  case DirContext.REPLACE_ATTRIBUTE:
  {
    if (resAttr == null)
    {
      lAttr = siAttr;
      lNoChangeFlg = false;
    }
  }
}

```

```

else
{
    NamingEnumeration resAttrEnum = resAttr.getAll();
    while (resAttrEnum.hasMoreElements())
    {
        lAttrVal = resAttrEnum.nextElement();
        if (!siAttr.contains(lAttrVal))
        {
            lNoChangeFlg = false;
        }
    }

    if (lNoChangeFlg)
    {
        NamingEnumeration siAttrEnum = siAttr.getAll();
        while (siAttrEnum.hasMoreElements())
        {
            lAttrVal = siAttrEnum.nextElement();
            if (!resAttr.contains(lAttrVal))
            {
                lNoChangeFlg = false;
            }
        }
    }

    lAttr = siAttr; // in case, we have to replace
}
}

return (lNoChangeFlg) ? null : lAttr;
}

```

Matching Managed Connections

Connections (or instances of **SIConnector**) are created by the application server on a demand basis, when an operation is called on the connector. This connection is returned back to the application server connection pool upon completion of the operation. However, the physical connection to the resource need not be destroyed until the application server explicitly makes the request.

The application server calls on the **ManagedConnectionFactory** implementation to match connections in the pool before creating a new connection. The criteria for matching the connection must depend on all the connection parameters that are passed in the **TACConnectorParamValueBean** implementation.

Following is a code snippet to show the matching:

```

public ManagedConnection matchManagedConnections(
    Set arg0,
    Subject arg1,
    ConnectionRequestInfo arg2)
    throws ResourceException {

    if (!(arg2 instanceof TACConnectorParamValueBean)) {
        throw new ResourceException(

```

```

        "Invalid parameter:Expected "
        +
LDAPParamValueBean.class.getName());
    }

    // Make a local copy of the bean
    LDAPParamValueBean lBean = new
LDAPParamValueBean((TACConnectorParamValueBean) arg2);

    for (Iterator it = arg0.iterator(); it.hasNext();) {
        Object conn = it.next();
        if (conn instanceof LDAPManagedConnection) {
            LDAPManagedConnection ldapc = (LDAPManagedConnection) conn;
            LDAPParamValueBean o = ldapc.getBean();
            if (o.equals(lBean)) {
                if (msLogger.isInfoEnabled()) msLogger.info("Found matched
Connection:"+ldapc);
                return ldapc;
            }
        }
    }
    if (msLogger.isDebugEnabled()) msLogger.warn("Unable to find matched
connection");
    return null;
}

```

The equals() method of LDAPParamValueBean looks like this:

```

public boolean equals(LDAPParamValueBean other) {
    return hashCode() == other.hashCode();
}

public int hashCode()
{
    return this.toString().hashCode();
}

public String toString() {
    StringBuffer sb = new StringBuffer("LDAPParamValueBean[");

    String lKey = null;
    java.util.Iterator lIter = mValuesMap.keySet().iterator();
    while (lIter.hasNext())
    {
        lKey = (String) lIter.next();
        sb.append(lKey).append("=").append(get(lKey)).append(",");
    }
    sb.append("]");
    return sb.toString();
}

```

And so it depends on all the connection parameters kept in mValuesMap.

Schema Reloading

The attribute mapping file (or the file that has this mapping) must be reloaded only when `test()` method is called. This method is called when the Select Identity resource using this connector implementation is either newly deployed or updated. Following are the steps involved:

- Clear out the old mapping information
- Reload the file and mapping again

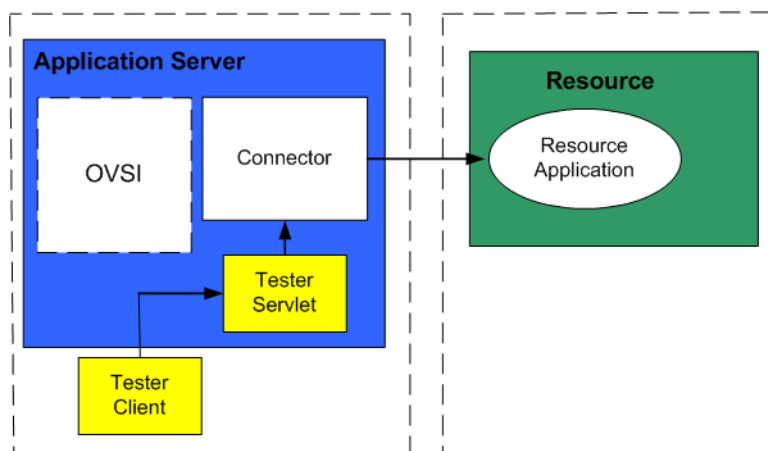
5 Connector Tester Tool

After you build the connector, you can use the connector tester tool to test provisioning operations. The tool certifies the connector before it is deployed in OVSI. It consists of the following:

- A standalone servlet WAR module, which is deployed on the application server where the OVSI server is deployed and where the target connector is installed
- SPML scripts that perform all forward-provisioning operations supported by OVSI
- A client that sends requests to the servlet (by running the SPML scripts)

The following diagram illustrates how the tester tool communicates with the connector.

Figure 10 Connector Tester Tool Communication with the Connector



Using the client, you run the scripts, which send SPML requests to the servlet. The servlet converts SPML requests to Connector API requests and invokes the connector. The connector then sends the requests to the resource application. The connector then sends results or errors back to the servlet.

The client can be used to perform bulk or iterative operations, for regression testing of the connector. You can use multiple clients for this purpose.

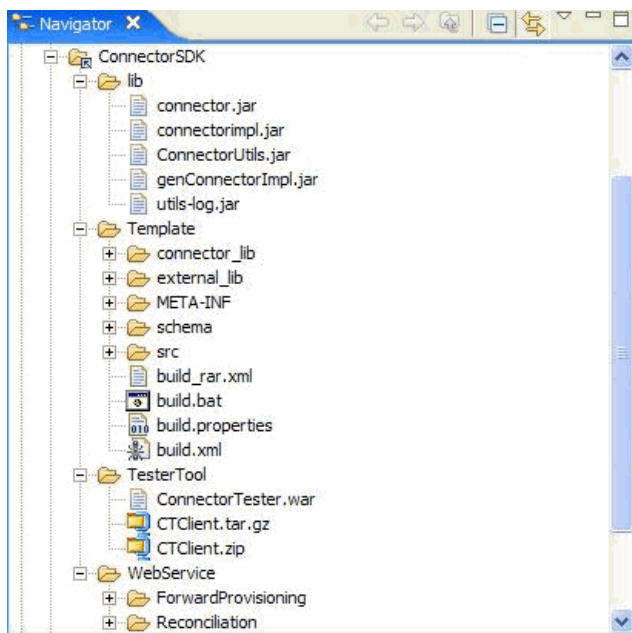
This chapter contains the following sections:

- [Installing the Tester Tool](#)
- [Testing the Connector](#)
- [Tester Tool Scripts](#)

Installing the Tester Tool

The Connector Tester Tool is part of the Connector SDK as shown in the following figure:

Figure 11 Connector SDK Structure With the Tester Tool



The following files are provided:

- `ConnectorTester.war` — The web application module that contains the servlet.
- `CTClient.tar.gz` — The client library and sample scripts if you intend to install the client on a UNIX application server

Complete the following steps to deploy and install the servlet WAR module, SPML scripts, client, and connector:

- 1 Copy the `ConnectorTester.war` file (for UNIX) from the `/ConnectorSDK/` directory on the OVSI Connector CD to the local system.
- 2 Deploy the WAR file in the application server. Here is an example of how to deploy the file on WebLogic:
 - a Log on to the WebLogic Server Console. (To load the console, load its URL in a browser. The URL is typically **`http://localhost:7001/console`**.)
 - b Navigate to ***My_domain*** → **Deployments**.
 - c Click **Web Application Modules**.
 - d In the right side of the page, click **Deploy a new Web Application Module...**
 - e Locate and select the `ConnectorTester.war` file, then click the **Target Module** button.
 - f Click the **Deploy** button.

WebLogic loads and deploys the WAR file. It should report "Success" in the Status of Last Action column on the next page.

For further verification, enter the URL of the tester servlet in an Internet Explorer browser: **`http://localhost:port/ConnectorTester`** (where *localhost* and *port* are those of the application server). If the servlet is deployed correctly, the following is displayed:

SI TestConnector servlet is up !!!

- 3 To install the client and scripts, extract the contents of the `CTClient.tar.gz` file to a directory on the server. The following directories and files are created in the target directory:

Subdirectory	Contents
<code>bin/</code>	The <code>run.ksh</code> file (for UNIX), which run the client
<code>lib/</code>	The library JAR files needed by the client to invoke an HTTP request that is sent to the servlet
<code>samplescripts/</code>	The sample scripts that perform forward-provisioning operations through the connector

- 4 Install and deploy your implementation of the connector on the application server. Refer to [Installing a Connector On WebLogic](#) on page 95 and [Configuring a Connector in Select Identity](#) on page 96 for instructions.

To verify that the connector is deployed properly, make sure that the JNDI names for the connector's connection factory and parameter factory are listed in the application server's JNDI tree view.

Testing the Connector

Complete the following steps to test the connector using the tester tool. See each step for details:

- [Step 1: Get the connection parameters of your connector.](#)
- [Step 2: Prepare the properties file with names and values of these connection parameters.](#)
- [Step 3: Test the connection to the resource.](#)
- [Step 4: Run the Tester Tool client using one of the scripts.](#)

Step 1: Get the connection parameters of your connector.

- a Be sure the `JAVA_HOME` environment variable is set.
- b Change directories to the `install_dir/bin` directory and run `run.ksh` (on UNIX) to invoke the tester client.
- c Be sure you have the following information to work with the tester tool scripts:
 - JNDI name of the connection factory
 - All parameters (and values) defined by the `TACConnectorParameterFactory` implementation and contained in the extension of `TACConnectorParamValueBean`

You can use the script `getConnectionParams.xml` to get the connection parameters of the connector you are testing. Following is an example:

Edit the `ctOpAttributes.properties` file to set the correct Connection factory JNDI Name of your connector. For example `eis/LDAPv3` and run it against the Tester Tool as follows:

```
run ../samplescripts/getConnectionParams.xml
http://localhost:7001/ConnectorTester
```

You will see a response like this:

```
<spml:extendedResponse xmlns:spml='urn:oasis:names:tc:SPML:1:0'
xmlns:dsml='urn:oasis:names:tc:DSML:2:0:core' requestID='1769'
result='urn:oasis:names:tc:SPML:1:0#success'>
  <spml:operationalAttributes>
    <dsml:attr name='urn:trologica:concero:2.0#resourceId'>
      <dsml:value>eis/LDAPv3</dsml:value>
    </dsml:attr>
    <dsml:attr name='urn:trologica:concero:2.0#elapsedTime'>
      <dsml:value>511</dsml:value>
    </dsml:attr>
    <dsml:attr name='urn:trologica:concero:2.0#connectionParams'>
      <dsml:value>accessURL</dsml:value>
      <dsml:value>suffix</dsml:value>
      <dsml:value>rootDN</dsml:value>
      <dsml:value>rootPassword</dsml:value>
      <dsml:value>userPrefix</dsml:value>
      <dsml:value>userSuffix</dsml:value>
      <dsml:value>userObjectClass</dsml:value>
      <dsml:value>groupSuffix</dsml:value>
      <dsml:value>groupObjectClass</dsml:value>
      <dsml:value>groupIdAsDn</dsml:value>
      <dsml:value>cleanUpGrpsOnDel</dsml:value>
      <dsml:value>mappingFile</dsml:value>
    </dsml:attr>
  </spml:operationalAttributes>
</spml:extendedResponse>
```

Step 2: Prepare the properties file with names and values of these connection parameters.

You can use the response from the above operation to put all the connection parameters for your connector. Then provide values for each of these parameters in this file.

Following is an example of the properties file for an LDAP resource:

```
urn\:trologica\:concero\\:2.0#resourceId=eis/LDAPv3
accessURL=ldap://127.0.0.1:62394
suffix=dc=americas,dc=hpqcorp,dc=net
rootDN=cn=Directory Manager
rootPassword=abcd1234
userPrefix=
userSuffix=ou=People
userObjectClass=top,person,organizationalPerson,inetorgperson
groupSuffix=ou=Groups
cleanUpGrpsOnDel=true
groupObjectClass=top,groupofuniquenames
mappingFile=iPlanet.xml
```

Step 3: Test the connection to the resource.

Run the `doTest.xml` script to test the connectivity between the connector and the resource (or agent).

You need to do this next after running the `getConnectionParams.xml` script and setting up your `ctOpAttributes.properties` file.

Following is a sample result of `doTest`:

Failure Case (Invalid resource password used):

