IBM Tivoli Identity Manager, Version 5.1 Custom Adapter Developer's Guide

For Tivoli Directory Integrator-based Adapters using RMI

Whitepaper version 2.3 (June 18, 2009)

Revision History

Version	**Date**	**Created by**	**Short Description**
2.3	06/18/2009	Bassam Hassoun	Added group management support.
2.2	06/18/2009	Bassam Hassoun	Reformat. Added – missing “Optional Features”.
2.1	12/01/2008	Amita Lele	Added – section Handling non UTF-8 characters in Assembly Lines - IZ37468 - PMR 51641,SGC,724

Table of Contents

1. CUSTOM ADAPTER DEVELOPMENT OVERVIEW **1**
PREREQUISITES FOR USING THIS DOCUMENT 2
2. ADAPTER ARCHITECTURE AND PROCESS FLOW **3**
ADAPTER PROCESS FLOW 4
3. CUSTOM ADAPTER COMPONENTS **6**
SUPPORTED OPERATIONS FOR A CUSTOM ADAPTER 7
4. EXAMPLE CUSTOM ADAPTER DEVELOPMENT **8**
CUSTOM ADAPTER DEVELOPMENT OVERVIEW 8
SUPPLEMENTAL REFERENCES WITHIN THIS DOCUMENT 9
STEP 1: CREATE THE MINIMUM FILES FOR A PROFILE 10
schema.dsml *10*
service.def *10*
STEP 2: CREATE A NEW SERVICE TYPE BASED ON THE PROFILE 11
STEP 3: CREATE A SERVICE INSTANCE ON TIVOLI IDENTITY MANAGER 12
STEP 4: INSTALL THE RMI DISPATCHER ON TIVOLI DIRECTORY INTEGRATOR 13
STEP 5: DEVELOP THE REQUIRED ASSEMBLYLINES 14
STEP 6: CREATE CUSTOM FORMS 18
STEP 7: CREATE A CUSTOMLABELS.PROPERTIES FILE 20
STEP 8: UPDATE SERVICE.DEF TO INCLUDE DEFAULT FORM FILES 21
STEP 9: CREATE A SINGLE PROFILE JAR FILE 22
5. DETAILS FOR ADAPTER PROFILE DEVELOPMENT **23**
DETAILS FOR SERVICE.DEF 24
service.def *24*
Operation Definition *24*
Pool ID *26*
Typical format for service.def file *26*
DETAILS FOR SCHEMA.DSML 28
schema.dsml *28*
DETAILS FOR ER<ADAPTER>ACCOUNT.XML 30
DETAILS FOR ER<ADAPTER>SERVICE.XML 30
DETAILS FOR CUSTOMLABELS.PROPERTIES 30
DETAILS FOR <ASSEMBLYLINE>.XML 30
OPTIONAL FEATURES 31
Dormant Account Attribute *31*
Supporting Data only reconciliation *31*
Service group definition (Logical Entitlements Groups) *32*
Send Only Attributes *33*
Multiple Values Attributes *34*
Adapter Category *34*
Adapter Profile Version *35*
6. DETAILS FOR ASSEMBLYLINE DEVELOPMENT **36**
ATTRIBUTE MAPPING 36
ASSEMBLYLINES FOR ADAPTER OPERATIONS 36
HOW TO GET THE DISPATCHER PARAMETERS 36
HOW TO CREATE A RETURN ENTRY AND PASS IT BACK TO DISPATCHER 37
THROWING EXCEPTIONS FROM THE ASSEMBLYLINE 39
HANDLING ASSEMBLY LINES WITH NON UTF-8 CHARACTERS 40
7. ADDING SUPPORT FOR SERVICE GROUPS MANAGEMENT **41**
INTRODUCTION 41

TIM DEFINITION 41
PROFILE CHANGES 41
Define / expand the group object class in schema.dsml *41*
Create labels for new attributes. *43*
Add / Update service group definition in service.def *43*
Add new operations definitions in service.def *45*
Create one assembly line per group operation *45*
Customize the group form *47*
Modify the search (reconciliation) assembly line to return new updated object *47*
Additional processing specifically for LDAP groups *47*
FINALIZING THE ADAPTER PROFILE 48
APPENDIX A: EXAMPLE SCHEMA.DSML 49
APPENDIX B: EXAMPLE SERVICE.DEF 51
APPENDIX C: ADD A HOOK FOR GROUPS "ADD SUCCESSFUL" 54

1. Custom Adapter Development Overview

IBM Tivoli Identity Manager manages all provisioning operations within an organization. The Tivoli Identity Manager server contains information for various policies that determine how login IDs are created, how passwords are created, which users get access to various resources, and which requests require use of approvals found in the workflow engine.

The Tivoli Identity Manager architecture consists of the following main components:

- Tivoli Identity Manager server
  The server is supported by the following data storage units:
  - Directory server (such as LDAP)
  - Database server (holds workflow process data and reports)
- Web server (Web-based user interface for administrative and end user functions)
- Adapters (previously referred to as “agents” in earlier versions of Tivoli Identity Manager)
- Managed resources

Adapters serve as the links between the Tivoli Identity Manager server and the managed resources (such as operating systems, applications, e-mail, and databases) in an organization's computing system. An adapter is an interface that functions as a trusted virtual administrator managing the user accounts on its assigned platform.

Adapters manipulate identity data for the managed resource, performing such tasks as adding, modifying, and deleting accounts. Adapters can reside on the managed resource or operate remotely. The adapter, runs as a service, independent of whether or not a user is logged on to the Tivoli Identity Manager server.

With Tivoli Identity Manager version 4.6, a new framework was introduced for building custom adapters using Tivoli Directory Integrator with Remote Method Invocation (RMI) technology. RMI technology has the following distinct advantages over DSML technology:

- RMI technology uses Java calls over a distributed system. This improves the communication and performance between Tivoli Identity Manager components.
- RMI allows more flexibility and freedom in the passing of values to Tivoli Directory Integrator. The following container objects can be used:
  - Task Call Block or TCB
  - Connector attribute
  - Work entry
- RMI implements a caching layer for initialized assemblylines that helps improve the performance of bulk and reconciliation operations.
- RMI technology improves error handling and the returning of status codes back to the Tivoli Identity Manager server.

- The custom adapter framework is based on agent-less architecture. The Tivoli Directory Integrator assemblyline becomes part of the adapter profile and is passed to the Tivoli Directory Integrator instance just before execution. Upgrading to future versions of the adapter involves only importing the latest profile.

Prerequisites for Using This Document

This document is intended for developers who want to develop a custom Tivoli Identity Manager adapter using Tivoli Directory Integrator with RMI technology.

The developer should be familiar with the following concepts and skills:

- Tivoli Identity Manager administration.
- Tivoli Directory Integrator management.
- Tivoli Directory Integrator assemblyline development.
- LDAP schema management.
- Working knowledge of Java scripting language.
- Working knowledge of LDAP object classes and attributes.
- Working knowledge of XML document structure.

Note: If the custom adapter requires a new Tivoli Directory Integrator connector, the developer must also be familiar with Tivoli Directory Integrator connector development and working knowledge of the Java programming language.

2. Adapter Architecture and Process Flow

This document discusses the custom development of a Tivoli Identity Manager adapter based on the Tivoli Directory Integrator and Remote Method Invocation (RMI). The Tivoli Identity Manager version 5.1 adapter is not a single entity, but rather, a process made up of three components:

- The adapter profile.
- Tivoli Directory Integrator assemblylines.
- Tivoli Directory Integrator connectors.

In this adapter process, the Tivoli Directory Integrator runs the appropriate assemblyline to perform a requested operation on a managed resource. An assemblyline is a data flow that receives information from an input unit, performs an operation on this input, and then conveys the finished product through an output unit.

The adapter process uses connectors to establish communication between Tivoli Directory Integrator and the managed resource. Connectors are the input and output units of a Tivoli Directory Integrator assemblyline.

Custom adapter development involves creating custom adapter profiles and modifying existing assemblylines or creating new assemblylines. This document discusses these development tasks.

Tivoli Directory Integrator includes a set of default connectors for standard resources. A custom connector for a special resource can be developed. This document, however, does not cover custom connector development. Information about custom connector development can be found in the *IBM Tivoli Directory Integrator Reference Guide*.

Dedicated Tivoli Directory Integrator servers are used to host the adapter process. On the Tivoli Directory Integrator server an RMI dispatcher must be installed. The RMI dispatcher cannot be customized, it serves as the communication link between the Tivoli Identity Manager server and the Tivoli Directory Integrator: the RMI dispatcher handles requests incoming from the Tivoli Identity Manager server and invokes the proper assemblylines.

The adapter's assemblylines are connected to corresponding service profiles and are stored as XML documents on the LDAP directory server supporting Tivoli Identity Manager. The adapter's assemblylines are originally stored on the LDAP directory server during the importing of the adapter's profile.

The LDAP directory server stores user, account, and organizational information for the Tivoli Identity Manager system. Tivoli Identity Manager has its own reserved root node in the directory server. This is the region of the server where the adapter profiles are stored.

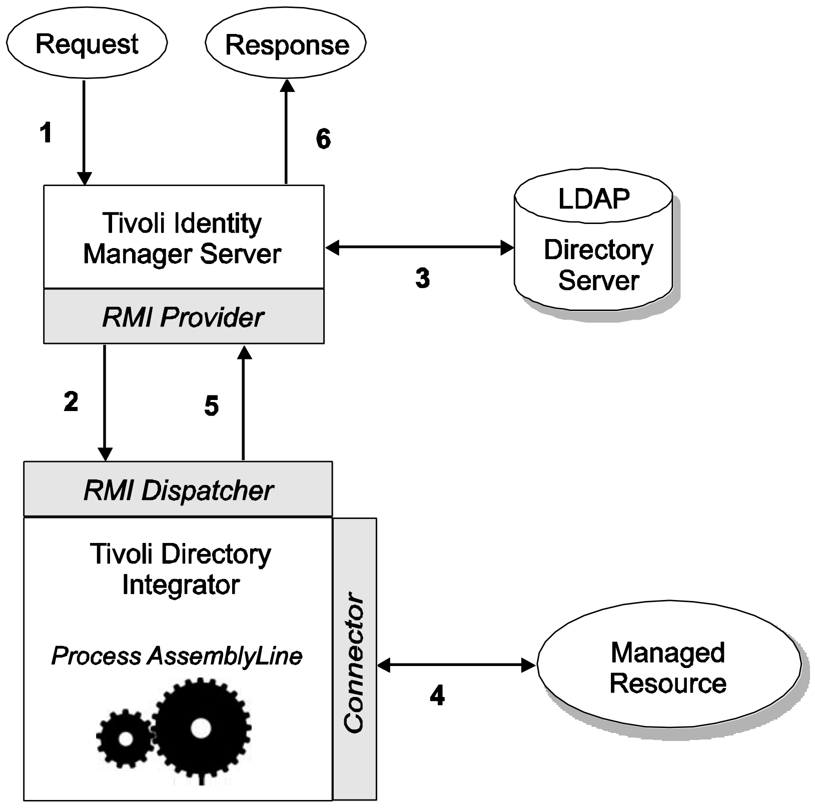
An adapter profile contains forms and parameters that allow a managed resource to be recognized as a service in Tivoli Identity Manager. The profile includes, among other things, the assemblylines used by the Tivoli Directory Integrator to perform the various operations.