```

<spml:extendedResponse xmlns:spml='urn:oasis:names:tc:SPML:1:0'
xmlns:dsml='urn:oasis:names:tc:DSML:2:0:core' requestID='1769'
result='urn:oasis:names:tc:SPML:1:0#failure'
error='urn:oasis:names:tc:SPML:1:0#customError'>
  <spml:operationalAttributes>
    <dsml:attr name='urn:trologica:concero:2.0#resourceId'>
      <dsml:value>eis/LDAPv3</dsml:value>
    </dsml:attr>
    <dsml:attr name='errorMessages'>
      <dsml:value>Code=urn:trologica:concero:2.0#generalError, Message=[LDAP: error
code 49 - Invalid Credentials]</dsml:value>
    </dsml:attr>
  </spml:operationalAttributes>
  <spml:errorMessage>Failure in Handling request</spml:errorMessage>
</spml:extendedResponse>

```

Success Case:

```

<spml:extendedResponse xmlns:spml='urn:oasis:names:tc:SPML:1:0'
xmlns:dsml='urn:oasis:names:tc:DSML:2:0:core' requestID='1769'
result='urn:oasis:names:tc:SPML:1:0#success'>
  <spml:operationalAttributes>
    <dsml:attr name='urn:trologica:concero:2.0#resourceId'>
      <dsml:value>eis/LDAPv3</dsml:value>
    </dsml:attr>
    <dsml:attr name='urn:trologica:concero:2.0#elapsedTime'>
      <dsml:value>3826</dsml:value>
    </dsml:attr>
  </spml:operationalAttributes>
</spml:extendedResponse>

```

Step 4: Run the Tester Tool client using one of the scripts.

Following is an example of sending an Add User Request to the connector:

```
run ../samplescripts/addUser.xml http://localhost:7001/
ConnectorTester
```

This sends the contents of the `addUser.xml` script to the servlet deployed in the local application server instance. See [Tester Tool Scripts](#) on page 87 for a list and description of each SPML script available for use with the tester client.

Tester Tool Scripts

The servlet is driven by SPML requests. The sample scripts provide a basis for creating (or generating) your own scripts to test the connector implementation. The following sections describe the sample scripts and the operations they perform.

Initial Connectivity-Related Scripts

- `getConnectionParams.xml`
Retrieves the connection parameters for the resource. The properties file (created in step [Step 2: Prepare the properties file with names and values of these connection parameters.](#)

on page 86) must specify the `urn\:trulogica\:concerro\:2.0#resourceId` value set to the JNDI name. All others are retrieved from the connector. This is useful in the initial stage if you do not have the connection parameter information.

- `getUserAttrDefinitions.xml`
Retrieves all user attributes as configured in the schema mapping.
- `getEntAttrDefinitions.xml`
Retrieves all entitlement attributes.
- `doTest.xml`
Performs a connectivity test between the connector and the resource, where all the connection parameters are validated. Typically, if this fails, one or more connection parameters is not given or is assigned an invalid value.

Provisioning-Related Scripts

- `getAllEntitlements.xml`
Retrieves all entitlements in the resource. Optionally, you can set the value of the identifier element with the user ID to get all entitlements of the given user.
- `getFilteredEntitlements.xml`
Performs a filtered search for entitlements.
- `isUserExists.xml`
Verifies that a user exists in the resource.
- `addUser.xml`
Adds a new user.
- `addUser-ents.xml`
Adds a user with a set of entitlements.
- `addUser-mva.xml`
Adds a user with one multi-valued attribute.
- `modifyUser.xml`
Modifies an attribute value of the user.
- `modifyUser-attr-del.xml`
Modifies a user by deleting an attribute from the user.
- `modifyUser-ents-add.xml`
Modifies a user by adding an entitlement.
- `modifyUser-ents-del.xml`
Modifies a user by deleting an entitlement.
- `modifyUser-mva-add.xml`
Modifies a user by adding a sub-value to a multi-valued attribute.
- `modifyUser-mva-del.xml`
Modifies a user by deleting a sub-value from a multi-valued attribute.
- `modifyUser-mva-replace.xml`
Modifies a user by replacing all sub-values of a multi-valued attribute.
- `resetPassword.xml`
Resets a user's password.
- `expirePassword.xml`
Expires or unexpires a user's password.

- `disable.xml`
Disables a user.
- `disableSvcMembership.xml`
Removes all entitlements from a user.
- `enable.xml`
Enables a user.
- `enableSvcMembership.xml`
Adds a list of entitlements to a user.
- `getUser.xml`
Retrieves the current attribute values for a user in the resource.
- `getUserEntitlements.xml`
Retrieves all user entitlements.
- `deleteUser.xml`
Deletes a user.

Bulk Provisioning Scripts

- `batchAdd.xml`
Adds a list of users.
- `batchModify.xml`
Modifies one attribute on a list of users.

6 Installation and Deployment

This chapter describes how to install and deploy your connector once you have built it.

This chapter contains the following sections:

- [Installing a Connector On WebLogic](#)
- [Configuring a Connector in Select Identity](#)

Installing a Connector On WebLogic

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.

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.

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.

7 Connector Migration

This chapter provides detailed information on migrating Select Identity 3.3.x connectors to Select Identity 4.0.

This chapter contains the following:

- [Reasons to Migrate](#)
- [Interface Changes](#)
- [Steps to Migrate Connectors](#)

Reasons to Migrate

The Connector API has been enhanced with Select Identity 4.0 with the additional support of the following features. You need to update your Select Identity 3.3.x-based connector to take advantage of the new features.

However, the Select Identity 3.3.x connector should be able to run without any changes with Select Identity 4.0, using the new `connector.jar` file in the CLASSPATH.

- **Multi-value attributes**

Each attribute can have multiple values in Select Identity 4.0. The value passed to the Connector by Select Identity is in an instance of `TAAttrValueBean` which holds the complete value of the attribute. Refer to the Javadoc of this class to explain the details on how to get the value of the attribute, specifically for add and modify operations.

- **Multiple entitlement types**

Depending on resource support, connectors can now support multiple types of entitlements such as groups, roles, ACLs and so on.

There is a `type` field in `EntitlementModel` which contains the entitlement type.

- **Addition/deletion/modification/emptying of attributes**

Modify user might mean not only replacing the attribute value, but also the addition of a new attribute, deletion of an existing attribute, cleaning up all the values of the attribute, or modifying the attribute. This might further mean adding/deleting sub-values. All this can be done with Select Identity 4.0 connectors.

- **Enhanced User Modify**

The user modify operation called by Select Identity now contains only the changed attributes, as compared to earlier versions where all attributes are passed and the connector does not know which ones are changed.

This allows the connector to update only the changed attributes on the resource.

- **Bulk association/dissociation of entitlements**

The Select Identity 3.3.x interface supports assignment and removal of entitlements one at a time. For example, if you are adding a user with ten entitlements in Select Identity 3.3.x, there will be one call to add the user on the connector, followed by ten calls to assign each entitlement to the user.

With the Select Identity 4.0 interface, all the entitlements are given in a single list instance, it is up to the connector to carry out the bulk assignment or call multiple times on the resource.

- **Enhanced search criteria for entitlement retrieval**

Multiple search criteria can be given to retrieve entitlements from the resource.

Interface Changes

This section lists all the changes in the Select Identity 4.0 connector API. Follow this section closely to understand what needs to be done to migrate your Select Identity 3.3.x connector.

This section contains the following subsections:

- [Connector API Changes](#)
- [Attribute Operations](#)
- [Schema Changes](#)

Connector API Changes

The tables in the following sections show the changes on the Select Identity Connector API:

- [Classes and Interfaces](#)
- [Connector Interface](#)

Classes and Interfaces

Select Identity 3.3 framework	Select Identity 4.0 framework	Comments
TACConnector	SIConnector	This is the main connector interface. That is the interface that Select Identity uses to talk to every connector. The name of this interface has changed along with cleanup of some of the unused methods.
AbstractTACConnector	AbstractSIConnector	Connector implementation can alternatively extend this abstract class, which provides some dummy implementation for the rarely used methods. Removed some un-used methods.
TACConnectorFactory	SIConnectorFactory	This is the factory for the new connector implementation.
UserModel	SIUserModel	Main Java interface that holds the user attribute data. This is what is passed from Select Identity to the connector. There are many changes in this Java interface to support more granular support for attribute level operations.
JCAUserModel	SIJCAUserModel	Implementation class for SIUserModel Java interface.

Select Identity 3.3 framework	Select Identity 4.0 framework	Comments
GroupModel	EntitlementModel	Java interface that holds the user entitlement data. This has been enhanced to hold different types of entitlements such as groups, roles, privileges, ACLs, responsibilities, or any generic entitlement type.
JCAGroupModel	JCAEntitlementModel	Implementation class for EntitlementModel Java interface.
RoleModel	Removed	Not used any longer.
EntitlementModel	Removed	Not used any longer.
ChangeLogModel	SIChangeLogModel	This is the Java class to report changes that occurred in the resource when Select Identity polls the connector. This class has been enhanced to use a cursor class. This cursor replaces the interface used earlier, which is not sufficient in most cases.
Absent	TAConnectoRequest Intf, TAConnectoResponse Intf, TAConnectoRequest, TAConnectoResponse	Introducing a generic interface to address all future API changes.

Connector Interface

This section details the connector interface. The following table shows the main changes to the SIConnector interface:

Select Identity 3.3 framework	Select Identity 4.0 framework	Comments
All provisioning methods that have UserModel argument	Changed to use SIUserModel	

Select Identity 3.3 framework	Select Identity 4.0 framework	Comments
Methods that have GroupModel argument	Changed to have EntitlementModel	All methods that get entitlements from the resource, link and unlink methods to associate and dissociate entitlements to/from users must now use the EntitlementModel interface — actually the implementation class JCAEntitlementModel instead of GroupModel (or JCAGroupModel class).
link(UserModel, GroupModel)unlink(UserModel, GroupModel)	link(SIUserModel, List)unlink(SIUserModel, List)	All single link/unlink operations must now change to link/unlink multiple entitlements that are passed in the Java List instance. This helps limit the number of times Select Identity calls the entitlements. It is up to the connector and resource to support bulk link/unlink operations.
getGroups() and getGroups(TAFilter)	getEntitlements(TAQuery)	Earlier use of TAFilter had the limitation of just one filter. TAQuery is a combination of a many TAFilter instances. This API supports multiple search criteria provided by Select Identity.
getUsers()	getUsers(TAQuery)	This method is mainly used for User Import. This now supports filtered retrieval of users.

Select Identity 3.3 framework	Select Identity 4.0 framework	Comments
<code>getUserAttributes()</code>	<code>getUserAttributes()</code>	Name of API has no change, but the implementation now must return all different entitlement types that are supported by the connector and resource. Example: GROUP, ROLE, ACL, and so on. Default is ENTITLEMENTS (if no entitlement types are returned).
String attribute value	TAAtrValueBean and TAAtrMemberValue Bean	Enhanced way of more granular control over attribute values and operations on these values. Example: add/delete/modify attributes sent as part of a modify user. Modify attribute value might further be add/delete sub-values.
void API return	TAStatus	Return status of API methods.
<code>getChangeLog(int)</code>	<code>getChangeLog(ChangeLogCursor)</code>	Cursor for iterative retrieval of records. Earlier it was just a single integer which might not be sufficient in all cases.

Select Identity 3.3 framework	Select Identity 4.0 framework	Comments
Absent	<code>loadResourceSchema()</code>	New method introduced for connectors to load schema from the resource. Can be used by attribute mapper.
Absent	<code>process(TAConnectorRequestIntf)</code>	Added a new method to support any/all future API change requirements. Generic enough to hold any data agreed upon by the caller and the connector implementation.

Attribute Operations

Starting with Select Identity 4.0 the following operations are supported with an update user operation. This information is carried in the `TAAtrValueBean` instance.

- Replace attribute value

This was the only operation supported in earlier operations, to replace the attribute value with a new value.

- Add attribute

Add a new attribute to the user in resource, with a possible value.

- Modify attribute value

For multi-valued attributes a Modify operation is supported which might mean:

- Add one or more sub-values
- Remove one or more sub-values

- Delete attribute

Remove the attribute from user in the resource.

- No change

This means that Select Identity has not changed this attribute, but is being given to the connector since it is marked as a required attribute in the mapping. Do not modify the user attribute in the resource. However, you may use the value given — for example, to address the user in the resource.

Schema Changes

The XML schema mapping file has added more properties for each mapping. The tables in the following sections explain the changes to the XML schema mapping file:

- Entity Definition
- Relationship Definition
- XML Mapping File Changes

Entity Definition

This defines how entities (users and entitlements) are mapped onto the resource. The following table shows the change in the `objectClassDefinition` of the XML mapping file.

Select Identity 3.3 framework	Select Identity 4.0 framework	Comments
<code>name="User"</code>	<code>"Name="SIUser"</code>	To identify the user model co-relating to SIUserModel.

Relationship Definition

This defines the entity relationships. The following table shows the change in the `relationshipDefinition` of the XML mapping file..

Select Identity 3.3 framework	Select Identity 4.0 framework	Comments
<code>concerro:entity="User"</code>	<code>"concerro:entity="SIUser"</code>	To identify the user model co-relating to SIUserModel.

XML Mapping File Changes

This defines each attribute mapping from the Select Identity attribute onto the resource/connector attribute. The following table shows the changes in the `attributeDefinitionReference` in the new interface.



The old mapping file will still work without any changes with the new interface. Change the mapping file only to take advantage of the new features as shown in the following table.

s

Select Identity 3.3 framework	Select Identity 4.0 framework	Comments
<code>name</code>	<code>name</code>	no change
<code>required</code>	<code>required</code>	no change
<code>concerro:tafield</code>	<code>concerro:tafield</code>	no change
<code>concerro:resfield</code>	<code>concerro:resfield</code>	no change
<code>concerro:init</code>	<code>concerro:init</code>	no change

Select Identity 3.3 framework	Select Identity 4.0 framework	Comments
absent	conzero:isPassword	Added this to identify the password attribute mapping.
absent	conzero:isMulti	To specify if the resource attribute can take a multi-valued value.
absent	conzero:isSensitive	To identify mappings of sensitive attributes. May be used to avoid logging these values.

Steps to Migrate Connectors

Following are the main steps to migrate existing connectors to the new interface. See details in each step:

- [Step 1: Change the Connector Implementation](#)
- [Step 2: Change the XML Mapping File](#)
- [Step 3: Remove Deprecated Methods](#)
- [Step 4: Use Commons Logging](#)
- [Step 5: Package the Connector](#)
- [Step 6: Use the Connector Tester Tool](#)

Step 1: Change the Connector Implementation

Change the Connector implementation so that it now implements the `SICConnector` interface and all the required methods in it.

Change all occurrences of `UserModel` to `SIUserModel` and all occurrences of `GroupModel` to `EntitlementModel`.

Implement the `SICConnector`

Change your current implementation of `TACConnector` to `SICConnector` interface. Follow the details in the Java doc of the `SICConnector` to implement all the required interface methods. It might be better to extend the abstract class `AbstractSICConnector`, which has default implementations for most of the rarely used methods so that you could focus only on the mainly required methods.

Following is a detailed explanation of the changes to be done in each of the most commonly implemented connector methods:

- **`getUserAttributes()`**

This method should return all the attributes supported by the connector/resource. In addition to what the earlier version of this method does, you need to return all the types of entitlements supported as follows:

```
    ArrayList attrList = new ArrayList();

// Add all attribute TACConnectorParamBean instances first
...
...

// add all entitlement types here
    TACConnectorParamBean entitlementBean = new TACConnectorParamBean(
"ENTITLEMENTS",

TACConnectorParamBean.ATTR_TYPE_ENTITLEMENT );
    entitlementBean.setMaxLength( 255 );
    entitlementBean.setMultiValued( true );
    entitlementBean.setRequired( true );
    attrList.add( entitlementBean );

ret = ( TACConnectorParamBean[] ) attrList.toArray( new TACConnectorParamBean[ 0
] );
return ret;
```

The above example shows the return of only the default entitlement type. This could very well return all types of entitlements supported by the connector and the resource. Following is another example returning 3 types:

```
    ArrayList attrList = new ArrayList();

// Add all attribute TACConnectorParamBean instances first
...
...

// add all entitlement types here
    TACConnectorParamBean entitlementBean = new TACConnectorParamBean( "GROUPS",

TACConnectorParamBean.ATTR_TYPE_ENTITLEMENT );
    entitlementBean.setMaxLength( 255 );
    entitlementBean.setMultiValued( true );
    entitlementBean.setRequired( true );
    attrList.add( entitlementBean );

entitlementBean = new TACConnectorParamBean( "ROLES",

TACConnectorParamBean.ATTR_TYPE_ENTITLEMENT );
    entitlementBean.setMaxLength( 255 );
    entitlementBean.setMultiValued( true );
    entitlementBean.setRequired( true );
    attrList.add( entitlementBean );

entitlementBean = new TACConnectorParamBean( "ACLs",

TACConnectorParamBean.ATTR_TYPE_ENTITLEMENT );
    entitlementBean.setMaxLength( 255 );
    entitlementBean.setMultiValued( true );
    entitlementBean.setRequired( true );
    attrList.add( entitlementBean );
```

```

        ret = ( TACConnectorParamBean[] ) attrList.toArray( new TACConnectorParamBean[ 0
    ] );
    return ret;

```

Note that the same type is returned to the connector in the `getEntitlements()`, `link()` and `unlink()` methods in the `TAFilter` instance in `TAQuery`. These methods are explained below.

- **getEntitlementAttributes()**

Earlier method: This was `getGroupAttributes()`.

This must now be changed to `getEntitlementAttributes()`

- **add(SIUserModel)**

New signature of this method is:

```
public TACStatus add(SIUserModel userModel) throws TACConnectorException
```

`userModel` has been changed to `SIUserModel`. The value of each attribute obtained is no longer a Java String as in the earlier `userModel`. It is now an instance of a bean class `TAAAttrValueBean`. Change your code to now work with this bean instance.

Following are some of the important methods in `SIUserModel`:

- `getUserId()`

This returns the Select Identity user id value. That means this is the key identifying value of the user in Select Identity

- `getResUserId()`

This returns the user id value in the resource. After an initial `add()` operation is successful, the connector returns the key value of the user in the resource by calling the method `setResUserId()`. This key value can be brought back for use in other methods, by calling `getResUserId()`. This is useful in cases where the key value of the user is different in Select Identity and the resource. Even if the key value is the same in Select Identity and the resource, you must call `setResUserId()` with the key value at the end of a successful `add()` operation.

- `get(String)`

The signature of this method shows the return value as `Object`, but the method actually now returns an instance of the `TAAAttrValueBean` class. This bean represents the value of the attribute passed to the connector by Select Identity. See the Javadoc of this class for details on how to extract the attribute value.

- `getStrValue()`

For connectors that support only single valued attributes, there is a new method in `SIUserModel` to return the string value of the attribute passed in by Select Identity. This is a convenience method and if the connector implementation knows that the value is single valued, it can directly use this method instead of `SIUserModel.get()`, which returns `TAAAttrValueBean` and extracts the single string value out of it.

- `getAttrNames()`

This returns the names of attributes present in `userModel`. In Select Identity 3.3.x, it was not possible to get this list, and the connector had to iterate through all the mappings in the XML mapping file. It is possible that Select Identity is currently interested in only a few of the attributes that are mapped in the mapping file.

Following is a brief explanation of the value contained in TAAAttrValueBean for the passed in SIUserModel add() operation:

```
abbreviation:  
    TAAAttrValueBean=av  
    TAAAttrMemberValueBean=amv
```

```
default operation=replace
```

Let's say we are adding a user with the following attributes:

- a1 - a single value attribute
- a2 - a multi-valued attribute
- a3 - another single-valued attribute

SIUserModel looks like this:

```
SIUserModel {  
    a1 - av=[---, List=[amv={---, a1sv1}]], // single-valued  
    a2 - av=[---, List=[amv={---, a2mv1}, amv={---, a2mv2}]],  
    // two values  
    a3 - av=[---, List=[amv={---, a3sv1}]], // also single-valued  
    ...  
}
```


Use the above methods to change your add() implementation. The above explained methods are useful in other connector methods as well.

You must set the key value of the user in the resource upon a successful add operation by calling the userModel.setResUserId(String) method.

Finally return the result of the add() operation in the TAStatus instance.

- **update(SIUserModel)**

Use the SIUserModel.getResUserId() to get the key value of the user so that it can be addressed in the resource.

 You must have called SIUserModel.setResUserId() in the add(SIUserModel) method.

The major change in this method is a more granular support for user modifications. A user modify may generally mean replacing an existing user's attributes with a new set of values. With the Select Identity 4.0 interface, it is now possible to go a step deeper where you could add attributes, delete attributes, replace attribute values, clear attribute values, and add/delete sub-values in a multi-valued attribute.

All this is possible now, but it depends on resource support. For example, it is possible to think of such a level of granularity with LDAP or DB provisioning. In some cases, it is simply not possible to do this and the resource or resource API only supports replace attribute value.

The value of the attribute value contained in TAAAttrValueBean has details on the attribute level operations. Following is a brief description with an example:

```
abbreviation:  
    TAAAttrValueBean=av  
    TAAAttrMemberValueBean=amv  
    --- = not to be considered
```

user modify (example 1):

- Let's say a1 changed its value to a new value
- a2 has a new sub-value
- a3 has not changed

a4 a new attribute added
a5 got deleted
SIUserModel looks like this:

```
SIUserModel {
    a1 - av=[replace, List=[{---, a1sv2}]], // a1sv2 is the new value

    a2 - av=[modify, List=[amv={add, a2mv3}]], // add sub-value a2mv3 to
the two sub-values

    a3 - av=[nochange, List=[amv={---, a3sv1}]], // only given for
required fields
// non-required fields are not given with nochange
operation

    a4 - av=[add, List=[amv={---, a4sv1}]] // new attribute added
    a5 - av=[delete, List=null] // attribute deleted in Select Identity
}
```

user modify (example 2):

Let's say a1 attribute is deleted
a2 value completely changed
a3 is nullified
a4 is emptied

SIUserModel looks like this:

```
SIUserModel {
    a1 - av=[delete, List=null],
    a2 - av=[replace, List=[amv={---, a2sv4}]], // value changed to a
single sub-value
    a3 - av=[replace, List=[amv=null]]
    a4 - av=[replace, List=[amv={replace, ""}]] // use "" as a function
to empty the value
}
```

Use the above examples to convert your update() method to do the required user level as well as attribute level operations.



- Only the changed attributes are now passed to the connector. The earlier interface used to pass all attribute values and the connector had to replace all given values.
- All attributes marked as required attributes when returning the attribute list (in getUserAttributes()) method, are passed in with all operations. The operation in TAAAttrValueBean instance for these attributes is marked with NOCHANGE, so that you know this need not be updated on the resource.

- **isUserExists(SIUserModel)**

This is a new method introduced in Select Identity 4.0 and is called by Select Identity to check if a user exists in the resource. Earlier implementation of `get(UserModel)` must be changed to this method, with a small change as shown in the following note.

▶ As noted earlier all the required attributes are passed in `SIUserModel`. If you need to compute the key you may use these attributes.

This method returns a Boolean and should not throw `ObjectNotFoundException` if the user does not exist in the resource. Instead, it should return false.

- **`get(SIUserModel)`**

This method is not used by Select Identity to check for user existence any longer. This changed to `isUserExists(SIUserModel)` as explained above.

`get(SIUserModel)` is used in User Import to get the details of a user in a resource. It must throw `ObjectNotFoundException` if the user does not exist.

- **`remove(SIUserModel)`**

Use `SIUserModel.getResUserId()` to perform the delete user operation in resource. If the user is not present, this method must throw `ObjectNotFoundException`.

▶ All attributes marked as required when returning the attribute list (in the `getUserAttributes()` method, are passed in with all operations.

- **`getEntitlements(TAQuery)`**

This method was called `getGroups()`. This method should return all entitlements in the resource. Note that there is a new argument `TAQuery`, which might include a `TAFilter`, which gives the type of entitlement that Select Identity is looking for.

If `TAQuery` and its `TAFilter` list is not empty, you must return only those types of entitlements.

▶ This method may also be used to verify entitlements that were returned earlier. In this case, the `TAFilter` instances in `TAQuery` contain values and operations to match the specific entitlement. This is a required part of the implementation of this method, to return only the entitlements asked for in `TAQuery`.

`TAQuery` is a grouping of `TAFilter` instances. Following are some of the main methods in this class:

- `getTaFilterList()`

This returns a Java List of `TAFilter` instances.

- `getMaxResults()`

This contains the maximum number of values to be returned.

- `isFilterListAnded()`

This is a boolean indicating whether the `TAFilter` instances need to be ANDed or ORed. If true, this means that all `TAFilter` instances must match, and if false this means that any of the `TAFilter` instances can match.

`TAFilter` is a filter criteria to match the results of an operation. Following are some of the main methods in this class:

- `getName()`

This returns the name of the TAFilter. In the context of entitlements this means the type of entitlement. If the connector implements only one type of entitlement, then this name can be ignored. If it implements multiple types, then this contains one of the types returned in the `getUserAttributes()` method.

— `getOperation()`

This returns the operation or criteria for the filter. The value returned is one of the following:

`TAFilter.EQUALITY`

`TAFilter.BEGINS_WITH`

`TAFilter.ENDS_WITH`

`TAFilter.CONTAINS`

`TAFilter.GTE`

`TAFilter.LTE`

`TAFilter.NOT_EQUAL`

`TAFilter.NOT_CONTAINS`

• **`link(SIUserModel, List)`**

This method was called `link(UserModel, GroupModel)`. This new method now allows bulk link operation from Select Identity. It is up to the connector to do it one-by-one or in a group.

Use `SIUserModel.getResUserId()` to get the resource key value of the user.

The List contains instances of `JCAEntitlementModel` class. Following are some of the useful methods in this class:

— `getId()`

This method was called `GroupModel.getGroupId()`, and returns the id of the entitlement. Now change all places where you called `getGroupId()` to `getId()`.

— `getType()`

This method is useful in cases where the connector/resource supports multiple entitlement types. This type returns the type of entitlement being linked to the user.



This is one of the types that you returned in the `getUserAttributes()` method.

• **`unlink(SIUserModel, List)`**

This method was called `unlink(UserModel, GroupModel)`. This new method now allows the bulk unlink operation from Select Identity. It is up to the connector to do it one-by-one or in a group.

Use `SIUserModel.getResUserId()` to get the resource key value of the user.

The List contains instances of the `JCAEntitlementModel` class. Following are some of the useful methods in this class:

— `getId()`

This method was called `GroupModel.getGroupId()`, and returns the id of the entitlement. Now, change all places where you called `getGroupId()` to `getId()`.

— `getType()`

This method is useful in cases where the connector/resource supports multiple entitlement types. This type returns the type of entitlement being linked to the user.



This is one of the types that you returned in `getUserAttributes()` method.

- **setStatus(SIUserModel, int)**

Earlier method: `setStatus(UserModel, int)`

Use `SIUserModel.getResUserId()` to get the resource key value of the user.

- **resetPassword(SIUserModel)**

Earlier method: `resetPassword(UserModel)`

Use the method `SIUserModel.getPassword()` to get the new value of the password to replace with on the resource.

- **expirePassword(SIUserModel, boolean)**

Earlier method: `expirePassword(UserModel, boolean)`

Use `SIUserModel.getResUserId()` to get the resource key value of the user.

- **getChangeLog(ChangeLogCursor)**

Earlier method: `getChangeLog(int)`

This method should check the resource for all changes that occurred after the previous call to this method and must prepare an instance of `SICChangeLogModel` with the details of these changes.

The `SICChangeLogModel` method represents the changes that occurred in the resource, in a normalized format. Any resource-specific API return values or format returned, must be parsed and converted into an instance of this class. This class contains the following main methods:

- **setCursor(ChangeLogCursor)**

The new value of the cursor must be set in `SICChangeLogModel`. A cursor identifies a checkpoint in the resource change log, so that a next call to `getChangeLog()` will read the changes past this checkpoint.

With Select Identity 3.3.x, an integer number was used as the checkpoint, which may not be sufficient. With Select Identity 4.0, you can use this cursor which has an integer along with a Java Serializable object, which can hold more information about this checkpoint.

- **addCLEntry(ChangeLogEntry)**

One instance of `SICChangeLogModel` can contain multiple instances of `ChangeLogEntry` instances which represents each change. For example, user added, user modified, user deleted are all different changes that can be reported.

`ChangeLogEntry` contains the following useful methods:

- **setUserId(String)**

This is used to set the id of the user in the resource.

- **setChangeType(int)**

This is to set the type of change that occurred in a resource. Following are the possible types:

`ChangeLogEntry.USER_ADDED`

ChangeLogEntry.USER_MODIFIED
ChangeLogEntry.USER_DELETED
ChangeLogEntry.USER_ENABLED
ChangeLogEntry.USER_DISABLED
ChangeLogEntry.USER_RESET_PASSWD

– **addAttrEntry(ChangeLogAttribute)**

This is used to add the attribute value in the change. Select Identity contains the ID and value of the attribute. The ID should represent the Select Identity attribute ID and not the physical resource attribute. If these two are different, re-mapping of the name must be done.

ConnectorFactory Implementation

This is a factory of SIconnector instances which are returned by calling the method: getConnection(TAConnectorParamValueBean).

The factory must implement SIconnectorFactory. Earlier implementation used to implement ConnectorFactory. This must be changed.

Step 2: Change the XML Mapping File

If you are using XML Mapping file to map Select Identity attributes onto resource attributes, you must change the following (see each section for details):

- [ObjectClass Definition](#)
- [Attribute Mapping Definitions](#)
- [Relationship Definition](#)

ObjectClass Definition

Change the name of the user object class to SIUser. Earlier this was "User". Following is an example:

```
<objectClassDefinition name="SIUser" description="LDAP User">
  <properties>
    <!--
      "value" can be one of: true/false/bypass
      true: the operation is supported
      false: operation is not supported and results in an exception
being thrown
      bypass: not supported, but exception is suppressed
              (currently on CREATE, UPDATE, DELETE)
    -->
    <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>
  ...
  ...

```

Attribute Mapping Definitions

You can now mark an attribute as multi-valued, password, sensitive and so on. This is optional, in the sense that the same mapping file that was used with the Select Identity 3.3.x connector can be used with the Select Identity 4.0 interface.

The following is an example taken from `iPlanet.xml`:

```

<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"
concero:isPassword="true"/>

  <attributeDefinitionReference name="Email" required="false"
concero:tafield="[Email]" concero:resfield="mail" concero:init="true"
concero:isMulti="true"/>
  <attributeDefinitionReference name="FirstName" required="false"
concero:tafield="[FirstName]" concero:resfield="givenname" concero:init="true"
concero:isMulti="true"/>
  <attributeDefinitionReference name="LastName" required="false"
concero:tafield="[LastName]" concero:resfield="sn" concero:init="true"
concero:isMulti="true"/>
  <attributeDefinitionReference name="Common Name" required="true"
concero:tafield="[FirstName] [LastName]" concero:resfield="cn" concero:init="true"
concero:isMulti="true"/>

```

```

        <attributeDefinitionReference name="employeeNumber" required="false"
concero:tafield="[Employee ID]" concero:resfield="employeeNumber" concero:init="true"/
>
        <attributeDefinitionReference name="telephoneNumber" required="false"
concero:tafield="[Business Phone]" concero:resfield="telephoneNumber"
concero:init="true" concero:isMulti="true"/>
...
...
</memberAttributes>

```

Relationship Definition

Change the user entity name in the relationship definition to use "SIUser". Following 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="SIUser" concero:cardinality="ZERO_OR_MORE"
concero:start="false" />
  <concero:party entity="Group" concero:cardinality="ZERO_OR_MORE"
concero:start="true" concero:linkfield="uniqueMember"/>
</concero:relationshipDefinition>

```

Step 3: Remove Deprecated Methods

Many of the unused methods in the Select Identity 3.3.x interface are now deprecated or removed. Compile your code with the Java compile option `deprecation="true"`, so that all usages of deprecated methods are displayed. Fix all these in your code, as they are no longer supported by Select Identity 4.0.