In summary, the adapter process - running on Tivoli Directory Integrator - involves the following actions:

a. Receives a request to perform an operation for the managed resource.
b. Processes the requested operation by running the appropriate assemblyline and using the appropriate connectors to the managed resource.
c. Returns the result of the operation to Tivoli Identity Manager via RMI.

Adapter Process Flow

The following diagram describes the process flow for a Tivoli Directory Integrator-based adapter using RMI, when performing a requested operation for a managed resource.



1. A request for a task, or operation, to be performed for the managed resource is initiated with Tivoli Identity Manager.

   Example operations include: a manual password reset operation by a user, or an internal process that creates a new user account required by a policy-driven identity feed.

   Components of a customized adapter include assemblylines appropriate for the operations and connectors appropriate for the managed resource. Assemblylines should normally support the following operations: Add, Delete, Modify, Test, and Search.

2. The RMI dispatcher on the Tivoli Directory Integrator receives the request from the RMI provider.

   Tivoli Directory Integrator expects to use an appropriate assemblyline to perform the requested operation. If the correct assemblyline is currently cached by Tivoli Directory Integrator, then it is used. If the correct assemblyline is not present, it is downloaded from the LDAP directory server (see step 3).

   The RMI provider is included as part of the Tivoli Identity Manager server.
   The RMI dispatcher is installed as an add-on to Tivoli Directory Integrator.

3. If required, Tivoli Identity Manager downloads the appropriate assemblyline for the requested operation and passes the assemblyline via RMI to the Tivoli Directory Integrator.

   Assemblylines are one of the components included in the custom profile for the adapter. Profiles are normally stored by the LDAP directory server that supports the Tivoli Identity Manager system.

4. Tivoli Directory Integrator invokes the assemblyline and uses one or more appropriate connectors (stored on Tivoli Directory Integrator) to communicate with the managed resource.

   The requested operation is performed for the managed resource and the result is returned to Tivoli Directory Integrator.

5. Tivoli Directory Integrator returns the result of the operation via RMI to the Tivoli Identity Manager server.

6. If necessary, the Tivoli Identity Manager server provides an appropriate response to the request.

3. Custom Adapter Components

When Tivoli Directory Integrator-based adapters using RMI are used with Tivoli Identity Manager for user provisioning on a particular managed resource, the following components are involved:

- **ITDI provider**
  For custom adapters, the communication between Tivoli Identity Manager and Tivoli Directory Integrator uses RMI technology.

  The ITDI provider is a Tivoli Identity Manager service provider for RMI. The ITDI provider is included as part of the Tivoli Identity Manager server.

  When a requested operation is made from Tivoli Identity Manager (for example, User Add, User Delete, Reconciliation), the ITDI provider passes the request to the RMI dispatcher. The RMI dispatcher resides on the Tivoli Directory Integrator and passes the necessary information to perform the requested operation.
- **RMI dispatcher**
  The RMI dispatcher receives the requested operation sent by the ITDI provider and then invokes the appropriate assemblyline.

  The RMI dispatcher is installed as an add-on to Tivoli Directory Integrator.
- **Adapter profile**
  The adapter profile, normally stored on the LDAP directory server, is imported by the Tivoli Identity Manager server. The profile has the following components or files: (The variable <ADAPTER> should be replaced by the adapter name.)
  a. Schema definition (schema.dsml) (required).
  b. Service definition (service.def) (required).
     - Supported operation definitions.
  c. Assemblyline XML files for all operations
     - One or more XML file.
  d. Account form (er<*ADAPTER*>account.xml).
     - Defines the account form layout.
  e. Service form (er<*ADAPTER*>service.xml).
     - Defines the service form layout.
  f. CustomLabels.properties.
     - Labels displayed on forms for attributes.
- **Adapter**
  The adapter consists of Tivoli Directory Integrator assemblylines and connectors.
  - **AssemblyLines**
    Assemblylines are one of the components included in the adapter profile and are normally stored by the LDAP directory server that supports the Tivoli Identity

Manager system. When a request is made, the ITDI provider sends the appropriate assemblyline to Tivoli Directory Integrator, which then runs that assemblyline.

- **Connectors**
  The connectors are instantiated when an assemblyline is run. Connectors allow communication between Tivoli Directory Integrator and the managed resource. Depending on the managed resource, a Tivoli Directory Integrator connector can either be present out-of-the-box with Tivoli Directory Integrator installation or developed.

Supported Operations for a Custom Adapter

A Tivoli Identity Manager adapter has the ability to manipulate identity data on a managed resource.

A custom Tivoli Identity Manager adapter must support one or more of the following operations for managing the resource. These operations are specifically supported by RMI.

Operation	**Description**
Add	Add an account.
Delete	Delete an account.
Modify	Modify the specified attributes of an account.
Password Change	Modify the password of an account (if applicable).
Suspend	Suspend an account (if applicable).
Restore	Restore an account (if applicable).
Test	Test connection parameters by opening a connection to the managed resource.
Search	Search for accounts matching the specified search filter.
add-<group object class>	❖ Add group
modify-<group object class>	❖ Modify group
delete-<group object class>	❖ Delete group

➔ In order to support service groups management, three operation are introduced. The operation name consists of one of the key words *add, modify, delete* followed by a dash "-", then followed by the group object class name. This will give us the ability to manage more than one group type from one adapter.

add-<*group object class name*>
modify-<*group object class name*>
delete-<*group object class name*>

➔ See section (7. Adding Support for Service Groups Management) for more detail on Service Groups.

4. Example Custom Adapter Development

This section presents an example approach to developing a custom adapter. Many other approaches to developing profiles and assemblylines exist. Use the information in this section as a guideline for your own development process.

Custom Adapter Development Overview

1. Develop an adapter **profile**.

   Determine the attributes for the User Account and Group(s) on a managed resource.

   Refer to the section "Developing a Custom Adapter Profile" for details.
2. (Optional) Develop a **connector** for a specific managed resource.

   Connectors are components that connect to and access data in a data source. For example, you can use a JDBC connector to read and write to an SQL database, or an LDAP connector to access directory information. Connectors can also be set up to handle events from a data source; such as changes in a directory or database, mail arriving in a mailbox, and messages appearing in a message queue.

   IBM Tivoli Directory Integrator gives you a rich selection of connectors to choose from: such as LDAP, JDBC, Microsoft® NT4 Domain, Lotus® Notes®, and POP3/IMAP. If you cannot find the required connector, you can extend an existing connector by overriding any or all of its functions using JavaScript. You can even create your own custom connector with either JavaScript or with a traditional development language like Java or C/C++.

   The details for developing a connector is not within the scope of this document. Information about custom connector development can be found in the *IBM Tivoli Directory Integrator Reference Guide*.
3. Develop the set of Tivoli Directory Integrator **assemblylines** for the required operations. These operations can include: add, delete, modify, suspend, restore, changePassword search, and test.

   Adapter assemblylines are associated with the corresponding adapter profile and are stored as XML documents on the LDAP directory server supporting Tivoli Identity Manager.

   Refer to the section "Developing AssemblyLines for a Custom Adapter" for details.

Supplemental references within this document

- Section 5. Details for Adapter Profile Development
- Section 6. Details for AssemblyLine Development
- Appendix A: Example schema.dsml
- Appendix B: Example service.def

Step 1: Create the minimum files for a profile

Purpose of this step:

To create, at a minimum, the two files required for a profile: `schema.dsml` and `service.def`.

schema.dsml

The `schema.dsml` file should represent the accurate schema used with the finished adapter. This is the schema that is loaded into the Tivoli Identity Manager LDAP server.

You must be familiar with LDAP schema and know how to use the native LDAP administrative tools to view and modify the schema.

Note: You might be required to delete objects that were created in any unsuccessful attempts to import the profile.

See Appendix A: Example schema.dsml.

service.def

The content of this file should be as complete as possible.

The lines in the file that call out the default form files should be commented out until actual form files are created (see Step 8).

Note: You must make sure that the configuration file names (.xml files) and the assemblyline names match the values specified in the `service.def` file.

See Appendix B: Example service.def.

To complete the assemblyline development described in the next few sections, you must create a temporary profile that includes these two files.

After completing assemblyline development, you will create a final `.jar` file containing all necessary files for the profile. See Step 9.

Step 2: Create a new service type based on the profile

Purpose of this step:

To create a new service type on the Tivoli Identity Manager server based on the profile.

The profile describes all the objects that are to be managed and contains attribute and account details.

Technical notes:

Use the Tivoli Identity Manager GUI to import the test profile into Tivoli Identity Manager.

The new service type does not show up immediately and could take several minutes of refreshes before you see it.

To monitor the success or failure of this procedure, debug-level logging should be enabled on the Tivoli Identity Manager server for remote services.

Typical reasons for failure include:

- Class violations on the LDAP server when installing the schema.
- XML parser errors when loading the assemblylines.

Step 3: Create a service instance on Tivoli Identity Manager

Purpose of this step:

To create a test service instance on the Tivoli Identity Manager server.

Use this service to send requests that test the assemblylines you will develop.

Technical notes:

If you want to develop your adapter on a machine other than the Tivoli Identity Manager server, you must specify the "Tivoli Directory Integrator location" value on the service form.

Use the following syntax:

```
rmi://9.72.121.70:16231/ITDIDispatcher
```

Use the IP address of the machine where you are developing the adapter.

In addition, you will probably need to provide connector attribute values for the test environment. These will be specific to the connectors you are using in Tivoli Directory Integrator and are defined in the `schema.dsml` and `service.def` files for your custom adapter.

Step 4: Install the RMI Dispatcher on Tivoli Directory Integrator

Purpose of this step:

To install the RMI dispatcher into the Tivoli Directory Integrator instance that you will be using to develop the assemblylines.

Technical notes:

The RMI Dispatcher is not installed on Tivoli Directory Integrator by default.

An installer for the RMI Dispatcher is provided with Tivoli Identity Manager.

You can also install the required RMI Dispatcher files by installing the LDAP or UNIX/Linux adapter. These adapters install the RMI Dispatcher as part of the adapter installation.

Step 5: Develop the required assemblylines

Purpose of this step:

To develop the assemblylines for the required operations performed by this custom adapter.

Technical notes:

The method described in this section only allows you to work with one assemblyline at a time.

Because the assemblylines must be in separate files, best practice is to create a separate configuration file for each required assemblyline in Tivoli Directory Integrator.

In contrast, the UNIX and Linux adapter creates all the assemblylines in one configuration file. In order to create the profile, that configuration file must be split into individual files representing the assemblylines for each operation.

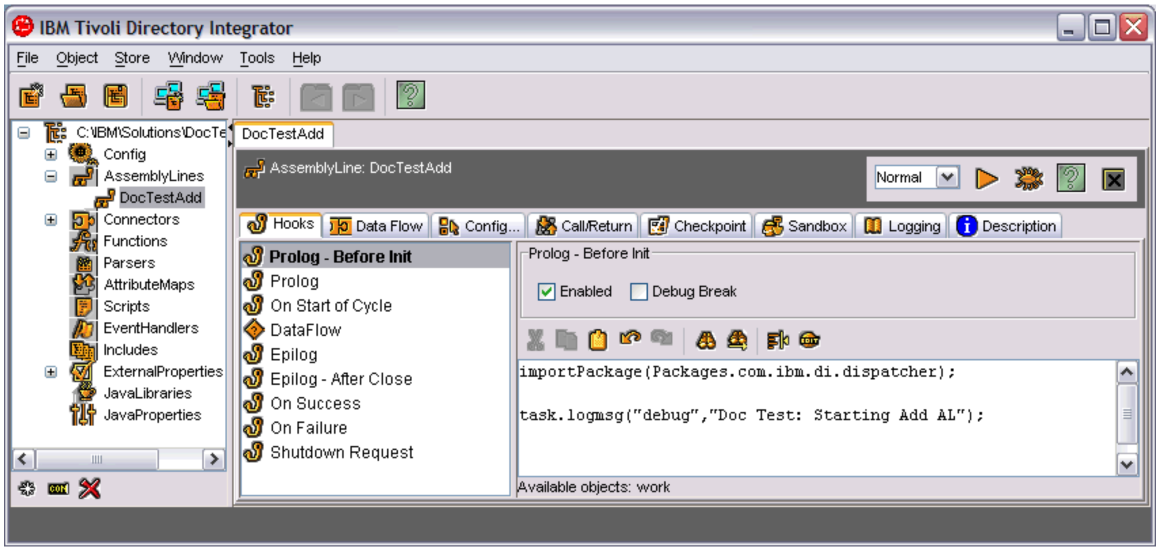
Refer to the Tivoli Directory Integrator documentation for details on using the Tivoli Directory Integrator GUI to build an assemblyline for a specific operation.

Specific requirements for RMI-based assemblylines:

Each assemblyline must include a hook for “Prolog – Before Init” that loads the RMI Dispatcher package with at least the following Javascript:

```
importPackage(Packages.com.ibm.di.dispatcher);
```

Example configuration on the Tivoli Directory Integrator GUI:



```
importPackage(Packages.com.ibm.di.dispatcher);

task.logmsg("debug","Doc Test: Starting Add AL");
```

The work entry must include the result of the assemblyline by including attributes defined in the dispatcher package.

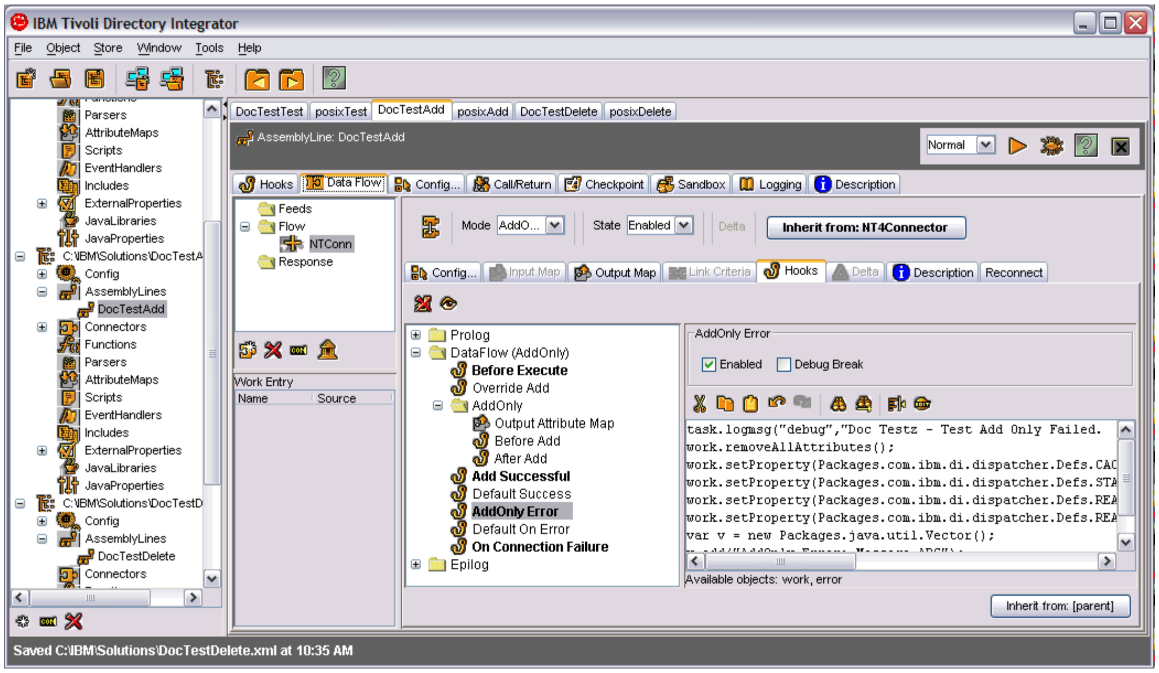
Example of success:

```
work.setProperty(Packages.com.ibm.di.dispatcher.Defs.STATUSCODE,
new Packages.java.lang.Integer
(Packages.com.ibm.itim.remoteservices.provider.Status.SUCCESSFUL));
```

Error conditions must include a reason code and reason message. Example error:

```
work.setProperty(Packages.com.ibm.di.dispatcher.Defs.STATUSCODE, new
Packages.java.lang.Integer(Packages.com.ibm.itim.remoteservices.prov
ider.Status.UNSUCCESSFUL));

work.setProperty(Packages.com.ibm.di.dispatcher.Defs.REASONCODE, new
Packages.java.lang.Integer(Packages.com.ibm.itim.remoteservices.prov
ider.Reason.OPERATION_NOT_SUPPORTED_ERROR));

work.setProperty(Packages.com.ibm.di.dispatcher.Defs.REASON_MESSAGE,
Packages.com.ibm.di.dispatcher.DispatcherReasonMessage.ADAPTER_ADD_F
AILED);

var v = new Packages.java.util.Vector();
v.add("AddOnly Error: Message ARG");

work.setProperty(Packages.com.ibm.di.dispatcher.Defs.REASON_MESSAGE_
ARGS, v);
```

Example configuration on the Tivoli Directory Integrator GUI:



```
task.logmsg("debug","Doc Testz - Test Add Only Failed.
work.removeAllAttributes();
work.setProperty(Packages.com.ibm.di.dispatcher.Defs.CAC
work.setProperty(Packages.com.ibm.di.dispatcher.Defs.STA
work.setProperty(Packages.com.ibm.di.dispatcher.Defs.REA
work.setProperty(Packages.com.ibm.di.dispatcher.Defs.REA
var v = new Packages.java.util.Vector();
```

It is important to verify that all of the possible exit point hooks return accurate status.

Development cycle:

The development cycle for assemblylines involves the following steps:

1 Modify the assemblyline in the Tivoli Directory Integrator GUI.
2 Save the assemblyline.
3 Use the **ibmdisrv** command to load the assemblyline you just saved with the RMI Dispatcher.

  Note: The RMI Dispatcher does not run in the Tivoli Directory Integrator GUI. Must run it on the server to test it.

If you store the assemblyline file in the `solutions` directory for Tivoli Directory Integrator, both the GUI and the command line server versions can use the same file.

An example command line for the server appears as follows:

Windows:

```
ibmdisrv.bat -c XTest.xml ITIM_RMI.xml -d wrkdir="C:\IBM\Solutions"
```

UNIX and Linux:

```
ibmdisrv.sh -c XTest.xml ITIM_RMI.xml -d wrkdir="/home/IBM/Solutions"
```

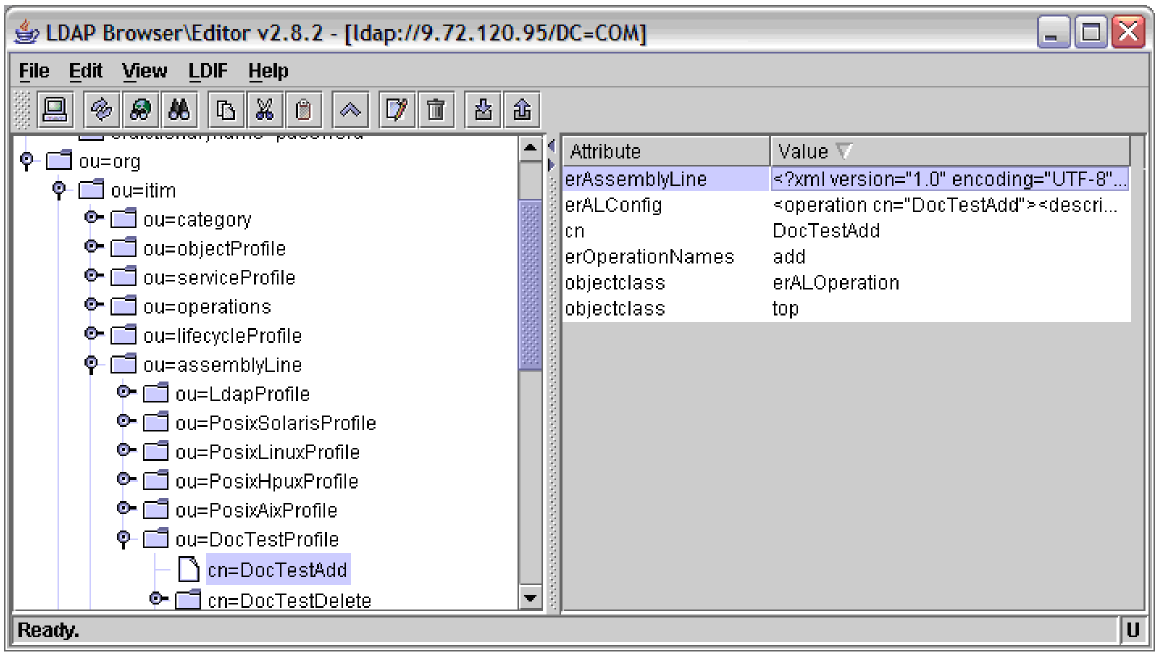
Where XTest.xml is the name of the "Test" operation assemblyline for this adapter. Substitute the appropriate operation name for each of the other assembly lines.

Make changes to the assemblyline in the Tivoli Directory Integrator GUI, save the changes, and restart the Tivoli Directory Integrator server (**CTRL-c** to stop, run the command line again). You can view the results in the log file (same log file as when running completed adapter).

Do this until all of the assemblylines are working correctly.

Once an assemblyline is working successfully, it can be loaded into the Tivoli Identity Manager LDAP directory as the attribute "erAssemblyLine" on the corresponding assemblyline object under the profile entry. This step is not required. However, by doing so, this operation will now function while you are testing other operations.

For example:



ou=org
ou=itim
ou=category
ou=objectProfile
ou=serviceProfile
ou=operations
ou=lifecycleProfile
ou=assemblyLine
ou=LdapProfile
ou=PosixSolarisProfile
ou=PosixLinuxProfile
ou=PosixHpuxProfile
ou=PosixAixProfile
ou=DocTestProfile
cn=DocTestAdd
cn=DocTestDelete

Attribute	Value
erAssemblyLine	<?xml version="1.0" encoding="UTF-8"...
erALConfig	<operation cn="DocTestAdd"><descri...
cn	DocTestAdd
erOperationNames	add
objectclass	erALOperation
objectclass	top

Note:
When you issue a request for the Add, Modify, or Delete operations from Tivoli Identity Manager, there is usually a delay in the request being received by Tivoli Directory Integrator (where the appropriate assemblyline is run). The request might reside in a pending state on Tivoli Identity Manager for several minutes. Allow sufficient time for the request to clear this pending state in Tivoli Identity Manager. Search and Test operations do not experience this delay.

Step 6: Create custom forms

Purpose of this step:

To create user-friendly entry forms for account and service data.

Technical notes:

The Tivoli Identity Manager server generates a default form from the schema. This form uses the schema names as the field labels and all fields are string entry fields. The default fields are listed in an arbitrary order.