Step 4: Use Commons Logging

Earlier connectors used the Select Identity-provided utils logging. This is comparatively slower and it is highly recommended to use the commons logging API. Following is sample code to show the usage:

```

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

...

public class LDAPConnector extends AbstractSIConnector {

```

```

private static final String msClsName = LDAPConnector.class.getName();
private static final Log msLogger = LogFactory.getLog( msClsName );

...

public TStatus add( SIUserModel userModel ) throws TConnectorException {
    if ( msLogger.isDebugEnabled() ) {
        msLogger.debug("ENTER:\r\n"+userModel );
    }

    ...

    ...

    if (msLogger.isInfoEnabled()) msLogger.info("Resource Key Value="+keyValue);
    um.setResUserId(keyValue);
}

```

Step 5: Package the Connector

Make sure you package the connector as one single RAR and it includes all the required library Jar files that are used within the connector.

All schema files (XML mapping file or any others) are to be packaged in a single JAR file.



This is not any different from the previous version, but just a recommended style.

Step 6: Use the Connector Tester Tool

With Select Identity 4.0, a connector tester tool is packaged along with sample scripts. Use this tool to perform complete functionality testing of the connector. This saves time, and once this is done, the connector can be quickly integrated with Select Identity.

8 The Connector Template

This chapter contains the following sections:

- [Template Files](#)
- [Connector Template Code](#)

Template Files

The connector template comes with the following files. The following table briefly explains the main files included in this template:

File Name	Description
<code>ra.xml</code>	Deployment descriptor for the Resource Adapter (RA) representing the connector.
<code>weblogic-ra.xml</code>	WebLogic-specific additional deployment descriptor for the RA.
<code>DummyConnectorMapping.xml</code>	XML Schema mapping file that maps Select Identity attribute names onto resource attribute names.
<code>DummyConnector.java</code>	Main connector implementation.
<code>DummyParamResources.properties</code>	Connection parameter definitions.
<code>Build.xml</code>	Main build file for Apache ANT tool.
<code>build_rar.xml</code>	RAR build file for Apache ANT tool.
<code>build.properties</code>	Properties file for building the connector, which includes the details of the name and package name of the connector being built.

Connector Template Code

A sample called the Dummy Connector is provided in the `Connector SDK/Template` directory on the HP OpenView Select Identity product CD.

This section provides 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:

Figure 12 Hierarchy of the Dummy Connector Source

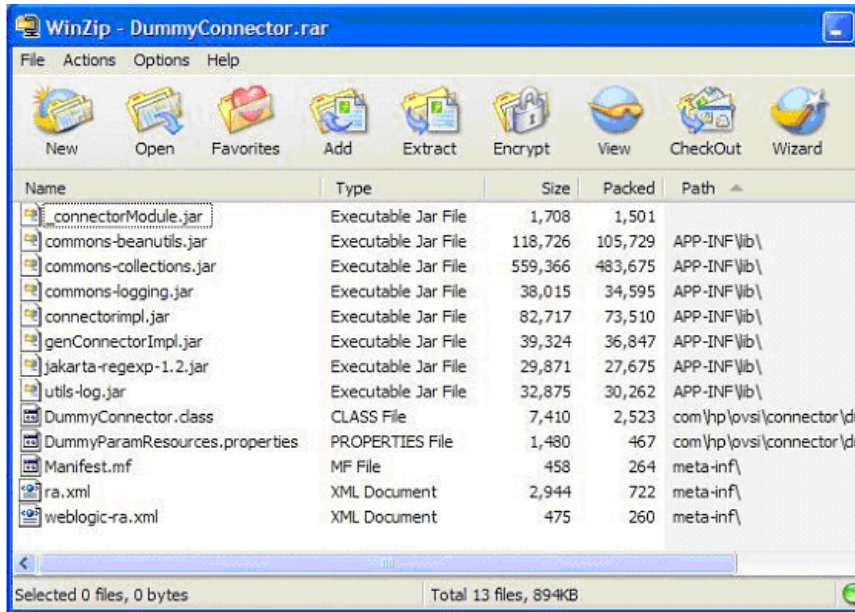


Following is an explanation of the folders:

- 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.

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

Figure 13 RAR File Contents Built From the Dummy Connector Source



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

Figure 14 Schema JAR File Contents



Index

A

- agent-based
 - forward provisioning, 61
 - reverse provisioning, 64
- Agent communication, security, and logging, 19
- agent-less
 - forward provisioning, 55
 - reverse provisioning, 62
- API
 - architecture, 12
 - interfaces and classes, 14
- architecture
 - OVSI API, 12
 - OVSI connector, 12

B

- building a connector
 - for forward provisioning, 55
 - for reverse provisioning, 62

C

- coding guidelines, 72
 - computing changes to be made example, 75
 - key value computation, 73
 - key value return, 73
 - loading existing user from resource example, 75
 - matching managed connections code example, 79
 - schema reloading, 81
 - user modification, 74
- connectors
 - API interfaces and classes, 14
 - architecture of OVSI connector, 12
 - deploying, 96
 - installing, 95
 - introduction, 11
 - mapping file, 70
 - types, 11
- connector SDK, 35
 - connector template, 39
 - connector tester tool, 38
 - simplified connector interface, 37
 - XML schema handling, 37

- connector template
 - code, 127
 - files, 127

D

- deploying a connector, 96

F

- forward provisioning
 - agent-based, 61
 - agent-less, 55
 - building a connector, 55

I

- identity objects, schema mapping, and provisioning, 20
- implementing
 - getChangeLog(ChangeLogCursor), 62
 - OVSI connector, 43
- installing
 - connector on WebLogic, 95
 - Tester Tool, 84

K

- key value computation code example, 73
- key value return code example, 73

L

- LDAP connector
 - mapping files, 66

M

- mapping
 - creating a mapping file, 70
 - OVSI attributes to resource schema, 66
- mapping file
 - LDAP connector, 66
 - overview, 70
 - simple example, 71

- migration, 99
 - attribute operations, 105
 - connector API changes, 100
 - interface changes, 100
 - reasons, 99
 - schema changes, 105
 - schema changes,entity definition, 106
 - schema changes,relationship definition, 106
 - schema changes,XML mapping changes, 106
 - steps to migrate connectors, 107

N

- new features in 4.0, 17
 - generic JCA interface implementation, 38

O

- OVSI API
 - architecture, 12
 - interfaces and classes, 14
- OVSI connector
 - architecture, 12

P

- platform support, 19

R

- reverse provisioning
 - agent-based, 64
 - agent-less, 62
 - building a connector, 62

S

- schema reloading code example, 81

T

- Tester Tool
 - files, 84
 - installing, 84
 - overview, 83
 - testing the connector, 85
- Tester Tool scripts, 87
 - bulk provisioning, 89
 - initial connectivity-related, 87
 - provisioning-related, 88
- types of connectors, 11

U

- user modification code examples, 74

W

- WebLogic
 - installing a connector, 95

X

- XML schema handling, 37