Use the Tivoli Identity Manager Form editor to create custom forms for this service by modifying the auto-generated default forms. With the Form editor, you can create multiple tabs, change the order of attributes, provide tags for the attributes names ($tagName), etc.

To create a form with labels that can be translated, use $<attributeName> for the labels and create a CustomLabels.properties file that contains the label text (see step 7).

Once the forms are ready, you need to export them into XML files.

Saved form templates are retrievable through a Web browser by logging into Tivoli Identity Manager and issuing an HTTP GET to the FormManagerServlet (alias "formdesigner") with the following URL:

```
http://localhost/itim/formdesigner/formdesigner?request_type=1&object_profile=Person
```

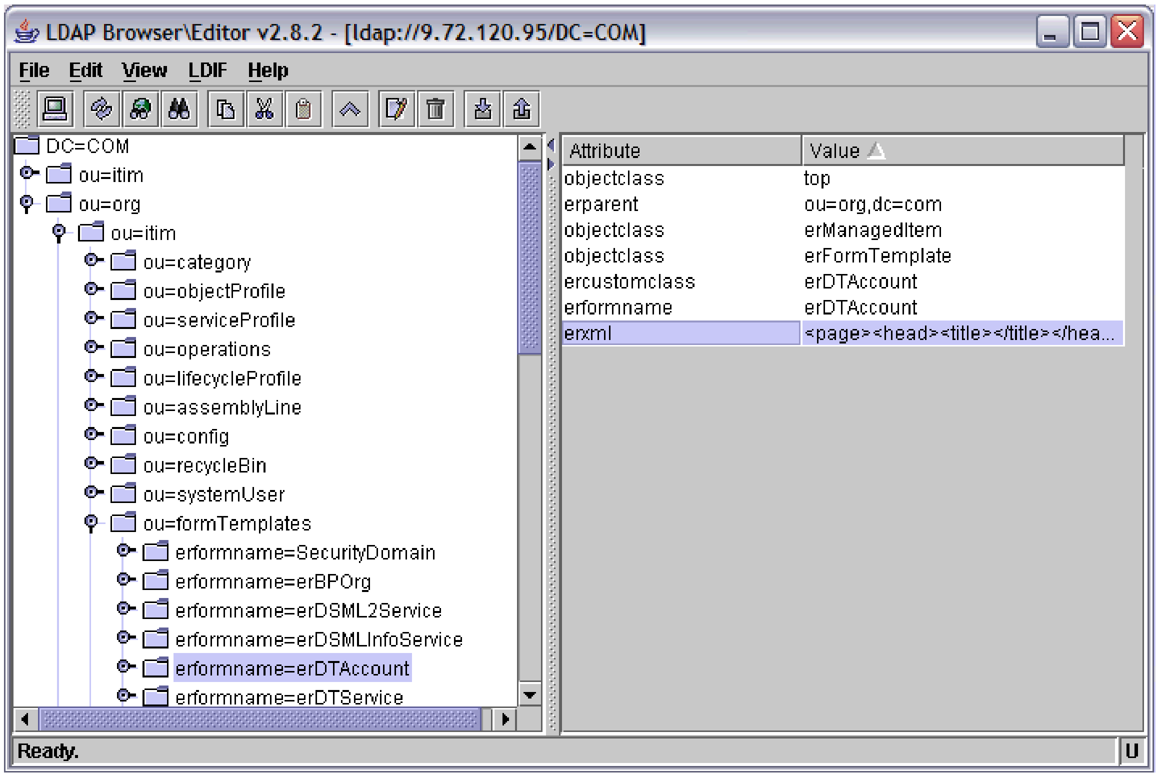
Where the first name/value pair in the query string (`request_type=1`) asks the formdesigner for a form template, and the second name/value pair (`object_profile=Person`) specifies which one (by profile name).

The response is the full XML content of the template as stored in the directory.

To export to a file, select "View Page Source ..." and "Save File As ...".

Alternatively, you can open an LDAP browser and point to the location in the Tivoli Identity Manager LDAP directory where the form template is stored and copy and paste the text into a text file.

For example, in the following LDAP Browser image, erDTAccount and erDTService are the two custom forms created for the test. The attribute “erxml” contains the actual XML text describing the form. This is the text that is copied into the XML files.



DC=COM
- ou=itim
- ou=org
  - ou=itim
    - ou=category
    - ou=objectProfile
    - ou=serviceProfile
    - ou=operations
    - ou=lifecycleProfile
    - ou=assemblyLine
    - ou=config
    - ou=recycleBin
    - ou=systemUser
    - ou=formTemplates
      - erformname=SecurityDomain
      - erformname=erBPOrg
      - erformname=erDSML2Service
      - erformname=erDSMLInfoService
      - erformname=erDTAccount
      - erformname=erDTService

Attribute	Value
objectclass	top
erparent	ou=org,dc=com
objectclass	erManagedItem
objectclass	erFormTemplate
ercustomclass	erDTAccount
erformname	erDTAccount
erxml	<page><head><title></title></hea...

Step 7: Create a customlabels.properties file

Purpose of this step:

To provide field labels for custom forms and allow labels to be translated into other languages.

If you used tag names ($) on the default forms, you need to create a `CustomLables.properties` file that includes the text for each tag.

Technical notes:

If you do not need to translate the forms to other languages, this file is not necessary. Instead, enter the labels directly using the form designer.

Step 8: Update service.def to include default form files

Purpose of this step:

To include the custom form templates in the profile so that subsequent installs of the profile will include these forms.

Technical notes:

Uncomment the lines in the `service.def` file that specify the form files (refer also to step 1).

Step 9: Create a single profile jar file

Purpose of this step:

To package all of the elements needed for this adapter into a single profile `.jar` file.

A single `.jar` file allows you to save all of the development work and allows this new adapter to be installed on other Tivoli Identity Manager instances.

Technical notes:

The new `.jar` file will include the complete set of files that make up the adapter profile.

Create a new directory using the name of the profile. Place all files in this directory.

For example:

```
schema.dsml
service.def
<accountform>.xml
<serviceform>.xml
<testAssemblyLine>.xml
<addAssemblyLine>.xml
<modifyAssemblyLine>.xml
<deleteAssemblyLine>.xml
<searchAssemblyLine>.xml
CustomLabels.properties
```

From the parent directory of this new directory, run the following **jar** command to create a `.jar` file. The name of this `.jar` file likewise must use the profile name.

```
jar -cvf TestProfile TestProfile.jar
```

Where TestProfile is the name of the profile.

While it is highly recommended that you create a complete adapter profile, the adapter that was developed using the process described in this document should be able to function properly without needing to be installed again.

5. Details for Adapter Profile Development

An adapter profile is required to deliver the following files in an installation package (`Profile JAR` file).

NOTE: Replace the variable <ADAPTER> with the actual name of the adapter

File	Description
`schema.dsml`	Required. Defines the LDAP schema used by the adapter.
`service.def`	Required. Describes the service (adapter) being configured.
`er<ADAPTER>Account.xml`	Optional. Defines the form presented by Tivoli Identity Manager while creating a new account of this profile.
`er<ADAPTER>Service.xml`	Optional. Defines the form presented by Tivoli Identity Manager while creating a new service of this profile.
`customLabels.properties`	Optional. Provides localized definitions of labels used in the form definitions.
`<assemblyline>.xml`	Optional. The Tivoli Directory Integrator configuration file that defines an assemblyline for a required adapter operation. Each assemblyline operation requires its own configuration filel. The profile installer needs each assemblyline defined in a specific format.

Details for service.def

service.def

`service.def` broadly defines the following:

Entity	Description
Service and Account	Defines the service and account objects classes used by the adapter by referencing the `schema.dsml` file.
Operation Definitions	Maps the operations listed in the "Operations Table" to assembly lines.
Properties	Properties needed by this profile, such as the location of Tivoli Directory Integrator server host.

The attribute `erServiceProviderFactory` defines the class used by the provider service. Currently it is always:

```
    com.ibm.itim.remoteservices.provider.itdiprovider.ItdiServiceProviderFactory
```

Operation Definition

There are multiple operation definition sections in the `service.def` file, one section per assemblyline used by the adapter. One assemblyline can map to one or more operations (as listed in the operations definition table). The service must define mappings for the following operations: add, modify, delete, search, and test.

All account attributes provided to Tivoli Directory Integrator service provider during operation are mapped to Initial Work Entry (refer to Tivoli Directory Integrator documentation for details on IWE). Attribute operations are set on Attribute and AttributeValue levels.

An operation may have the following additional subsections:

- parameter
- input
- dispatcherParameter
- connector
- properties

If an adapter requires an attribute to be present in the Initial Work Entry that is not sent by Tivoli Identity Manager as a part of the operation, "input" and "parameter" elements can be used to ask the Tivoli Directory Integrator Provider to add those attributes to the Initial Work Entry.

For example, in a delete operation Tivoli Identity Manager may call **delete_account dn="eruid=1000"**. This information may not be enough to delete the account if the managed target does not have the same namespace or have a naming attribute other than **eruid**. Therefore it may be required to send the value of "surname" which is the naming attribute on the managed target along with the request. For example:

```
<input name="surname" source="sn"></input>
```

This example would cause the provider to read the attribute “sn” from the account object class and place it in the Initial Work Entry as the attribute “surname”.

Similarly, if the assemblyline needs the attribute from the service attribute, then:

```
<parameter name="DummyServiceBool" source="erDummyServiceBool">
   <description>
          erDummyServiceBool attribute is needed with request
   </description>
   <default>false</default>
</parameter>
```

All the elements in the table below follow the same consistent syntax. The sources from which the attributes are taken are specified in the table along with the destination where the attributes are placed. For “input” and “parameter” the source is implicit.

For the “dispatcherParameter” the source is explicit.

There are three sources:

- **$(SO)** – Attributes from Service Object (service objectclass)
- **$(AO)** – Attributes from Account Object (account objectclass)
- **($OO)** – Attributes from Operation Object (passed during operation to Tivoli Directory Integrator service provider)

For example, “cn” from an account is specified as “$(AO!cn)”.

TAG	Source	Destination
input	Account (implicit)	IWE
parameter	Service (implicit)	IWE
dispatcherParameter	$(SO),$(AO),$(OO)	TCB.Alsettings
connector/input	Account (implicit)	TCB.connectorParameter
connector/parameter	Service (implicit)	TCB.connectorParameter
connector/dispatcherParameter	$(SO),$(AO),$(OO)	TCB.connectorParameter

Search attributes, "searchFilter" and “searchBase" are available during a search operation inside the operation object.

Attributes from Account Object have “old” values if they are modify during operation.

Element connector may have input, parameter and dispatcherParameter elements, but destination is TCB.connectorParameters, not IWE.

Element properties contain properties of the service, for example:

```
<properties>
   <property name="com.ibm.itim.itdi.properties.RMI_URL" source="erITDIurl">
          <value>rmi://localhost:16231/ITDIDispatcher</value>
   </property>
</properties>
```

Those properties are accessible via Service Object.

Pool ID

The RMI dispatcher supports assemblyline caching. After an assemblyline execution is finished, it can be returned to a cache. A specific initialized assemblyline is associated with an identifier called the "POOLID".

If an adapter wishes that an assemblyline be cached, an attribute called "POOLID" should be specified in connector of the specific connector. This "POOLID" should be uniquely able to identify a connector initialized to a specific target. POOLIDs of all the connectors of an assemblyline are used for caching and uniquely identifying the assemblyline. The developer is responsible for specifying this unique key.

Ex. For a connector named posixConnector, the following is an appropriate poolID.

```
<connector name="posixConnector">
     <dispatcherParameter name="poolID">
      <default>posixConnector/$(SO!targetURL)/$(SO!targetAdminUser)<default>
     </dispatcherParameter>
</connector>
```

If a "POOLID" is not provided for a connector, that connector is not eligible for caching.

Typical format for service.def file

```
<Service erserviceproviderfactory="" name=""
                              xmlns:xsi="" xsi:schemaLocation= "" >
   <type name="" profile="" category="" location="">
          <key>
                 <field> Key field name </field>
          </key>
          <form location="">
   </type>
   ONE OR MORE TYPES

   <operation cn="OPER NAME">
          <name> OPERATION </name>
          <description> DESCRIPTION </description>
          <input name="" source="">
          <connector name="">
                 <parameter name="" source=""> DESC </parameter>
                 ONE OR MORE CONNECTOR PARAMETERS
                 <dispatcherParameter name="">
                        <default> DEFAULT VALUE </default>
                        ONE OR MORE DISPATCHER PARAMS
          </connector>
          ONE OR MORE CONNECTORS FOR THIS OPERATION
          <dispatcherParameter name="">
                 <default> DEFAULT VALUE </default>
          ONE OR MORE DISPATCHER PARAMS FOR THIS OPERATION
```

```
            <assemblyLine>AL file name </assemblyLine>
    </operation>
    ONE OR MORE OPERATIONS FOR THIS ADAPTER
            <properties name="" source="">
                <value> VALUE </value>
            </properties>

</Service>
```

Details for schema.dsml

schema.dsml

The `schema.dsml` file describes the LDAP objects classes that are used by the adapter. Typically the following LDAP object classes need to be defined, unless the adapter uses LDAP object classes from the standard LDAP schema.

NOTE: The variable <ADAPTER> should be replaced by the Adapter name.

Object Class	**Description**
erRMI<ADAPTER>Service	Instance of a Tivoli Identity Manager service of this profile.
erRMI< ADAPTER >Account	Instance of an account of this profile.
erRMI< ADAPTER >Group	Optionally an Instance of a group of this profile. There could me more then one type of supporting data with arbitrary names.

`Schema.dsml` *file has the following format:*

```
<?xml version="1.0" encoding="UTF-8"?>
<dsml>
  <directory-schema>
    List of attributes, if any, go here.
    List of classes follow the attributes.(object classes described above)
  </directory-schema>
</dsml>
```

Attributes

The Tivoli Identity Manager server keeps a common `schema.dsml` file that contains all the common attributes that are used by adapter. Prior to defining your own attributes, try to find an attribute in the common `schema.dsml` file.

The following syntax shows how to define an attribute in the `schema.dsml` file:

```
<!-- ************************************************* -->
<!--        Single Dummy attributes                    -->
<!-- ************************************************* -->
<attribute-type single-value="true">
       <name>erDummySingle</name>
      <object-identifier> erDummySingle-OID</object-identifier>
      <syntax>1.3.6.1.4.1.1466.115.121.1.15</syntax>
</attribute-type>
```

The above lines define the erDummySingle attribute in Tivoli Identity Manager. Note the comments lines begin with "<!--" and end with "-->". Also note that <name> is not the actual variable name the adapter will use.

The <object-identifier> is the OID for this attribute.

Object Classes

The class is a collection of pre-defined attributes. Each class has a unique OID. The class section is required in the `schema.dsml` file. For example:

```
<!-- ******************************************* -->
<!--          Account classes                   -->
<!-- ******************************************* -->
<class superior="top">
      <name>erDummyAccount</name>
<description>Class representing a sample user account</description>
      <object-identifier>1.3.6.1.4.1.6054.1.3.8.2.2</object-identifier>
        <attribute ref="erUid" required="true" />
        <attribute ref="erPassword" required="false" />
        <attribute ref="erDummySingle" required="false" />
        <attribute ref="erDummyMulti" required="false" />
        <attribute ref="erDummyBool" required="false" />
        <attribute ref="erDummyInteger" required="false" />
        <attribute ref="erDummyGroupName" required="false" />
</class>
```

Note that erUid is a built-in attribute in LDAP on the Tivoli Identity Manager side. Also, erPassword is defined in the common `schema.dsml` file. This attribute is stored in LDAP in encrypted form.

The Group Object classe(s), if any, should be defined in the same way, by including appropriate attributes.

In addition to the adapter account class, Tivoli Identity Manager needs to know if there are any attributes associated with the current protocol it needs to use to communicate with the adapter. This is represented in a class as follow:

```
<!-- ******************************************************* -->
<!--               Service class                          -->
<!-- ******************************************************* -->
< class superior="top">
   <name>erDummyService</name>
      <description>Class representing a dummy service</description>
      <object-identifier>1.3.6.1.4.1.6054.1.3.8.2.1</object-identifier>
      <attribute ref="erServiceName" required="true" />
      <attribute ref="erUid" required="true" />
      <attribute ref="erPassword" required="false" />
      <attribute ref="erITDIurl" required="false" />
</class>
```

Note that all the attributes in the erDummyService class are defined in the common `schema.dsml` file. The eruid, erpassword, and erITDIurl are built-in attributes in LDAP on the Tivoli Identity Manager side. The actual protocol definition is described in the `service.def` file described later in this document.

Details for er<ADAPTER>Account.xml

This file provides the specifications for the input form that is displayed while creating a new account of this adapter profile. This file is not mandatory and default value will be created during profile installation. After that, Tivoli Identity Manager GUI customization can be used to update the Account form.

Currently Tivoli Identity Manager does not provide any means to extract new definitions from LDAP back to file, but you can use an LDAP browser/editor of your choice to extract it from LDAP. The XML definition is stored in the attribute erXML of the corresponding form item on ou=formTemplates,ou=itim,ou=<tenantname>,dc=com LDAP entry.

Details for er<ADAPTER>Service.xml

This file provides the specifications for the input form that is displayed while creating a new service for this adapter profile. This file is not mandatory and default value will be created during profile installation. After that Tivoli Identity Manager GUI customization can be used to update the Service form.

Details for CustomLabels.properties

This file provides translated text for the account and service XML forms. Each entry in this file has a format AttributeName=Description text, where AttributeName is the name of the attribute as defined in `schema.dsml`.

For example:

```
erdummyservicebool=Service BOOL
erdummyservicetargeturl=Service String
erdummysingle=Account SINGLE String
erdummymulti=Account MULITVALUE String
erdummybool=Account BOOL
erdummyinteger=Account INTEGER
erdummygroupname=Account GROUPNAME
```

This file should define the labels only for those attributes, which are defined in the adapter profile. If the profile is re-using the attributes already defined in Tivoli Identity Manager common profile, then its labels need not be defined.

Tag names must be in all lower case when entered in the `CustomLabels.properties` file.

Details for <assemblyline>.xml

This file contains the assemblyline for an adapter operation. It is created on Tivoli Directory Integrator using the Tivoli Directory Integrator toolkit.

Optional Features

Dormant Account Attribute

During profile definition, the Tivoli Identity Manager requires that you indicate the account attribute that contains the last access date to the managed resource "Dormant account attribute indicator" (i.e. that date on which a user logged into the managed resource).

If not supplied, the default mapping will be erLastAccessDate. If an adapter already uses erLastAccessDate for this information, then this step is not needed.

The profile and account definition file, service.def, will be modified as follow:
Add:

```
<AttributeMap>
        <Attribute name = "erLastAccessDate" value="attribute-name" profile = "account"/>
</AttributeMap>
```

If an adapter does not support this feature (i.e. the managed resource does not have this information), then the value should be set to none as follow:

```
<AttributeMap>
        <Attribute name = "erLastAccessDate" value="none" profile = "account"/>
</AttributeMap>
```

The location of <AttributeMap> will be under <Service>, at the same level as <operation> and <properties>.

Supporting Data only reconciliation

Supporting Data reconciliation instructs the adapter to return data for supporting data only. No account information will be returned in the reconciliation.

When Supporting Data reconciliation is requested, the ITIM server will send as recon filter value
(!(objectclass=<account_class>)).

The same filter will be passed to itdi-provider (ITIM internal component). The provider will try to find in corresponding profile operation "searchData". If the assembly line that supports this operation is found, the provider will load and execute the AL.

If the AL with "searchData" operation is not found it use the AL with "search" operation and execute it. The property search Filter will be passed as part of Working Entry (WE). It's the responsibility of the "search" AL to perform a Supporting Data only reconciliation given that search filter.

Service group definition (Logical Entitlements Groups)

Identify the attributes for "Who Has Access to What" (i.e. groups, roles ...). For every account attribute with supporting data, the adapter profile can include it as a group attribute with the following conditions:

- The attributes used must be multi-valued attributes in the account object class.
- Only one Group Definition per supporting data object class may be specified.

The profile and account definition file, service.def, will be modified as follow:

```
<ServiceGroups>
  <GroupDefinition profileName="a unique groups profile name (see convention below)"
                                     className = "the supporting data object class name"
                                     rdnAttribute= "the naming attribute if the supporting data object class"
                                     accountAttribute = "the account attribute that contains this value" />
       <AttributeMap>
         <Attribute name = "erGroupId" value="supporting data attribute used to send the value to the adapter"/>
         <Attribute name = "erGroupName" value= "supporting data attribute used for display"/>
         <Attribute name = "erGroupDescription" value= "an attribute map to the supporting data description.
                                                                  If the supporting data object class does not have a
                                                                  description field, then map it itself (erGroupDescription)" />
       </AttributeMap>
  </GroupDefinition>

 <GroupDefinition
         ...
         ...
         ...
  </GroupDefinition>

</ServiceGroups>
```

The location of <ServiceGroups>, in service.def, will be under <Service> at the same level of <operation> and <properties>.

Example of AIX groups and AIX roles:

```
<ServiceGroups>
      <GroupDefinition profileName="AixGroupProfile"
                              className = "erPosixAixGroup"
                              rdnAttribute = "erPosixGroupName"
                              accountAttribute = "erPosixSecondGroup">
                              <AttributeMap>
                                        <Attribute name = "erGroupId" value="erPosixGroupName"/>
                                        <Attribute name = "erGroupName"  value= "erPosixGroupName"/>
                                        <Attribute name = "erGroupDescription" value= "erGroupDescription"/>
                              </AttributeMap>
      </GroupDefinition>
      <GroupDefinition profileName="AixRoleProfile"
                              rdnAttribute = "erPosixAixRole"
                              className = "erPosixAixRole"
                              accountAttribute = "erPosixRoles">
                              <AttributeMap>
                                        <Attribute name = "erGroupId" value="erPosixRoleName"/>
                                        <Attribute name = "erGroupName"  value= "erPosixRoleName"/>
                                        <Attribute name = "erGroupDescription" value= "erGroupDescription"/>
                              </AttributeMap>
```

```
        </GroupDefinition>
    </ServiceGroups>
```

Group profile name convention:

The groups profile name (profileName) will be used as the naming attribute in ITIM LDAP. Therefore it must be unique within an adapter profile and across all profiles as well. As a convention, the following should be used:

<2-4 characters adapter name><supporting data type><Profile>

For example, the Windows AD Adapter AD Groups will be:

<AD><Group><Profile> -> ADGroupProfile

Other example: Oracle Adapter Roles will be:

<Ora><Role><Profile> -> OraRolesProfile

Custom Label: The profile name and the supporting data object class name must be added to the CustomLabels.properties file with a descriptive label. Examples are:

The profile name:

```
AixGroupProfile=AIX groups
AixRoleProfile=AIX roles
```

The supporting data object class name:

```
erPosixAixGroup=AIX groups
erPosixAixRole=AIX roles
```

Send Only Attributes

Indicates the Send-only attributes (these attributes will be sent to the adapter during add/modify, but will not be stored in TIM LDAP).

The sendOnly property can be added under the <operation> tag in service.def file as follow:

```
<operation cn="posixAdd">
    ….
    ….
        <sendOnly name=”attribute-name1” />
        <sendOnly name=”attribute-name2” />
</operation>
```

Multiple Values Attributes

The TIM server sends multi-values attributes with "add" and "delete" operations (add new selected values and deleted old un-selected values). Some manage resources require the "replace" option where TIM must send the new set of values. In order to flag such attributes, a property "replaceMultiValue" can be added under the <operation> <parameter> tag in service.def file as follow:

```
…
<operation cn="posixModify">
        ….
        ….
        < replaceMultiValue name="attribute-name1" />
        < replaceMultiValue name="attribute-name2" />
</operation>
```

Adapter Category

Indicates the adapter category: System, Database, or Application. Although this was dropped from TIM 5.0, the profile import (service type import) will read the new AdapterCategory and stores it under the "Service Profile" definition as a property. The TIM code does not read this property as of TIM 5.0. It could be used in the future.

Add the "AdapterCategory" property under <properties> in service.def file as follow:

```
<properties>
        <property name="com.ibm.itim.itdi.properties.RMI_URL"
    source="erITDIURL">
            <value>rmi://localhost:16231</value>
        </property>
        <property name = "AdapterCategory">
            <value>System</value>
        </property>
    <property name = "ProfileVersion"> <value>5.0.1003</value> </property>
</properties>
```

Adapter Profile Version

With TIM 5.0, the adapter profile service definition can contain a property to indicate the profile version number. This value will only be visible from an LDAP browser and will be used to verify that a client has imported the correct profile.

Add the "ProfileVersion" property under <properties> in service.def file as follow:

```
<properties>
        <property name="com.ibm.itim.itdi.properties.RMI_URL" source="erITDIURL">
                <value>rmi://localhost:16231</value>
        </property>
        <property name = "AdapterCategory">
                <value>System</value>
        </property>
        <property name = "ProfileVersion">
                <value>5.0.1003</value>
        </property>
</properties>
```

6. Details for AssemblyLine Development

The RMI dispatcher calls an assemblyline in response to a call from the RMI provider. The dispatcher then collects the results from the assemblyline and responds to the RMI provider. An assemblyline defines an adapter in the view of the RMI dispatcher. For an assemblyline to be compliant with the dispatcher it must adhere to the following rules:

Attribute mapping

All attributes defined in the Account Object class in the adapter profile must be mapped to corresponding attributes on the managed resource.

This can be done by using the Schema Discovery on the managed resource, using the Tivoli Directory Integrator connector for the specific managed resource, and then mapping profile attributes to them.

Assemblylines for adapter operations

A one assemblyline per adapter operation should be created. Operations include: ADD, MODIFY (Include Attribute modification, Suspend, Restore and Password change), DELETE, SEARCH, and TEST.

How to get the dispatcher parameters

When a request is made from Tivoli Identity Manager, the dispatcher initializes the connector (as defined in `service.def`) and then invokes the appropriate assemblyline and passes the TCB (Task Control block) to the connector. An important set of information in TCB is the Dispatcher Parameters, which are adapter-specific attributes on the Profile Service form. The assemblyline needs to read the values of these parameters before it can perform an operation.

The following script shows, how an assemblyline can read the dispatcher parameters:

```
//Get AL config object.
var gALCfg = task.getConfigClone();

//Get AL settings object from AL config.
var gALSettings = gALCfg.getSettings();

// Get the required parameter value by specifying the parameter name.
var paramValue =  gALSettings.getStringParameter(<Param Name>);
```

How to create a RETURN entry and pass it back to dispatcher

A com.ibm.di.entry.Entry object (or a list of such objects) is the sole vehicle of communication between an assemblyline and the dispatcher. The data is carried in the Entry object in the form of attributes.

The meta-data is carried in the form of properties in the Entry object.

After an operation assemblyline is executed by the dispatcher, the dispatcher expects the assemblyline to set the following properties in the Entry objects and return that Entry object back to the dispatcher. The properties are set using the Entry.setProperty() function.

The Valid Values column specifies the class name where the values are defined. Actual valid values are specified in a separate table below.

Property	Description	Valid Values	Type
STATUSCODE	End status of the current request.	com.ibm.itim.remoteservices .provider.Status	String representation of int.
REASONCODE	The reason of an error if one occurs.	com.ibm.itim.remoteservices .provider.Reason	String representation of int.
REASON_MESSAGE	The message applicable to the current situation. The messages are translated.	com.ibm.di.dispatcher.DispatcherReasonMessage	String. All the values defined in this class always maps to corresponding values in the file "tmsMessages.properties" on Tivoli Identity Manager installation (\\itim\data folder). Check for all the values in this file starting with "com.ibm.di". Any value other than those present in this file, are not allowed.
REASON_MESSAGE_ARGS	The reason message may have substitution arguments such as assemblyline name etc. These arguments are passed in this property.	Context dependant. Consult the specific message in the Javadoc. It describes the arguments the message accepts.	Java.lang.Vector
SEARCH_FINISHED	"true" if this the last entry of the search session. Assemblyline can use this property to signal end of	"true" or "false"	String representation of Boolean.

	search to the Dispatcher.		
CACHE_CONNECTORS	This property is used to signal the Dispatcher that the connectors associated with the assemblyline may be cached.	"true" or "false"	String representation of Boolean.
DO_NOT_FILTER	The Dispatcher provides ldap search filtering service. If the assemblyline itself supports filtering it may not need the Dispatcher filtering service. Using this property the assemblyline can ask the Dispatcher to stop filtering. The dispatcher remembers the last value of this property if the property is not set.	"true" or "false" The default value is "false", which means the Dispatcher starts the process with filtering enabled.	String representation of Boolean.

Actual values to be used while generating return Entry:

Property	Valid Values
STATUSCODE	com.ibm.itim.remoteservices.provider.Status.SUCCESSFUL com.ibm.itim.remoteservices.provider.Status.UNSUCCESSFUL com.ibm.itim.remoteservices.provider.Status. SUCCESSFUL_WARNING

Property	Valid Values
REASONCODE	All values are defined in the class com.ibm.itim.remoteservices.provider.Reason PROCESSING_ERROR COMMUNICATION_ERROR CONFIGURATION_ERROR AUTHENTICATION_ERROR NAME_INVALID_ERROR OPERATION_NOT_SUPPORTED_ERROR NO_SUCH_ATTRIBUTE_ERROR NAME_NOT_FOUND_ERROR

	INVALID_SEARCH_FILTER_ERROR SIZE_LIMIT_EXCEEDED_ERROR TIME_LIMIT_EXCEEDED_ERROR NAME_ALREADY_BOUND_ERROR INVALID_SEARCH_CONTROLS_ERROR

In addition, the entry should also contain the attributes and values that could not be set during ADD/MOD type of operations.

For example: Assume that a MODIFY request is made from Tivoli Identity Manager with the following three attributes:

```
Attr1   Val11
Attr2   Val21, Val22, Val23 (multi value attribute)
Attr3   Val31
```

While processing this MODIFY request, if

```
Setting Attr1 failed on managed resource AND
Setting Val22 of Attr2 failed on managed resource
```

then, the return ENTRY should contain:

```
Attr1   Val11
Attr2   Val22
```

In this case, Attr1 and Val22 of Attr2 will be treated as FAILED and other attributes will be considered as SUCCEEED on Tivoli Identity Manager server side.

Throwing Exceptions from the assemblyline

The dispatcher defines ITDIAdapterException (com.ibm.di.exception.ITDIAdapterException) as a communication vehicle for exceptions between the assembly lines and the dispatcher.

Any checked exception (declared) thrown by the assemblyline must be of class ITDIAdapterException or a subclass of it. The assemblyline is required to catch all exceptions that occur downstream and wrap them in a ITDIAdapterException subclass before returning them to the dispatcher

This exception carries a payload of an Entry object. This object is used to carry meta-data (in Properties) as described in the previous section. The component detecting the exceptional condition should set as many properties in this Entry object as possible.

Handling Assembly Lines with non UTF-8 characters

The AL encoding is declared in the first line of the AL xml file. The default encoding for ALs is UTF-8. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
```

If the AL contains no UTF-8 characters then follow either of these two methods so that TDI is able to load the AL using the correct encoding at run time.

Method 1

- Change the encoding in AL from UTF-8 to a character set which supports the non UTF-8 characters.
- For e.g. if the AL contains German characters such as ä, ö etc, then ISO-8859-1 encoding can be used.
- Replace <?xml version="1.0" encoding="UTF-8"?> with <?xml version="1.0" encoding="ISO-8859-1"?>.

Method 2

- If the non UTF-8 characters have equivalent UTF-8 representations then use those in the AL.
- For e.g. &auml;&ouml;&uuml;&Ouml;&Auml;&Uuml; is the UTF-8 representation of ä;ö;ü;Ö;Ä;Ü.
- Replace non UTF-8 chars with equivalent UTF-8 representation

7. Adding Support for Service Groups Management

Introduction

The ability to manage service groups is a new feature introduced in TIM 5.1. By service groups, TIM is referring to any logical entity that can group accounts together on the managed resource. Examples of service groups are: UNIX groups, AIX roles, LDAP groupOfNames, and AD groups. Managing service groups implies the following:

- Create service groups on the managed resource.
- Modify attribute of a service group.
- Delete a service group.

➔ Note that renaming a service group is not supported in the TIM 5.1 release.

TIM Definition

Service groups are managed within a service type instance. When groups are created, modified or deleted, the operation is applied to one service instance only. Service group profiles are defined within the same service profile as account profiles. The adapter profile has been updated to define a group profile and indicate whether this group profile is manageable by TIM. All service group operation requests are serviced by the same adapter that manages account requests.
Service groups are stored under the service type instance in the TIM LDAP similar to supporting data. In fact, they are supporting data that can be managed.

Profile Changes

To add service groups management support to any TDI adapter, the following steps must be done:

1. Define / expand the group object class in schema.dsml
2. Create labels for new attributes.
3. Add / Update service group definition in service.def (with "managed" set to true) .
4. Add new operations definitions in service.def
5. Create one assembly line per group operation.
6. Customize the group form.
7. Modify the search (reconciliation) assembly line to return new updated object.

Define / expand the group object class in schema.dsml

Using the LDAP groups as an example, we start with the definition of LDAP group object class groupOfNames:

```
MUST ( cn $ member )
MAY ( businessCategory $ seeAlso $ owner $ ou $ o $ description ) )
```

In order to manage this object in TIM we need to create an equivalent object class used by the LDAP adapter. In this case, we already have a LDAP adapter group object class, so we will update the object class:

Current LDAP adapter groups object class:

```
<class superior="top">
        <name>erLdapGroupAccount</name>
        <description>Class representing LDAP Group</description>
        <object-identifier>1.3.6.1.4.1.6054.3.139.1.2</object-identifier>
        <attribute ref = "erLdapGroupName" required = "true" />
        <attribute ref = "erLdapGroupRDN" required = "false" />
</class>
```

Updated LDAP adapter groups object class to manage groups:

```
<class superior="top">
        <name>erLdapGroupAccount</name>
        <description>Class representing LDAP Group</description>
        <object-identifier>1.3.6.1.4.1.6054.3.139.1.2</object-identifier>
        <attribute ref = "erLdapServiceGroup" required = "true" />
        <attribute ref = "erLdapGroupRDN" required = "false" />
        <attribute ref = "erLdapContainerName" required = "false" />
        <attribute ref = "erLdapGroupFullName" required = "false" />
        <attribute ref = "erLdapGroupOwner" required = "false" />
        <attribute ref = "erLdapGroupDescription" required = "false" />
        <attribute ref = "erLdapGroupBusinessCategory" required = "false" />
        <attribute ref = "erLdapGroupOrganization" required = "false" />
        <attribute ref = "erLdapGroupOrganizationalUnit" required = "false" />
        <attribute ref = "erLdapGroupSeeAlso" required = "false" />
</class>
```

Where:

erLdapServiceGroup

The CN value used to create the group in LDAP. This will be a unique value on TIM.

erLdapGroupRDN

Contains the full DN value of a group. Will be used to assign the group to an LDAP account from TIM.

erLdapContainerName

The full DN of the container where the group will be created in LDAP.

erLdapGroupFullName

Additional CN values for the group.

erLdapGroupOwner

The "owner" attribute in groupOfNames object class.

erLdapGroupDescription

The "description" attribute in groupOfNames object class.

erLdapGroupBusinessCategory

The "businessCategory" attribute in groupOfNames object class.

erLdapGroupOrganization

The "o" attribute in groupOfNames object class.

erLdapGroupOrganizationalUnit

The "ou" attribute in groupOfNames object class.

erLdapGroupSeeAlso

The "seeAlso" attribute in groupOfNames object class.

Notes on the updated LDAP adapter group object class:

- The object groupOfNames attribute "member" has no equivalent attribute in the LDAP adapter group object class. This is by design since group membership is stored within the account object on TIM and not the group object.
- The attribute erLdapGroupName is changed to erLdapServiceGroup. This change was not necessary. It was made for distinction only, since the LDAP account object class uses the erLdapGroupName attribute. Furthermore, it is recommended not to change attribute names when updating object classes as it will create a challenge when updating existing data.
- erLdapServiceGroup and erLdapContainerName will construct the full DN of a group. It is by design that there are two attributes for ease of use on the TIM groups form. A user will enter a group name, and then select the container where the group should be created. The LDAP adapter "groups add" assembly line will concatenate the values to construct the full DN of the group
- erLdapGroupRDN is the full DN of the group. This value is returned by the adapter "groups add" assembly line with the success status.

Create labels for new attributes.

Add a key (attribute names) for every attribute that will be displayed on the group form to the CustomLabels.properties file:

```
erldapservicegroup=Group Name
erldapgroupdescription=Group Description
erldapgroupfullname=Group Full Name
erldapgroupowner=Group Owner
erldapgroupbusinesscategory=Group Business Category
erldapgrouporganization=Group Organization
erldapgrouporganizationalunit=Group Organizational Unit
erldapgroupseealso=See Also
erldapinitialgrpmem=Initial group member
```

Additional label are needed for the group profile name definition (see section 3.3):

```
LdapGroupProfile=LDAP groups
```

Add / Update service group definition in service.def

Within the service.def file, you can define a service group for an adapter.

```
<ServiceGroups>
        <GroupDefinition profileName="LdapGroupProfile"
                className="erLdapGroupAccount"
                rdnAttribute="erLdapServiceGroup"
                accountAttribute="erLdapGroupName">
                <AttributeMap>
                        <Attribute name="erGroupId" value="erLdapGroupRDN"/>
                        <Attribute name="erGroupName" value="erLdapServiceGroup"/>
                        <Attribute name="erGroupDescription"
                    value="erLdapGroupDescription"/>
                </AttributeMap>
                <properties>
                        <property name = "Managed">
                            <value>true</value>
                    </property>
                </properties>
                <form location="erLDAPGroupAccount.xml" />
        </GroupDefinition>
</ServiceGroups>
```

Where

profileName
Similar to a service or account profile, this is the unique value that defines a group profile in TIM's LDAP.

className
The name of the group object class.

rdnAttribute
Set to the attribute used as the naming attribute to create the group object under the service.

accountAttribute
Set to the account object attribute that contains the group membership.

erGroupId
Mapped to the attribute within the group object class that is used to assign a group to an account.

erGroupName
Mapped to the attribute within the group object class that is used as the unique value in TIM's LDAP. This is what the user enters as a group name on TIM when creating the group.

erGroupDescription
Optional. It is mapped to the attribute within the group object class that contains a group description.

Managed *property*
If set to "true", the TIM server will allow the creation, modification and deletion of this group type.

form location

The name of the xml file within the adapter profile jar file that contains the groups form. If not specified, TIM will list all attribute of the group object class on the form.

Add new operations definitions in service.def

The TDI/RMI adapters are designed in such a way that each operation within TIM can be linked to an assembly line in TDI. You can even link more than one operation to one assembly line. The itdi-provider within TIM uses service.def to determine which TIM operation correspond to what TDI assembly line.

The syntax to link a TIM operation with a TDI assembly line is as follow:

<operation cn="***assembly line name***">
        <name>***TIM operation***</name>
        ….
        ….
        ….
        <assemblyLine>***file name that contains the assembly line***</assemblyLine>
</operation>

In order to support service groups management, three operation are introduced. The operation name consists of one of the key words *add, modify, delete* followed by a dash "-", then followed by the group object class name. This will give us the ability to manage more than one group type from one adapter.

add-<*group object class name*>
modify-<*group object class name*>
delete-<*group object class name*>

For our example of LDAP groups, the names of the operations will be:

add- erLdapGroupAccount
modify- erLdapGroupAccount
delete- erLdapGroupAccount

Create one assembly line per group operation

As with accounts operation, groups operations need to execute an assembly line in TDI. For ease of maintenance, it's recommended that you create three assembly lines: add group, modify group and delete group assembly lines.

➔ The add group assembly line must return all the group attributes values when successfully adding a group. This can be done as an "Add Successful" hook in the add group assembly line. See Appendix "C" for detailed description.

Refer to section "Details for AssemblyLine Development" of this Custom Adapter Developer's Guide for detailed instructions on how to create an assembly line for a TIM operation. Below is a portion of the LDAP adapter groups add, modify and delete operations definitions in service.def.

For detailed reference, the full LDAP adapter service.def file can be found in the adapter profile "LdapProfile.jar", located under TIMHOME/config/adapters directory. Three assembly lines have been created: LDAPGroupAdd, LDAPGroupDelete and LDAPGroupModify:

```
<operation cn="LDAPGroupAdd">
        <name>add-erLdapGroupAccount</name>
        <description>
            group add operations are supported by AddAssemblyLine
        </description>
        <connector name="conLDAPGroup">
            …
        </connector>
        <connector name="conLDAPGroupLookup">
            …
        </connector>
        <dispatcherParameter
            …
        </dispatcherParameter>

        <assemblyLine>LDAPGroupAdd.xml</assemblyLine>
</operation>

<operation cn="LDAPGroupDelete">
        <name>delete-erLdapGroupAccount</name>
        <description>
            group delete operation supported by DeleteAssemblyLine
        </description>
        <connector name="conLDAPGroup">
            …
        </connector>
        <connector name="conLDAPGroupLookup">
            …
        </connector>
        <dispatcherParameter
                …
        </dispatcherParameter>
        <assemblyLine>LDAPGroupDelete.xml</assemblyLine>
</operation>

<operation cn="LDAPGroupModify">
        <name>modify-erLdapGroupAccount</name>
        <description>
            group modify operations are supported by AddAssemblyLine
        </description>
        <connector name="conLDAPGroup">
            …
        </connector>
        <connector name="conLDAPGroupLookup">
            …
        </connector>
        <dispatcherParameter
                …
        </dispatcherParameter>

        <assemblyLine>LDAPGroupModify.xml</assemblyLine>
</operation>
```

Customize the group form

Your initial profile with group management will not include the group form since you need the TIM "Design Forms" to customize the feel and look of the form. Since the group form will not exist in your initial profile jar file, TIM will display a generic form using all the attributes in the group object class.

Once the profile is imported, use the "Design Forms" in TIM to change it. Using the view source feature in the TIM "Design Forms", copy the content of the updated for and saved in a file. By convention, the name of the file should <group object class name>.xml. For the LDAP adapter, the file name would be erLDAPGroupAccount.xml.

As you recall, the service group definition in service.def contains the following line to indicate the form file name:

```
<form location="erLDAPGroupAccount.xml" />
```

Modify the search (reconciliation) assembly line to return new updated object

As we mentioned at the beginning, service groups are supporting data that can be managed. If you are modifying an existing adapter to support service groups management, then you must update the search assembly line to return the additional attributes that were added to the group object class.

Note that the object returned by the adapter assembly line must have ***rdnAttribute*** as the naming attribute for each object. In our example erLdapGroupServiceName is the naming attribute as defined in the service group definition in service.def.

Additional processing specifically for LDAP groups

The LDAP object class "groupOfNames" has two required attributes: cn and member. From TIM perspective, we can't force adding members to groups during group creation. To resolve this issue, you have two options:

- Hardcode the group add assembly line to always use "cn=dummy" as value for the member attribute.

OR

- Create a service object attribute called initial member (erLdapInitialGrpMem). This service attribute will be required when you create a service and the value will be sent to the group add assembly line as a dispatcherParameter:

```
<operation cn="LDAPGroupAdd">
            <name>add-erLdapGroupAccount</name>
            <description>
                group add operations are supported by AddAssemblyLine
            </description>
        <connector name="conLDAPGroup">
                ...
        </connector>
```

```
			<connector name="conLDAPGroupLookup">
				…
			</connector>
			<dispatcherParameter name="LdapInitialGrpMem">
				<default>$(SO!erLdapInitialGrpMem)</default>
			</dispatcherParameter>
			<assemblyLine>LDAPGroupAdd.xml</assemblyLine>
		</operation>
```

Finalizing the adapter profile

Prior to service group management, the LDAP adapter profile consisted of the following files:

- CustomLabels.properties
- erLDAPRMIService.xml
- erLDAPUserAccount.xml
- LDAPAdd.xml
- LdapAL.xml
- LDAPDelete.xml
- LDAPModify.xml
- LDAPSearch.xml
- LDAPTest.xml
- schema.dsml
- service.def

After adding service group management support, the following files have been added:

- erLDAPGroupAccount.xml …. (group form)
- LDAPGroupAdd.xml …………… (add groups assembly line)
- LDAPGroupDelete.xml ………. (delete groups assembly line)
- LDAPGroupModify.xml ……… (modify groups assembly line)

After adding service group management support, the following files have been updated:

- CustomLabels.properties ……… (added new groups labels)
- erLDAPRMIService.xml ………… (added new initial member attribute).
- LDAPSearch.xml …………………… (added reconciliation support of new/updated group object class)
- schema.dsml ……………………….. (added new group attributes and create/update group object class)
- service.def …………………………… (added/updated service group definition; added group operations)

Appendix A: Example schema.dsml

```
 <?xml version="1.0" encoding="UTF-8"?>
 <dsml>
 <directory-schema>

 <!-- ************************************************* -->
 <!—erCustomLdapGroupName -->
 <!-- ************************************************* -->
 <attribute-type single-value = "false" >
 <name> erCustomLdapGroupName </name>
     <description></description>
 <object-identifier> erCustomLdapGroupName -OID</object-identifier>
     <syntax>1.3.6.1.4.1.1466.115.121.1.15</syntax>
 </attribute-type>

 <!-- *************************************************-->
 <!-- erCustomLDAPUserAccount Class -->
 <!-- **************************************************-->
 <class superior="inetorgperson">
       <name> erCustomLDAPUserAccount </name>
 <description>Class representing LDAP RMI account</description>
     <object-identifier> erCustomLDAPUserAccount -OID</object-identifier>
     <attribute ref="eruid" required="true" />
     <attribute ref="erpassword" required="false" />
     <attribute ref="erLdapGroupName" required="false" />
     <attribute ref="erAccountStatus" required="false" />
 </class>

 <!-- *************************************************-->
 <!-- erCustomLdapGroupAccount Class     -->
 <!-- *************************************************-->

 <class superior="top">
       <name> erCustomLdapGroupAccount </name>
     <description>Class representingLDAP group</description>
 <object-identifier> erCustomLdapGroupAccount - OID</object-identifier>
 <attribute ref = " erCustomLdapGroupName " required = "true" />
 </class>


 <!-- **************************************************-->
 <!-- erCustomLDAPRMIService Class      -->
 <!-- ************************************************* -->

 <class superior="top">
 <name> erCustomLDAPRMIService </name>
 <description>Class representing a LDAP RMI service</description>

 <object-identifier> erCustomLDAPRMIService -OID</object-identifier>
     <attribute ref = "erServiceName" required = "true" />
       <attribute ref = "description" required = "false" />
 <!-- The Tivoli Directory Integrator location. The provider will use the
default        -->
 <!-- value from service.def unless it's provided on the form -->

 <attribute ref = "erITDIurl" required = "false" />
 <!-- erURL: the managed resource location.    -->
```

```
<attribute ref = "erURL" required = "true" />
<!-- erServiceUid: the managed resource account.-->

<attribute ref = "erServiceUid" required = "true" />
<!-- erServicePassword: the managed resource account password -->

<attribute ref = "erPassword" required = "true" />
</class>
</directory-schema>
</dsml>
```

Appendix B: Example service.def

Sample service.def describing "add" and "search" assembly lines.

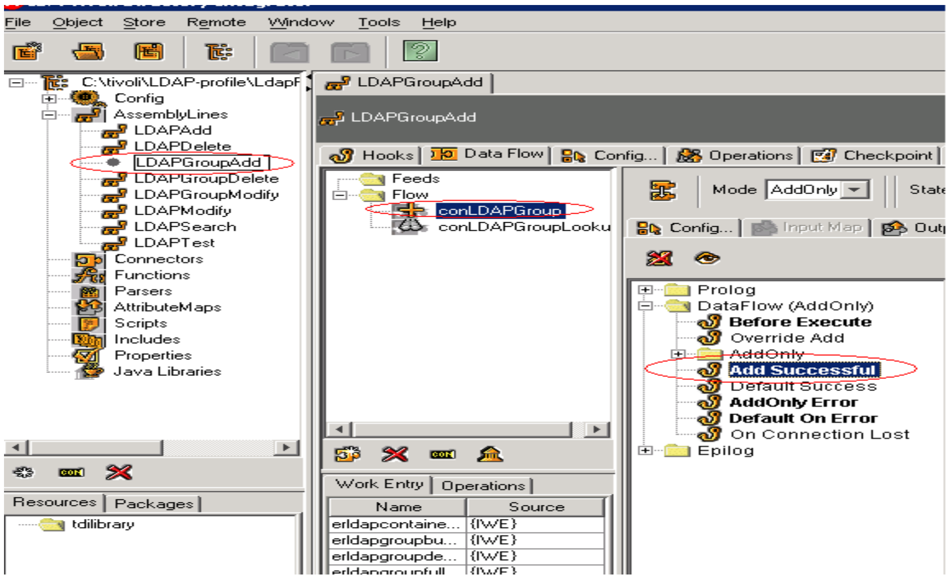
```
<?xml version="1.0" encoding="UTF-8"?>
<Service
erserviceproviderfactory="com.ibm.itim.remoteservices.provider.itdiprovider.Itd
iServiceProviderFactory"
name="CustomLdapProfile" xmlns:svc="urn:com:ibm:itim:service:1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:com:ibm:itim:service:1 service.xsd ">

<type name="erCustomLDAPRMIService" category="service"
location="schema.dsml" profile="CustomLdapProfile">
       <key>
         <field>erservicename</field>
       </key>
    <form location="erCustomLDAPRMIService.xml" />
</type>


<type name="erCustomLDAPUserAccount" category="account"
    location="schema.dsml" profile="CustomLdapAccount">
       <key>
             <field>eruid</field>
    </key>
    <form location="erCustomLDAPUserAccount.xml" />
</type>

<operation cn="LDAPAdd">
<name>add</name>
<description> add operations are supported by AddAssemblyLine </description>

<input name="eruid" source="eruid">
</input>
<!--Naming attribute of the target system and corresponding attribute in the
account object-->

<connector name="conLDAPUser">
<parameter name="ldapUrl" source="erURL">The LDAP Sever URL
</parameter>

<parameter name="ldapUsername" source="erServiceUid">The LDAP root username is
set from erServiceUid (from service class)
</parameter>

<parameter name="ldapPassword" source="erPassword">The LDAP root password
</parameter>

<dispatcherParameter name="poolID">
<default>ldapUserConnector/$(SO!erURL)/$(SO!erServiceUid)/$(SO!erPassword)</def
ault>
          </dispatcherParameter>

</connector>

<connector name="conLDAPMembership">
<parameter name="ldapUrl" source="erURL">The LDAP Sever URL
</parameter>
```

```
<parameter name="ldapUsername" source="erServiceUid">The LDAP root username is
set from erServiceUid (from service class)
</parameter>

<parameter name="ldapPassword" source="erPassword">The LDAP root password
</parameter>

<dispatcherParameter name="poolID">
<default>ldapMbrConnector/$(SO!erURL)/$(SO!erServiceUid)/$(SO!erPassword)</defa
ult>
    </dispatcherParameter>
</connector>

<dispatcherParameter name="userContainerDN">
<default>$(SO!erusercontainerdn)</default>
User Container DN
</dispatcherParameter>

<dispatcherParameter name="groupsContainerDN">
<default>$(SO!ergroupscontainerdn)</default>
Group Container DN
</dispatcherParameter>

<dispatcherParameter name="DSName" source="erDSName">
<default>SUN_ONE</default> Directory Server Name
</dispatcherParameter>


<assemblyLine>CustomLDAPAdd.xml</assemblyLine>
</operation>

<operation cn="LDAPSearch">
<name>search</name>
<description> operation recon supported by SearchAssemblyLine </description>

<input name="eruid" source="eruid">
<!--Naming attribute of the target system and corresponding attribute in the
account object-->
</input>

<connector name="conLDAPUser">
<parameter name="ldapUrl" source="erURL">The LDAP Sever URL
</parameter>

<parameter name="ldapUsername" source="erServiceUid">The LDAP root username is
set from erUid (from service class)
</parameter>

<parameter name="ldapPassword" source="erPassword">The LDAP root password
</parameter>

<dispatcherParameter name="poolID">
<default>ldapUserConnector/$(SO!erURL)/$(SO!erServiceUid)/$(SO!erPassword)</def
ault>
</dispatcherParameter>
</connector>

<connector name="conLDAPMembership">
<parameter name="ldapUrl" source="erURL">The LDAP Sever URL
</parameter>

<parameter name="ldapUsername" source="erServiceUid">The LDAP root username is
set from erUid (from service class)
</parameter>
```

```
<parameter name="ldapPassword" source="erPassword">The LDAP root password
</parameter>

<dispatcherParameter name="poolID">
<default>ldapMbrConnector/$(SO!erURL)/$(SO!erServiceUid)/$(SO!erPassword)</defa
ult>
</dispatcherParameter>
</connector>

<connector name="conLDAPGroup">
<parameter name="ldapUrl" source="erURL">The LDAP Sever URL</parameter>

<parameter name="ldapUsername" source="erServiceUid">The LDAP root username is
set from erUid (from service class)</parameter>

<parameter name="ldapPassword" source="erPassword">The LDAP root
password</parameter>

<dispatcherParameter name="poolID">
<default>ldapGrpConnector/$(SO!erURL)/$(SO!erServiceUid)/$(SO!erPassword)</defa
ult>
</dispatcherParameter>
</connector>

<dispatcherParameter name="userContainerDN">
<default>$(SO!erusercontainerdn)</default>
User Container DN
</dispatcherParameter>

<dispatcherParameter name="groupsContainerDN">
<default>$(SO!ergroupscontainerdn)</default>
Group Container DN
</dispatcherParameter>

<dispatcherParameter name="DSName" source="erDSName">
<default>SUN_ONE</default>
Directory Server Name
</dispatcherParameter>

<dispatcherParameter name="ldapSearchFilter">
<default>$(OO!searchFilter)</default>
</dispatcherParameter>
<assemblyLine>CustomLDAPSearch.xml</assemblyLine>
</operation>
</Service>
```

Appendix C: Add a hook for groups "Add Successful"

Add a hook for groups "Add Successful":



Name	Source
erldapcontaine...	{IWE}
erldapgroupbu...	{IWE}
erldapgroupde...	{IWE}
erldapgroupfull	{IWE}

Content of the "Add Successful" hook to return all group attributes back to TIM in the success status:

```
// sending implicit attributes to TIM
var groupName = work.getString("erLdapServiceGroup");
work.removeAllAttributes();

// Setting Lookup Entry that will be passed to Lookup Connector
var lookupEntry = system.newEntry();
lookupEntry.newAttribute("erLdapServiceGroup");
lookupEntry.setAttribute("erLdapServiceGroup",groupName);
lookupEntry.newAttribute("objectclass");
lookupEntry.setAttribute("objectclass","erLdapGroupAccount");

// Calling Group connector in lookup mode
conLDAPGroupLookup.lookup(lookupEntry);

// Copying all the attributes from lookup entry to retrun entry
var retWorkAttribute = "";
var returnAttributes = lookupEntry.getAttributeNames();
for (i=0; i<returnAttributes.length; i++) {
    var retLookupAttr = lookupEntry.getAttribute(returnAttributes[i]);
    var lookupAttrName = retLookupAttr.getName();
    work.newAttribute(lookupAttrName);
    var name = work.getAttributeNames();

    retWorkAttribute = work.getAttribute(lookupAttrName);
    for (j=0;j<retLookupAttr.size();j++ ) {
            retWorkAttribute.addValue(retLookupAttr.getValue(j));
            retWorkAttribute.setOper(Packages.com.ibm.di.entry.Attribute.ATTRIBUTE_ADD);
            retWorkAttribute.setValueOper(j,Packages.com.ibm.di.entry.AttributeValue.AV_ADD);
    }
}

// Setting status in work entry
work.setProperty(  Packages.com.ibm.di.dispatcher.Defs.STATUSCODE,
            new Packages.java.lang.Integer(Packages.com.ibm.itim.remoteservices.provider.Status.SUCCESSFUL));
```

End of document.